Security and Incremental Computation

Robustly Safe Compilation

Marco Patrignani[1,2(✉)] and Deepak Garg[3]

[1] Stanford University, Stanford, USA
mp@cs.stanford.edu
[2] CISPA Helmholz Center for Information Security, Saarbrücken, Germany
[3] Max Planck Institute for Software Systems, Saarbrücken, Germany

Abstract. Secure compilers generate compiled code that withstands many target-level attacks such as alteration of control flow, data leaks or memory corruption. Many existing secure compilers are proven to be fully abstract, meaning that they reflect and preserve observational equivalence. Fully abstract compilation is strong and useful but, in certain cases, comes at the cost of requiring expensive runtime constructs in compiled code. These constructs may have no relevance for security, but are needed to accommodate differences between the source and target languages that fully abstract compilation necessarily needs.

As an alternative to fully abstract compilation, this paper explores a different criterion for secure compilation called robustly safe compilation or *RSC*. Briefly, this criterion means that the compiled code preserves relevant safety properties of the source program against all adversarial contexts interacting with the compiled program. We show that *RSC* can be proved more easily than fully abstract compilation and also often results in more efficient code. We also develop two illustrative robustly-safe compilers and, through them, illustrate two different proof techniques for establishing that a compiler attains *RSC*. Based on these, we argue that proving *RSC* can be simpler than proving fully abstraction.

To better explain and clarify notions, this paper uses colours. For a better experience, please print or view this paper in colours.[1]

1 Introduction

Low-level adversaries, such as those written in C or assembly can attack co-linked code written in a high-level language in ways that may not be feasible in the high-level language itself. For example, such an adversary may manipulate or hijack control flow, cause buffer overflows, or directly access private memory,

[1] Specifically, in this paper we use a blue, sans-serif font for source elements, an **orange**, **bold** font for **target** elements and a *black*, *italic* font for elements common to both languages (to avoid repeating similar definitions twice). Thus, C is a source-level component, **C** is a target-level component and C is generic notation for either a source-level or a target-level component.

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 469–498, 2019.
https://doi.org/10.1007/978-3-030-17184-1_17

all in contravention to the abstractions of the high-level language. Specific countermeasures such as Control Flow Integrity [3] or Code Pointer Integrity [41] have been devised to address some of these attacks *individually*. An alternative approach is to devise a *secure compiler*, which seeks to defend against entire *classes* of such attacks. Secure compilers often achieve security by relying on different protection mechanisms, e.g., cryptographic primitives [4,5,22,26], types [10,11], address space layout randomisation [6,37], protected module architectures [9,53,57,59] (also know as enclaves [46]), tagged architectures [7,39], etc. Once designed, the question researchers face is how to formalise that such a compiler is indeed secure, and how to prove this. Basically, we want a criterion that specifies secure compilation. A widely-used criterion for compiler security is fully abstract compilation (*FAC*) [2,35,52], which has been shown to preserve many interesting security properties like confidentiality, integrity, invariant definitions, well-bracketed control flow and hiding of local state [9,37,53,54].

Informally, a compiler is fully abstract if it preserves and reflects observational equivalence of source-level components (i.e., partial programs) in their compiled counterparts. Most existing work instantiates observational equivalence with contextual equivalence: co-divergence of two components in any larger context they interact with. Fully abstract compilation is a very strong property, which preserves *all* source-level abstractions.

Unfortunately, preserving *all* source-level abstractions also has downsides. In fact, while *FAC* preserves many relevant security properties, it also preserves a plethora of other non-security ones, and the latter may force inefficient checks in the compiled code. For example, when the target is assembly, two observationally equivalent components must compile to code of the same size [9,53], else full abstraction is trivially violated. This requirement is security-irrelevant in most cases. Additionally, *FAC* is not well-suited for source languages with undefined behaviour (e.g., `C` and `LLVM`) [39] and, if used naïvely, it can fail to preserve even simple safety properties [60] (though, fortunately, no *existing* work falls prey to this naïvety).

Motivated by this, recent work started investigating alternative secure compilation criteria that overcome these limitations. These security-focussed criteria take the form of preservation of hyperproperties or classes of hyperproperties, such as hypersafety properties or safety properties [8,33]. This paper investigates one of these criteria, namely, *Robustly Safe Compilation* (*RSC*) which has clear security guarantees and can often be attained more efficiently than FAC.

Informally, a compiler attains *RSC* if it is correct and it preserves *robust safety* of source components in the target components it produces. Robust safety is an important security notion that has been widely adopted to formalize security, e.g., of communication protocols [14,17,34]. Before explaining *RSC*, we explain robust safety as a language property.

Robust Safety as a Language Property. Informally, a program property is a safety property if it encodes that "bad" sequences of events do not happen when the program executes [13,63]. A program is *robustly safe* if it has relevant (specified)

safety properties *despite* active attacks from adversaries. As the name suggests, robust safety relies on the notions of safety and robustness which we now explain.

Safety. As mentioned, safety asserts that "no bad sequence of events happens", so we can specify a safety property by the set of *finite observations* which characterise all bad sequences of events. A whole program has a safety property if its behaviours exclude these bad observations. Many security properties can be encoded as safety, including integrity, weak secrecy and functional correctness.

Example 1 (Integrity). Integrity ensures that an attacker does not tamper with code invariants on state. For example, consider the function `charge_account(n)` which deducts amount `n` from an account as part of an electronic card payment. A card PIN is required if `n` is larger than 10 euros. So the function checks whether `n` > 10, requests the PIN if this is the case, and then changes the account balance. We expect this function to have a safety (integrity) property in the account balance: A reduction of more than 10 euros in the account balance must be preceded by a call to `request_pin()`. Here, the relevant observation is a trace (sequence) of account balances and calls to `request_pin()`. Bad observations for this safety property are those where an account balance is at least 10 euros less than the previous one, without a call to `request_pin()` in between. Note that this function seems to have this safety property, but it may not have the safety property *robustly*: a target-level adversary may transfer control directly to the "else" branch of the check `n` > 10 after setting `n` to more than 10, to violate the safety property. ⊡

Example 2 (Weak Secrecy). Weak secrecy asserts that a program secret never flows *explicitly* to the attacker. For example, consider code that manages `network_h`, a handler (socket descriptor) for a sensitive network interface. This code does not expose `network_h` directly to external code but it provides an API to use it. This API makes some security checks internally. If the handler is directly accessible to outer code, then it can be misused in insecure ways (since the security checks may not be made). If the code has weak secrecy wrt `network_h` then we know that the handler is never passed to an attacker. In this case we can define bad observations as those where `network_h` is passed to external code (e.g., as a parameter, as a return value on or on the heap). ⊡

Example 3 (Correctness). Program correctness can also be formalized as a safety property. Consider a program that computes the `n`th Fibonacci number. The program reads `n` from an input source and writes its output to an output source. Correctness of this program is a safety property. Our observations are pairs of an input (read by the program) and the corresponding output. A bad observation is one where the input is `n` (for some `n`) but the output is different from the `n`th Fibonacci number. ⊡

These examples not only illustrate the expressiveness of safety properties, but also show that safety properties are quite *coarse-grained*: they are only concerned with (sequences of) relevant events like calls to specific functions, changes to

specific heap variables, inputs, and outputs. They do not specify or constrain how the program computes between these events, leaving the programmer and the compiler considerable flexibility in optimizations. However, safety properties are not a panacea for security, and there are security properties that are not safety. For example, noninterference [70,72], the standard information flow property, is not safety. Nonetheless, many interesting security properties are safety. In fact, many non-safety properties including noninterference can be conservatively approximated as safety properties [20]. Hence, safety properties are a meaningful goal to pursue for secure compilation.

Robustness. We often want to reason about properties of a component of interest that hold irrespective of any other components the component interacts with. These other components may be the libraries the component is linked against, or the language runtime. Often, these surrounding components are modelled as the *program context* whose hole the component of interest fills. From a security perspective the context represents the attacker in the threat model. When the component of interest links to a context, we have a whole program that can run. A property holds *robustly* for a component if it holds in *any* context that the component of interest can be linked to.

Robust Safety Preservation as a Compiler Property. A compiler attains robustly safe compilation or *RSC* if it maps any source component that has a safety property *robustly* to a compiled component that has the *same* safety property robustly. Thus, safety has to hold robustly in the target language, which often does not have the powerful abstractions (e.g., typing) that the source language has. Hence, the compiler must insert enough defensive runtime checks into the compiled code to prevent the more powerful target contexts from launching attacks (violations of safety properties) that source contexts could not launch. This is unlike correct compilation, which either considers only those target contexts that behave like source contexts [40,49,65] or considers only whole programs [43].

As mentioned, safety properties are usually quite coarse-grained. This means that *RSC* still allows the compiler to optimise code internally, as long as the sequence of observable events is not affected. For example, when compiling the `fibonacci` function of Example 3, the compiler can do any internal optimisation such as caching intermediate results, as long as the end result is correct. Crucially, however, these intermediate results must be protected from tampering by a (target-level) attacker, else the output can be incorrect, breaking *RSC*.

A *RSC*-attaining compiler focuses only on preserving security (as captured by robust safety) instead of contextual equivalence (typically captured by full abstraction). So, such a compiler can produce code that is more efficient than code compiled with a fully abstract compiler as it does not have to preserve *all* source abstractions (we illustrate this later).

Finally, robust safety scales naturally to thread-based concurrency [1,34,58]. Thus *RSC* also scales naturally to thread-based concurrency (we demonstrate

this too). This is unlike *FAC*, where thread-based concurrency can introduce additional undesired abstractions that also need to be preserved.

RSC is a very recently proposed criterion for secure compilers. Recent work [8,33] define *RSC* abstractly in terms of preservation of program behaviours, but their development is limited to the definition only. Our goal in this paper is to examine how *RSC* can be realized and established, and to show that in certain cases it leads to compiled code that is more efficient than what *FAC* leads to. To this end, we consider a specific setting where observations are values in specific (sensitive) heap locations at cross-component calls. We define robust safety and *RSC* for this specific setting (Sect. 2). Unlike previous work [8,33] which assumed that the domain of traces (behaviours) is the same in the source and target languages, our *RSC* definition allows for different trace domains in the source and target languages, as long as they can be suitably related. The second contribution of our paper is two proof techniques to establish *RSC*.

- The first technique is an adaption of trace-based backtranslation, an existing technique for proving *FAC* [7,9,59]. To illustrate this technique, we build a compiler from an untyped source language to an untyped target language with support for fine-grained memory protection via so-called capabilities [23,71] (Sect. 3). Here, we guarantee that if a source program is robustly safe, then so is its compilation.
- The second proof technique shows that if source programs are *verified* for robust safety, then one can simplify the proof of *RSC* so that no backtranslation is needed. In this case, we develop a compiler from a *typed* source language where the types already enforce robust safety, to a target language similar to that of the first compiler (Sect. 4). In this instance, both languages also support shared-memory concurrency. Here, we guarantee that all compiled target programs are robustly safe.

To argue that *RSC* is general and is not limited to compilation targets based on capabilities, we also develop a third compiler. This compiler starts from the same source language as our second compiler but targets an untyped concurrent language with support for *coarse-grained memory isolation*, modelling recent hardware extensions such as Intel's SGX [46]. Due to space constraints, we report this result only in the companion technical report [61].

The final contribution of this paper is a comparison between *RSC* and *FAC*. For this, we describe changes that would be needed to attain *FAC* for the first compiler and argue that these changes make generated code inefficient and also complicate the backtranslation proof significantly (Sect. 5).

Due to space constraints, we elide some technical details and limit proofs to sketches. These are fully resolved in the companion technical report [61].

2 Robustly Safe Compilation

This section first discusses robust safety as a language (not a compiler) property (Sect. 2.1) and then presents *RSC* as a compiler property along with an informal discussion of techniques to prove it (Sect. 2.2).

2.1 Safety and Robust Safety

To explain robust safety, we first describe a general *imperative* programming model that we use. Programmers write *components* on which they want to enforce safety properties robustly. A component is a list of function definitions that can be linked with other components (the context) in order to have a runnable whole program (functions in "other" components are like `extern` functions in C). Additionally, every component declares a set of "sensitive" locations that contain all the data that is safety-relevant. For instance, in Example 1 this set may contain the account balance and in Example 3 it may contain the I/O buffers. We explain the relevance of this set after we define safety properties.

We want safety properties to specify that a component never executes a "bad" sequence of events. For this, we first need to fix a notion of events. We have several choices here, e.g., our events could be inputs and outputs, all syscalls, all changes to the heap (as in CompCert [44]), etc. Here, we make a specific choice motivated by our interest in robustness: We define events as calls/returns that cross a component boundary, together with the state of the heap at that point. Consequently, our safety properties can constrain the contents of the heap at component boundaries. This choice of component boundaries as the point of observation is meaningful because, in our programming model, control transfers to/from an adversary happen only at component boundaries (more precisely, they happen at cross-component function call and returns). This allows the compiler complete flexibility in optimizing code within a component, while not reducing the ability of safety properties to constrain observations of the adversary.

Concretely, a component behaviour is a *trace*, i.e., a sequence of *actions* recording component boundary interactions and, in particular, the heap at these points. *Actions*, the items on a trace, have the following grammar:

$$\textit{Actions}\ \alpha ::= \texttt{call}\ f\ v\ H? \mid \texttt{call}\ f\ v\ H! \mid \texttt{ret}\ H! \mid \texttt{ret}\ H?$$

These actions respectively capture call and callback to a function f with parameter v when the heap is H as well as return and returnback with a certain heap H.[2] We use ? and ! decorations to indicate whether the control flow of the action goes from the context to the component (?) or from the component to the context (!). Well-formed traces have alternations of ? and ! decorated actions,

[2] A callback is a call from the component to the context, so it generates label `call` f v H!. A returnback is a return from such a callback, i.e., the context returning to the component, and it generates the label `ret` H?.

starting with ? since execution starts in the context. For a sequence of actions $\overline{\alpha}$, $\texttt{relevant}(\overline{\alpha})$ is the list of heaps $\overline{H}$ mentioned in the actions of $\overline{\alpha}$.

Next, we need a representation of safety properties. Generally, properties are sets of traces, but safety properties specifically can be specified as automata (or monitors in the sequel) [63]. We choose this representation since monitors are less abstract than sets of traces and they are closer to enforcement mechanisms used for safety properties, e.g., runtime monitors. Briefly, a safety property is a monitor that transitions states in response to events of the program trace. At any point, the monitor may refuse to transition (it gets *stuck*), which encodes property violation. While a monitor can transition, the property has not been violated. Schneider [63] argues that all properties codable this way are safety properties and that all enforceable safety properties can be coded this way.

Formally, a monitor M in our setting consists of a set of abstract states $\{\sigma \cdots\}$, the transition relation $\rightsquigarrow$, an initial state σ_0, the set of heap locations that matter for the monitor, $\{l \cdots\}$, and the current state σ_c (we indicate a set of elements of class e as $\{e \cdots\}$). The transition relation $\rightsquigarrow$ is a set of triples of the form (σ_s, H, σ_f) consisting of a starting state σ_s, a final state σ_f and a heap H. The transition (σ_s, H, σ_f) is interpreted as "*state* σ_s *transitions to* σ_f *when the heap is* H". When determining the monitor transition in response to a program action, we restrict the program's heap to the location set $\{l \cdots\}$, i.e., to the set of locations the monitor cares about. This heap restriction is written $H\big|_{\{l \cdots\}}$. We assume determinism of the transition relation: for any σ_s and (restricted heap) H, there is at most one σ_f such that $(\sigma_s, H, \sigma_f) \in \rightsquigarrow$.

Given the behaviour of a program as a trace $\overline{\alpha}$ and a monitor M specifying a safety property, $M \vdash \overline{\alpha}$ denotes that the trace satisfies the safety property. Intuitively, to satisfy a safety property, the sequence of heaps in the actions of a trace must never get the monitor stuck (Rule Valid trace). Every single heap must allow the monitor to step according to its transition relation (Rule Monitor Step). Note that we overload the $\rightsquigarrow$ notation here to also denote an auxiliary relation, the *monitor small-step semantics* (Rule Monitor Step-base and Rule Monitor Step-ind).

$$\frac{M; \texttt{relevant}(\overline{\alpha}) \rightsquigarrow M'}{M \vdash \overline{\alpha}}\ \text{(Valid trace)} \qquad \frac{}{M; \varnothing \rightsquigarrow M}\ \text{(Monitor Step-base)} \qquad \frac{M; \overline{H} \rightsquigarrow M'' \quad M''; H \rightsquigarrow M'}{M; \overline{H} \cdot H \rightsquigarrow M'}\ \text{(Monitor Step-ind)}$$

$$\frac{(\sigma_c, H\big|_{\{l \cdots\}}, \sigma_f) \in \rightsquigarrow}{(\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \{l \cdots\}, \sigma_c); H \rightsquigarrow (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \{l \cdots\}, \sigma_f)}\ \text{(Monitor Step)}$$

With this setup in place, we can formalise safety, attackers and robust safety. In defining (robust) safety for a component, we only admit monitors (safety properties) whose $\{l \cdots\}$ agrees with the sensitive locations declared by the component. Making the set of safety-relevant locations explicit in the component and the monitor gives the compiler more flexibility by telling it precisely which locations need to be protected against target-level attacks (the compiler may choose to not protect the rest). At the same time, it allows for expressive modelling. For instance, in Example 3 the safety-relevant locations could be the

I/O buffers from which the program performs inputs and outputs, and the safety property can constrain the input and output buffers at corresponding call and return actions involving the Fibonacci function.

Definition 1 (Safety, attacker and robust safety).

$$M \vdash C : \mathit{safe} \stackrel{\text{def}}{=} \mathit{if} \vdash C : \mathit{whole}\ \mathit{then}\ \mathit{if}\ \Omega_0(C) \stackrel{\overline{\alpha}}{\Longrightarrow} \_\ \mathit{then}\ M \vdash \overline{\alpha}$$

$$C \vdash A : \mathit{atk} \stackrel{\text{def}}{=} C = \{l \cdots\}, \overline{F} \text{ and } \{l \cdots\} \cap \mathtt{fn}(A) = \varnothing$$

$$M \vdash C : \mathit{rs} \stackrel{\text{def}}{=} \forall A.\ \mathit{if}\ M \frown C\ \mathit{and}\ C \vdash A : \mathit{atk}\ \mathit{then}\ M \vdash A[C] : \mathit{safe}$$

A whole program C is safe for a monitor M, written $M \vdash C : \mathit{safe}$, if the monitor accepts any trace the program generates from its initial state ($\Omega_0(C)$).

An attacker A is valid for a component C, written $C \vdash A : \mathit{atk}$, if A's free names (denoted $\mathtt{fn}(A)$) do not refer to the locations that the component cares about. This is a basic sanity check: if we allow an attacker to mention heap locations that the component cares about, the attacker will be able to modify those locations, causing all but trivial safety properties to not hold robustly.

A component C is robustly safe wrt monitor M, written $M \vdash C : \mathit{rs}$, if C composed with *any* attacker is safe wrt M. As mentioned, for this setup to make sense, the monitor and the component must agree on the locations that are safety-relevant. This agreement is denoted $M \frown C$.

2.2 Robustly Safe Compilation

Robustly-safe compilation ensures that robust safety properties *and their meanings* are preserved across compilation. But what does it means to preserve meanings across languages? If a source safety property says never write 3 to a location, and we compile to an assembly language by mapping numbers to binary, the corresponding target property should say **never write 0x11 to an address**.

In order to relate properties across languages, we assume a relation $\approx : \mathsf{v} \times \mathbf{v}$ between source and target values that is *total*, so it maps any source value v to a target value $\mathbf{v}$: $\forall \mathsf{v}.\exists \mathbf{v}.\mathsf{v} \approx \mathbf{v}$. This value relation is used to define a relation between heaps: $\mathsf{H} \approx \mathbf{H}$, which intuitively holds when related locations point to related values. This is then used to define a relation between actions: $\alpha \approx \alpha$, which holds when the two actions are the "same" modulo this relation, i.e., $\mathtt{call}\ \cdot\ \cdot\ \cdot ?$ only relates to $\mathtt{call}\ \cdot\ \cdot\ \cdot ?$ and the arguments of the action (values and heap) are related. Next, we require a relation $\mathsf{M} \approx \mathbf{M}$ between source and target monitors, which means that the source monitor M and the target monitor $\mathbf{M}$ code the same safety property, modulo the relation $\approx$ on values assumed above. The precise definition of this relation depends on the source and target languages; specific instances are shown in Sects. 3.3 and 4.3.[3]

[3] Accounting for the difference in the representation of safety properties sets us apart from recent work [8,33], which assumes that the source and target languages have the same trace alphabet. The latter works only in some settings.

We denote a compiler from language S to language $\mathbf{T}$ by $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$. A compiler $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ attains *RSC*, if it maps any component C that is robustly safe wrt M to a component $\mathbf{C}$ that is robustly safe wrt $\mathbf{M}$, provided that $\mathsf{M} \approx \mathbf{M}$.

Definition 2 (Robustly Safe Compilation).

$$\vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathit{RSC} \stackrel{\text{def}}{=} \forall \mathsf{C}, \mathsf{M}, \mathbf{M}.\ \mathit{if}\ \mathsf{M} \vdash \mathsf{C} : \mathsf{rs}\ \mathit{and}\ \mathsf{M} \approx \mathbf{M}\ \mathit{then}\ \mathbf{M} \vdash [\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathbf{rs}$$

A consequence of the universal quantification over monitors here is that the compiler cannot be property-sensitive. A robustly-safe compiler preserves all robust safety properties, not just a specific one, e.g., it does not just enforce that `fibonacci` is correct. This seemingly strong goal is sensible as compiler writers will likely not know what safety properties individual programmers will want to preserve.

Remark. Some readers may wonder why we do not follow existing work and specify safety as "programmer-written assertions never fail" [31,34,45,68]. Unfortunately, this approach does not yield a meaningful criterion for specifying a compiler, since assertions in the compiled program (if any) are generated by the compiler itself. Thus a compiler could just erase all assertions and the compiled code it generates would be trivially (robustly) safe – no assertion can fail if there are no assertions in the first place!

Proving *RSC*. Proving that a compiler attains *RSC* can be done either by proving that a compiler satisfies Definition 2 or by proving something *equivalent*. To this end, Definition 3 below presents an alternative, equivalent formulation of *RSC*. We call this characterisation *property-free* as it does not mention monitors explicitly (it mentions the $\mathtt{relevant}(\cdot)$ function for reasons we explain below).

Definition 3 (Property-Free *RSC*).

$$\begin{aligned}
&\vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathit{PF\text{-}RSC} \stackrel{\text{def}}{=} \forall \mathsf{C}, \mathbf{A}, \overline{\alpha}.\\
&\quad \mathit{if}\ \ [\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}} \vdash \mathbf{A} : \mathbf{atk}\ \mathit{and}\ \vdash \mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right] : \mathbf{whole}\ \mathit{and}\ \boldsymbol{\Omega_0}\left(\mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right]\right) \stackrel{\overline{\alpha}}{\Longrightarrow} _\\
&\quad \mathit{then}\ \ \exists \mathsf{A}, \overline{\alpha}.\ \mathsf{C} \vdash \mathsf{A} : \mathsf{atk}\ \mathit{and}\ \vdash \mathsf{A}[\mathsf{C}] : \mathsf{whole}\ \mathit{and}\ \Omega_0(\mathsf{A}[\mathsf{C}]) \stackrel{\overline{\alpha}}{\Longrightarrow} _\\
&\qquad \mathit{and}\ \mathtt{relevant}(\overline{\alpha}) \approx \mathtt{relevant}(\overline{\alpha})
\end{aligned}$$

Specifically, *PF-RSC* states that the compiled code produces behaviours that *refine* source level behaviours *robustly* (taking contexts into account).

PF-RSC and *RSC* should, in general, be equivalent (Proposition 1).

Proposition 1 (*PF-RSC* and *RSC* are equivalent).

$$\forall [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}, \vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathit{PF\text{-}RSC} \iff \vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathit{RSC}$$

Informally, a property is safety if and only if it implies programs not having any trace prefix from a given set of bad prefixes (i.e., finite traces). Hence, *not* having

a safety property robustly amounts to some context being able to induce a bad prefix. Consequently, preserving *all* robust safety properties (*RSC*) amounts to ensuring that all target prefixes can be generated (by some context) in the source too (*PF-RSC*). Formally, since Definition 2 relies on the monitor relation, we can prove Proposition 1 only after such a relation is finalised. We give such a monitor relation and proof in Sect. 3.3 (see Theorem 3). However, in general this result should hold for any cross-language monitor relation that correctly relates safety properties. If the proposition does not hold, then the relation does not capture how safety in one language is represented in the other.

Assuming Proposition 1, we can prove *PF-RSC* for a compiler in place of *RSC*. *PF-RSC* can be proved with a *backtranslation* technique. This technique has been often used to prove full abstraction [7–9,33,39,50,53,54,59] and it aims at building a source context starting from a target one. In fact *PF-RSC*, leads directly to a backtranslation-based proof technique since it can be rewritten (eliding irrelevant details) as:

$$\text{If } \exists \mathbf{A}, \overline{\alpha}.\boldsymbol{\Omega_0}\left(\mathbf{A}\left[\llbracket \mathsf{C} \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]\right) \overset{\overline{\alpha}}{\Longrightarrow} \_$$

$$\text{then } \exists \mathsf{A}, \overline{\alpha}.\Omega_0\left(\mathsf{A}\left[\mathsf{C}\right]\right) \overset{\overline{\alpha}}{\Longrightarrow} \_ \text{ and } \mathtt{relevant}(\overline{\alpha}) \approx \mathtt{relevant}(\overline{\alpha})$$

Essentially, given a target context $\mathbf{A}$, a compiled program $\llbracket \mathsf{C} \rrbracket^{\mathsf{S}}_{\mathbf{T}}$ and a target trace $\overline{\alpha}$ that $\mathbf{A}$ causes $\llbracket \mathsf{C} \rrbracket^{\mathsf{S}}_{\mathbf{T}}$ to have, we need to construct, or *backtranslate* to, a source context A that will cause the source program C to simulate $\overline{\alpha}$. Such backtranslation based proofs can be quite difficult, depending on the features of the languages and the compiler. However, backtranslation for *RSC* (as we show in Sect. 3.3) is not as complex as backtranslation for *FAC* (Sect. 5.2).

A simpler proof strategy is also viable for *RSC* when we compile only those source programs that have been *verified* to be robustly safe (e.g., using a type system). The idea is this: from the verification of the source program, we can find an invariant which is always maintained by the target code, and which, in turn, implies the robust safety of the target code. For example, if the safety property is that values in the heap always have their expected types, then the invariant can simply be that values in the target heap are always related to the source ones (which have their expected types). This is tantamount to proving type preservation in the target in the presence of an active adversary. This is harder than standard type preservation (because of the active adversary) but is still much easier than backtranslation as there is no need to map target constructs to source contexts syntactically. We illustrate this proof technique in Sect. 4.

***RSC* Implies Compiler Correctness.** As stated in Sect. 1, *RSC* implies (a form of) compiler correctness. While this may not be apparent from Definition 2, it is more apparent from its equivalent characterization in Definition 3. We elaborate this here.

Whether concerned with whole programs or partial programs, compiler correctness states that the behaviour of compiled programs *refines* the behaviour of source programs [18,36,40,44,49,65]. So, if $\{\overline{\alpha}\cdots\}$ and $\{\overline{\alpha}\cdots\}$ are the sets of

compiled and source behaviours, then a compiler should force $\{\overline{\alpha}\cdots\} \precsim \{\overline{\alpha}\cdots\}$, where $\precsim$ is the composition of $\subseteq$ and of the relation $\approx^{-1}$.

If we consider a source component C that is whole, then it can only link against empty contexts, both in the source and in the target. Hence, in this special case, *PF-RSC* simplifies to standard refinement of traces, i.e., whole program compiler correctness. Hence, assuming that the correctness criterion for a compiler is concerned with the same observations as safety properties (values in safety-relevant heap locations at component crossings in our illustrative setting), *PF-RSC* implies whole program compiler correctness.

However, *PF-RSC* (or, equivalently, *RSC*) does not imply, nor is implied by, any form of *compositional compiler correctness* (CCC) [40,49,65]. CCC requires that the behaviours produced by a compiled component linked against a target context that is related (in behaviour) to a source context can also be produced by the source component linked against the *related* source context. In contrast, *PF-RSC* allows picking *any* source context to simulate the behaviours. Hence, *PF-RSC* does not imply CCC. On the other hand, *PF-RSC* universally quantifies over all target contexts, while CCC only quantifies over target contexts related to a source context, so CCC does not imply *PF-RSC* either. Hence, compositional compiler correctness, if desirable, must be imposed in addition to *PF-RSC*. Note that this lack of implications is unsurprising: *PF-RSC* and CCC capture two very different aspects of compilation: security (against all contexts) and compositional preservation of behaviour (against well-behaved contexts).

3 *RSC* via Trace-Based Backtranslation

This section illustrates how to prove that a compiler attains *RSC* by means of a trace-based backtranslation technique [7,53,59]. To present such a proof, we first introduce our source language L^{U}, an untyped, first-order imperative language with abstract references and hidden local state (Sect. 3.1). Then, we present our target language $\mathbf{L}^{\mathbf{P}}$, an untyped imperative target language with a concrete heap, whose locations are natural numbers that the context can compute. $\mathbf{L}^{\mathbf{P}}$ provides hidden local state via a fine-grained capability mechanism on heap accesses (Sect. 3.2). Finally, we present the compiler $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ and prove that it attains *RSC* (Sect. 3.3) by means of a trace-based backtranslation. The section conclude with an example detailing why *RSC* preserves security (Example 4).

To avoid focussing on mundane details, we deliberately use source and target languages that are fairly similar. However, they differ substantially in one key point: the heap model. This affords the target-level adversary attacks like guessing private locations and writing to them that do not obviously exist in the source (and makes our proofs nontrivial). We believe that (with due effort) the ideas here will generalize to languages with larger gaps and more features.

3.1 The Source Language L^{U}

L^{U} is an untyped imperative while language [51]. Components C are triples of function definitions, interfaces and a special location written ℓ_{root}, so C ::=

$\ell_{\mathsf{root}}; \overline{\mathsf{F}}; \overline{\mathsf{I}}$. Each function definition maps a function name and a formal argument to a body s: $\mathsf{F} ::= \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return};$. An interface is a list of functions that the component relies on the context to provide (similar to C's `extern` declarations). The special location ℓ_{root} defines the locations that are monitored for safety, as explained below. Attackers A (program contexts) are function definitions that represent untrusted code that a component interacts with. A function's body is a statement, s. Statements are rather standard, so we omit a formal syntax. Briefly, they can manipulate the heap (location creation $\mathsf{let\ x = new\ e\ in\ s}$, assignment $\mathsf{x := e}$), do recursive function calls ($\mathsf{call\ f\ e}$), condition (if-then-else), define local variables (let-in) and loop. Statements use effect-free expressions, e, which contain standard boolean expressions ($\mathsf{e} \otimes \mathsf{e}$), arithmetic expressions ($\mathsf{e} \oplus \mathsf{e}$), pairing ($\langle \mathsf{e}, \mathsf{e} \rangle$) and projections, and location dereference ($!\mathsf{e}$). Heaps H are maps from abstract locations ℓ to values v.

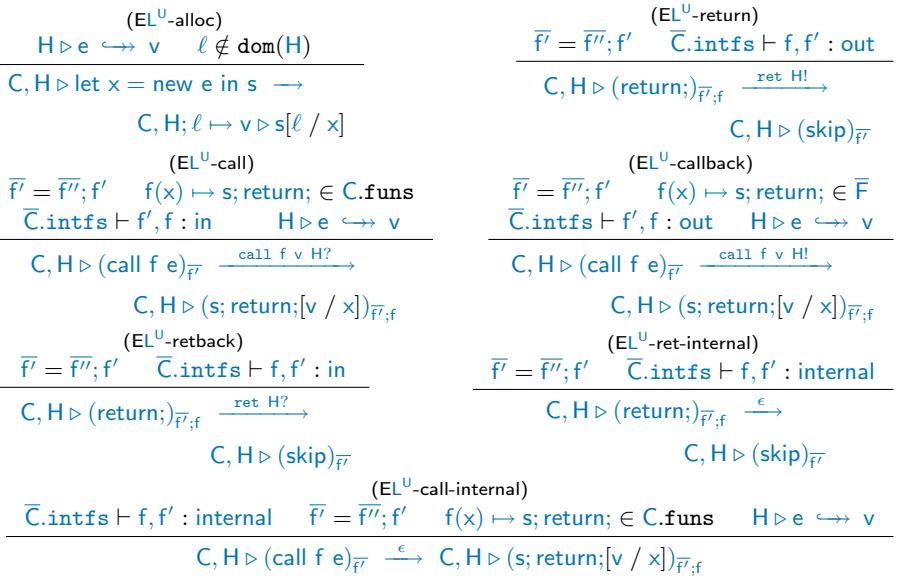
As explained in Sect. 2.1, safety properties are specified by monitors. L^{U}'s monitors have the form: $\mathsf{M} ::= (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}})$. Note that in place of the set $\{l \cdots\}$ of safety-relevant locations, the description of a monitor here (as well as a component above) contains a *single* location ℓ_{root}. The interpretation is that any location *reachable* in the heap starting from ℓ_{root} is relevant for safety. This set of locations can change as the program executes, and hence this is more flexible than statically specifying all of $\{l \cdots\}$ upfront. This representation of the set by a single location is made explicit in the following monitor rule:

(L^{U}-Monitor Step)

$$\frac{\mathsf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}}) \quad \mathsf{M}' = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{f}}) \quad (\sigma_{\mathsf{c}}, \mathsf{H}', \sigma_{\mathsf{f}}) \in \rightsquigarrow \quad \mathsf{H}' \subseteq \mathsf{H} \quad \mathtt{dom}(\mathsf{H}') = \mathtt{reach}(\ell_{\mathsf{root}}, \mathsf{H})}{\mathsf{M}; \mathsf{H} \rightsquigarrow \mathsf{M}'}$$

Other than this small point, monitors, safety, robust safety and *RSC* are defined as in Sect. 2. In particular, a monitor and a component agree if they mention the same ℓ_{root}: $\mathsf{M} \frown \mathsf{C} \stackrel{\mathsf{def}}{=} (\mathsf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}}))$ and $(\mathsf{C} = (\ell_{\mathsf{root}}; \overline{\mathsf{F}}; \overline{\mathsf{I}}))$

A program state $\mathsf{C}, \mathsf{H} \triangleright (\mathsf{s})_{\overline{\mathsf{f}}}$ (denoted with Ω) includes the function bodies C, the heap H, a statement s being executed and a stack of function calls $\overline{\mathsf{f}}$ (often omitted in the rules for simplicity). The latter is used to populate judgements of the form $\overline{\mathsf{I}} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{internal/in/out}$. These determine whether calls and returns are $\mathsf{internal}$ (within the attacker or within the component), directed from the attacker to the component (in) or directed from the component to the attacker (out). This information is used to determine whether the semantics should generate a label, as in Rules EL^{U}-return to EL^{U}-retback, or no label, as in Rules EL^{U}-ret-internal and EL^{U}-call-internal since internal calls should not be observable. L^{U} has a big-step semantics for expressions ($\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\rightarrow \mathsf{v}$) that relies on evaluation contexts, a small-step semantics for statements ($\Omega \xrightarrow{\lambda} \Omega'$) that has labels $\lambda ::= \epsilon \mid \alpha$ and a semantics that accumulates labels in traces ($\Omega \stackrel{\overline{\alpha}}{\Longrightarrow} \Omega'$) by omitting silent actions ϵ and concatenating the rest. Unlike existing work on compositional compiler correctness which only rely on having the component [40], the semantics relies on having both the component and the context.



3.2 The Target Language $\mathbf{L^P}$

$\mathbf{L^P}$ is an untyped, imperative language that follows the structure of $\mathsf{L^U}$ and it has similar expressions and statements. However, there are critical differences (that make the compiler interesting). The main difference is that heap locations in $\mathbf{L^P}$ are concrete natural numbers. Upfront, an adversarial context can guess locations used as private state by a component and clobber them. To support hidden local state, a location can be "hidden" explicitly via the statement **let x = hide e in s**, which allocates a new capability **k**, an abstract token that grants access to the location **n** to which **e** points [64]. Subsequently, all reads and writes to **n** must be authenticated with the capability, so reading and writing a location take another parameter as follows: **!e with e** and **x := e with e**. In both cases, the **e** after the **with** is the capability. Unlike locations, capabilities cannot be guessed. To make a location private, the compiler can make the capability of the location private. To bootstrap this hiding process, we assume that a component has one location that can only be accessed by it, a priori in the semantics (in our formalization, we always focus on only one component and we assume that, for this component, this special location is at address **0**).

In detail, $\mathbf{L^P}$ heaps **H** are maps from natural numbers (locations) **n** to values **v** and a tag η as well as capabilities, so **H** ::= $\varnothing \mid \mathbf{H}; \mathbf{n} \mapsto \mathbf{v} : \eta \mid \mathbf{H}; \mathbf{k}$. The tag η can be $\bot$, which means that **n** is globally available (not protected) or a capability **k**, which protects **n**. A globally available location can be freely read and written but one that is protected by a capability requires the capability to be supplied at the time of read/write (Rule E$\mathbf{L^P}$-assign, Rule E$\mathbf{L^P}$-deref).

$\mathbf{L^P}$ also has a big-step semantics for expressions, a labelled small-step semantics and a semantics that accumulates traces analogous to that of $\mathsf{L^U}$.

$$\frac{\mathbf{n} \mapsto \mathbf{v} : \eta \in \mathbf{H} \qquad (\eta = \bot) \text{ or } (\eta = \mathbf{k} \text{ and } \mathbf{v}' = \mathbf{k})}{\mathbf{H} \triangleright !\mathbf{n} \text{ with } \mathbf{v}' \hookrightarrow\!\!\!\!\rightarrow \mathbf{H} \triangleright \mathbf{v}} \; (\mathsf{EL}^{\mathsf{P}}\text{-deref})$$

$$\frac{\mathbf{H} = \mathbf{H}_1; \mathbf{n} \mapsto (\mathbf{v}, \eta) \qquad \mathbf{H} \triangleright \mathbf{e} \hookrightarrow\!\!\!\!\rightarrow \mathbf{v} \qquad \mathbf{H}' = \mathbf{H}; \mathbf{n}+1 \mapsto \mathbf{v} : \bot}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{let\ x = new\ e\ in\ s} \longrightarrow \mathbf{C}, \mathbf{H}' \triangleright \mathbf{s}[\mathbf{n}+1 \,/\, \mathbf{x}]} \; (\mathsf{EL}^{\mathsf{P}}\text{-new})$$

$$\frac{\mathbf{H} \triangleright \mathbf{e} \hookrightarrow\!\!\!\!\rightarrow \mathbf{n} \qquad \mathbf{k} \notin \mathtt{dom}(\mathbf{H}) \qquad \mathbf{H} = \mathbf{H}_1; \mathbf{n} \mapsto \mathbf{v} : \bot; \mathbf{H}_2 \qquad \mathbf{H}' = \mathbf{H}_1; \mathbf{n} \mapsto \mathbf{v} : \mathbf{k}; \mathbf{H}_2; \mathbf{k}}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{let\ x = hide\ e\ in\ s} \longrightarrow \mathbf{C}, \mathbf{H}' \triangleright \mathbf{s}[\mathbf{k} \,/\, \mathbf{x}]} \; (\mathsf{EL}^{\mathsf{P}}\text{-hide})$$

$$\frac{\mathbf{H} \triangleright \mathbf{e} \hookrightarrow\!\!\!\!\rightarrow \mathbf{v} \qquad \mathbf{H} = \mathbf{H}_1; \mathbf{n} \mapsto \_ : \eta; \mathbf{H}_2 \qquad \mathbf{H}' = \mathbf{H}_1; \mathbf{n} \mapsto \mathbf{v} : \eta; \mathbf{H}_2 \qquad (\eta = \bot) \text{ or } (\eta = \mathbf{k} \text{ and } \mathbf{v}' = \mathbf{k})}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{n} := \mathbf{e} \text{ with } \mathbf{v}' \longrightarrow \mathbf{C}, \mathbf{H}' \triangleright \mathbf{skip}} \; (\mathsf{EL}^{\mathsf{P}}\text{-assign})$$

A second difference between $\mathbf{L}^{\mathbf{P}}$ and L^{U} is that $\mathbf{L}^{\mathbf{P}}$ has no booleans, while L^{U} has them. This makes the compiler and the related proofs interesting, as discussed in the proof of Theorem 1.

In $\mathbf{L}^{\mathbf{P}}$, the locations of interest to a monitor are all those that can be reached from the address $\mathbf{0}$. $\mathbf{0}$ itself is protected with a capability $\mathbf{k_{root}}$ that is assumed to occur only in the code of the component in focus, so a component is defined as $\mathbf{C} ::= \mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}}$. We can now give a precise definition of component-monitor agreement for $\mathbf{L}^{\mathbf{P}}$ as well as a precise definition of attacker, which must care about the $\mathbf{k_{root}}$ capability.

$$\mathbf{M} \frown \mathbf{C} \stackrel{\text{def}}{=} (\mathbf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \mathbf{k_{root}}, \sigma_{\mathbf{c}})) \text{ and } (\mathbf{C} = (\mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}}))$$

$$\mathbf{C} \vdash \mathbf{A} : \mathbf{atk} \stackrel{\text{def}}{=} \mathbf{C} = (\mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}}), \mathbf{A} = \overline{\mathbf{F}'}, \mathbf{k_{root}} \notin \mathtt{fn}(\overline{\mathbf{F}'})$$

3.3 Compiler from L^{U} to $\mathbf{L}^{\mathbf{P}}$

We now present $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$, the compiler from L^{U} to $\mathbf{L}^{\mathbf{P}}$, detailing how it uses the capabilities of $\mathbf{L}^{\mathbf{P}}$ to achieve *RSC*. Then, we prove that $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ attains *RSC*.

Compiler $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ takes as input a L^{U} component C and returns a $\mathbf{L}^{\mathbf{P}}$ component (excerpts of the translation are shown below). The compiler performs a simple pass on the structure of functions, expressions and statements. Each L^{U} location is encoded as a pair of a $\mathbf{L}^{\mathbf{P}}$ location and the capability to access the location; location update and dereference are compiled accordingly. The compiler codes source booleans true to $\mathbf{0}$ and false to $\mathbf{1}$, and the source number n to the target counterpart $\mathbf{n}$.

$$\left[\!\left[\ell_{\mathsf{root}}; \overline{\mathsf{F}}; \overline{\mathsf{I}}\right]\!\right]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = \mathbf{k_{root}}; \left[\!\left[\overline{\mathsf{F}}\right]\!\right]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}; \left[\!\left[\overline{\mathsf{I}}\right]\!\right]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$$

$$[\![!\mathsf{e}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = !\,[\![\mathsf{e}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}.\mathbf{1} \text{ with } [\![\mathsf{e}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}.\mathbf{2}$$

$$\left[\!\left[\begin{array}{l}\mathsf{let\ x = new\ e}\\ \mathsf{in\ s}\end{array}\right]\!\right]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = \begin{array}{l}\mathbf{let\ x_{loc} = new}\ [\![\mathsf{e}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}\ \mathbf{in\ let\ x_{cap} = hide\ x_{loc}\ in}\\ \quad \mathbf{let\ x} = \langle \mathbf{x_{loc}}, \mathbf{x_{cap}} \rangle\ \mathbf{in}\ [\![\mathsf{s}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}\end{array}$$

$$[\![\mathsf{x} := \mathsf{e}']\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = \mathbf{let\ x_{loc} = x.1\ in\ let\ x_{cap} = x.2\ in\ x_{loc} :=}\ [\![\mathsf{e}']\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}\ \mathbf{with\ x_{cap}}$$

This compiler solely relies on the capability abstraction of the target language as a defence mechanism to attain *RSC*. Unlike existing secure compilers, $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ needs neither dynamic checks nor other constructs that introduce runtime overhead to attain *RSC* [9,32,39,53,59].

Proof of *RSC*. Compiler $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ attains *RSC* (Theorem 1). In order to set up this theorem, we need to instantiate the cross-language relation for values, which we write as $\approx_\beta$ here. The relation is parametrised by a partial bijection $\beta : \ell \times \mathbf{n} \times \eta$ from source heap locations to target heap locations which determines when a source location and a target location (and its capability) are related. On values, $\approx_\beta$ is defined as follows: $\mathsf{true} \approx_\beta \mathbf{0}$; $\mathsf{false} \approx_\beta \mathbf{n}$ when $\mathbf{n} \neq \mathbf{0}$; $\mathsf{n} \approx_\beta \mathbf{n}$; $\ell \approx_\beta \langle \mathbf{n}, \mathbf{k} \rangle$ if $(\ell, \mathbf{n}, \mathbf{k}) \in \beta$; $\ell \approx_\beta \langle \mathbf{n}, \_ \rangle$ if $(\ell, \mathbf{n}, \bot) \in \beta$; $\langle \mathsf{v}_1, \mathsf{v}_2 \rangle \approx_\beta \langle \mathbf{v_1}, \mathbf{v_2} \rangle$ if $\mathsf{v}_1 \approx_\beta \mathbf{v_1}$ and $\mathsf{v}_2 \approx_\beta \mathbf{v_2}$. This relation is then used to define the heap, monitor state and action relations. Heaps are related, written $\mathsf{H} \approx_\beta \mathbf{H}$, when locations related in β point to related values. States are related, written $\Omega \approx_\beta \mathbf{\Omega}$, when they have related heaps. The action relation ($\alpha \approx_\beta \alpha$) is defined as in Sect. 2.2.

Monitor Relation. In Sect. 2.2, we left the monitor relation abstract. Here, we define it for our two languages. Two monitors are related when they can *simulate* each other on related heaps. Given a monitor-specific relation $\sigma \approx \sigma$ on monitor states, we say that a relation $\mathcal{R}$ on source and target monitors is a *bisimulation* if the following hold whenever $\mathsf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}})$ and $\mathbf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{k_{root}}, \sigma_{\mathbf{c}})$ are related by $\mathcal{R}$:

1. $\sigma_0 \approx \sigma_{\mathbf{0}}$, and $\sigma_{\mathsf{c}} \approx \sigma_{\mathbf{c}}$, and
2. For all β containing $(\ell_{\mathsf{root}}, \mathbf{0}, \mathbf{k_{root}})$ and all $\mathsf{H}, \mathbf{H}$ with $\mathsf{H} \approx_\beta \mathbf{H}$:
   (a) $(\sigma_{\mathsf{c}}, \mathsf{H}, \_) \in \rightsquigarrow$ iff $(\sigma_{\mathbf{c}}, \mathbf{H}, \_) \in \rightsquigarrow$, and
   (b) $(\sigma_{\mathsf{c}}, \mathsf{H}, \sigma') \in \rightsquigarrow$ and $(\sigma_{\mathbf{c}}, \mathbf{H}, \sigma') \in \rightsquigarrow$ imply
   $(\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma') \mathcal{R} (\{\sigma \cdots\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{k_{root}}, \sigma')$.

In words, $\mathcal{R}$ is a bisimulation only if $\mathsf{M} \mathcal{R} \mathbf{M}$ implies that M and $\mathbf{M}$ simulate each other on heaps related by *any* β that relates ℓ_{root} to $\mathbf{0}$. In particular, this means that neither M nor $\mathbf{M}$ can be sensitive to the *specific* addresses allocated during the run of the program. However, they can be sensitive to the "shape" of the heap or the values stored in the heap. Note that the union of any two bisimulations is a bisimulation. Hence, there is a largest bisimulation, which we denote as $\approx$. Intuitively, $\mathsf{M} \approx \mathbf{M}$ implies that M and $\mathbf{M}$ encode the same safety property (up to the aforementioned relation on values $\approx_\beta$). With all the boilerplate for *RSC* in place, we state our main theorem.

Theorem 1 ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ attains *RSC*). $\vdash [\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} : RSC$

We outline our proof of Theorem 1, which relies on a backtranslation $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$. Intuitively, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$ takes a target trace $\overline{\alpha}$ and builds a *set* of source contexts such that *one* of them when linked with C, produces a related trace $\overline{\alpha}$ in the source (Theorem 2). In prior work, backtranslations return a single context [10,11,21,

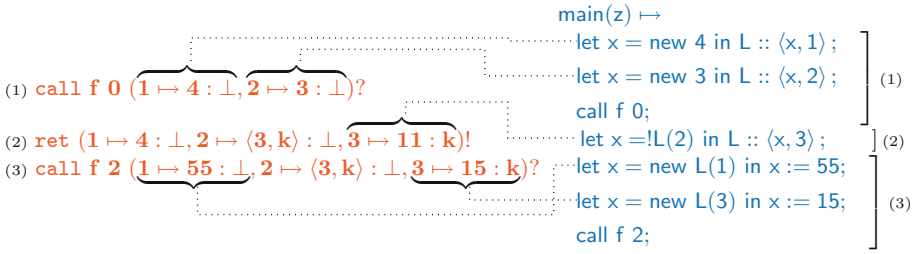


Fig. 1. Example of a trace and its backtranslated code.

28,50,53,59]. This is because they all, explicitly or implicitly, assume that $\approx$ is injective from source to target. Under this assumption, the backtranslation is unique: a target value $\mathbf{v}$ will be related to at most one source value v. We do away with this assumption (e.g., the target value $\mathbf{0}$ is related to both source values $\mathsf{0}$ and true) and thus there can be multiple source values related to any given target value. This results in a set of backtranslated contexts, of which at least one will reproduce the trace as we need it.

We bypass the lengthy technical setup for this proof and provide an informal description of why the backtranslation achieves what it is supposed to. As an example, Fig. 1 contains a trace $\overline{\alpha}$ and the the output of $\langle\!\langle \overline{\alpha} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$.

$\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$ first generates empty method bodies for all context methods called by the compiled component. Then it backtranslates each *action* on the given trace, generating code blocks that mimic that action and places that code inside the appropriate method body. Figure 1 shows the code blocks generated for each action. Backtranslated code maintains a support data structure at runtime, a list of locations denoted L where locations are added ($::$) and they are looked up ($\mathsf{L}(\mathsf{n})$) based on their second field n, which is their target-level address. In order to backtranslate the first call, we need to set up the heap with the right values and then perform the call. In the diagram, dotted lines describe which source statement generates which part of the heap. The return only generates code that will update the list L to ensure that the context has access to all the locations it knows in the target too. In order to backtranslate the last call we lookup the locations to be updated in L so we can ensure that when the $\mathsf{call\ f\ 2}$ statement is executed, the heap is in the right state.

For the backtranslation to be used in the proof we need to prove its correctness, i.e., that $\langle\!\langle \overline{\alpha} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$ generates a context A that, together with C, generates a trace $\overline{\alpha}$ related to the given target trace $\overline{\alpha}$.

Theorem 2 ($\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$ **is correct)**

if $\mathbf{A}\left[\llbracket \mathsf{C} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\right] \overset{\overline{\alpha}}{\Longrightarrow} \mathbf{\Omega}$ *then* $\exists \mathsf{A} \in \langle\!\langle \overline{\alpha} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} . \mathsf{A}[\mathsf{C}] \overset{\overline{\alpha}}{\Longrightarrow} \Omega$ *and* $\overline{\alpha} \approx_\beta \overline{\alpha}$ *and* $\Omega \approx_\beta \mathbf{\Omega}$.

This theorem immediately implies that $\vdash [\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} : PF\text{-}RSC$, which, by Theorem 3 below, implies that $\vdash [\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} : RSC$.

Theorem 3 (*PF-RSC* **and** *RSC* **are equivalent for** $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$**).**

$$\vdash [\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} : PF\text{-}RSC \iff \vdash [\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} : RSC$$

Example 4 (Compiling a secure program). To illustrate *RSC* at work, let us consider the following source component $\mathsf{C_a}$, which manages an account whose balance is security-relevant. Accordingly, the balance is stored in a location (ℓ_{root} that is tracked by the monitor. $\mathsf{C_a}$ provides functions to deposit to the account as well as to print the account balance.

$$\begin{aligned}\mathsf{deposit(x)} &\mapsto \mathsf{let\ q{=}abs(x)\ in\ let\ amt} = !\ell_{\mathsf{root}}\ \mathsf{in}\ \ell_{\mathsf{root}} := \mathsf{amt} + \mathsf{q}\\ \mathsf{balance()} &\mapsto !\ell_{\mathsf{root}}\end{aligned}$$

$\mathsf{C_a}$ never leaks any sensitive location (ℓ_{root}) to an attacker. Additionally, an attacker has no way to decrement the amount of the balance since deposit only adds the absolute value $\mathsf{abs(x)}$ of its input x to the existing balance.

By compiling $\mathsf{C_a}$ with $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$, we obtain the following target program.

$$\begin{aligned}\mathbf{deposit(x)} &\mapsto \mathbf{let\ q{=}abs(x)\ in}\\ &\qquad\qquad \mathbf{let\ amt{=}!0\ with\ k_{root}\ in\ 0 := amt + q\ with\ k_{root}}\\ \mathbf{balance()} &\mapsto \mathbf{!0\ with\ k_{root}}\end{aligned}$$

Recall that location ℓ_{root} is mapped to location **0** and protected by the $\mathbf{k_{root}}$ capability. In the compiled code, while location **0** is freely computable by a target attacker, capability $\mathbf{k_{root}}$ is not. Since that capability is not leaked to an attacker, an attacker will not be able to tamper with the balance stored in location **0**. ⊡

4 *RSC* via Bisimulation

If the source language has a verification system that enforces robust safety, proving that a compiler attains *RSC* can be simpler than that of Sect. 3—it may not require a back translation. To demonstrate this, we consider a specific class of monitors, namely those that enforce type invariants on a specific set of locations. Our source language, L^{τ}, is similar to L^{U} but it has a type system that accepts only those source programs whose traces the source monitor never rejects. Our compiler $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ is directed by typing derivations, and its proof of *RSC* establishes a specific cross-language invariant on program execution, rather than a backtranslation. A second, independent goal of this section is to show that *RSC* is compatible with concurrency. Consequently, our source and target languages include constructs for forking threads.

4.1 The Source Language L^{τ}

L^{τ} extends L^{U} with concurrency, so it has a fork statement $(\|\ \mathsf{s})$, processes and process soups [19]. Components define a set of safety-relevant locations Δ, so $\mathsf{C} ::= \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}}$ and heaps carry type information, so $\mathsf{H} ::= \varnothing \mid \mathsf{H}; \ell \mapsto \mathsf{v} : \tau$. Δ also specifies a type for each safety-relevant location, so $\Delta ::= \varnothing \mid \Delta; (\ell : \tau)$.

L^{τ} has an unconventional type system that enforces *robust type safety* [1,14,31,34,45,58], which means that no context can cause the static types of sensitive heap locations to be violated at runtime. Using a special type UN that is described below, a program component statically partitions heap locations it deals with into those it cares about (sensitive or "trusted" locations) and those it does not care about ("untrusted" locations). Call a value *shareable* if only untrusted locations can be extracted from it using the language's elimination constructs. The type system then ensures that a program component only ever shares shareable values with the context. This ensures that the context cannot violate any invariants (including static types) of the trusted locations, since it can never gets direct access to them.

Technically, the type system considers the types $\tau ::= \mathsf{Bool} \mid \mathsf{Nat} \mid \tau \times \tau \mid \mathsf{Ref}\ \tau \mid \mathsf{UN}$ and the following typing judgements (Γ maps variables to types).

$\vdash \mathsf{C} : \mathsf{UN}$	Component C is well-typed.	$\Delta, \Gamma \vdash \mathsf{e} : \tau$	Expression e has type τ.
$\tau \vdash \circ$	Type τ is shareable.	$\mathsf{C}, \Delta, \Gamma \vdash \mathsf{s}$	Statement s is well-typed.

$$\frac{}{\mathsf{Bool} \vdash \circ}\ (\mathsf{TL}^{\tau}\text{-bool-pub}) \qquad \frac{}{\mathsf{Nat} \vdash \circ}\ (\mathsf{TL}^{\tau}\text{-nat-pub}) \qquad \frac{\tau \vdash \circ \quad \tau' \vdash \circ}{\tau \times \tau' \vdash \circ}\ (\mathsf{TL}^{\tau}\text{-pair-pub})$$

$$\frac{}{\mathsf{UN} \vdash \circ}\ (\mathsf{TL}^{\tau}\text{-un-pub}) \qquad \frac{}{\mathsf{Ref}\ \mathsf{UN} \vdash \circ}\ (\mathsf{TL}^{\tau}\text{-references-pub})$$

Type UN stands for "untrusted" or "shareable" and contains all values that can be passed to the context. Every type that is not a subtype of UN is implicitly trusted and cannot be passed to the context. Untrusted locations are explicitly marked UN at their allocation points in the program. Other types are deemed shareable via subtyping. Intuitively, a type is safe if values in it can only yield locations of type UN by the language elimination constructs. For example, $\mathsf{UN} \times \mathsf{UN}$ is a subtype of UN. We write $\tau \vdash \circ$ to mean that τ is a subtype of UN.

Further, L^{τ} contains an *endorsement* statement ($\mathsf{endorse}\ \mathsf{x} = \mathsf{e}\ \mathsf{as}\ \varphi\ \mathsf{in}\ \mathsf{s}$) that dynamically checks the top-level constructor of a value of type UN and gives it a more precise superficial type $\varphi ::= \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{UN} \times \mathsf{UN} \mid \mathsf{Ref}\ \mathsf{UN}$ [24]. This allows a program to safely inspect values coming from the context. It is similar to existing type casts [48] but it only inspects one structural layer of the value (this simplifies the compilation).

The operational semantics of L^{τ} updates that of L^{U} to deal with concurrency and endorsement. The latter performs a runtime check on the endorsed value [62].

Monitors $\mathsf{M} ::= (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_{\mathsf{c}})$ check at runtime that the set of trusted heap locations Δ have values of their intended static types. Accordingly, the description of the monitor includes a list of trusted locations and their expected types (in the form of an environment Δ). The type τ of any location in Δ must be trusted, so $\tau \not\vdash \circ$. To facilitate checks of the monitor, every heap

location carries a type at runtime (in addition to a value). The monitor transitions should therefore be of the form (σ, Δ, σ), but since Δ never changes, we write the transitions as (σ, σ).

A monitor and a component agree if they have the same Δ: $\mathsf{M} \frown \mathsf{C} \stackrel{\text{def}}{=} (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_c) \frown (\Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}})$. Other definitions (safety, robust safety and actions) are as in Sect. 2. Importantly, a well-typed component generates traces that are always accepted, so every component typed at UN is robustly safe.

Theorem 4 (Typability Implies Robust Safety in L^{τ})

$$\textit{If } \vdash \mathsf{C} : \mathsf{UN} \textit{ and } \mathsf{C} \frown \mathsf{M} \textit{ then } \mathsf{M} \vdash \mathsf{C} : \mathsf{rs}$$

Richer Source Monitors. In L^{τ}, source language monitors only enforce the property of type safety on specific memory locations (robustly). This can be generalized substantially to enforce arbitrary invariants other than types on locations. The only requirement is to find a type system (e.g., based on refinements or Hoare logics) that can enforce robust safety in the source (cf. [68]). Our compilation and proof strategy should work with little modification. Another easy generalization is allowing the set of locations considered by the monitor to grow over time, as in Sect. 3.

4.2 The Target Language $\mathbf{L}^{\pi}$

Our target language, $\mathbf{L}^{\pi}$, extends the previous target language $\mathbf{L}^{\mathbf{P}}$, with support for concurrency (forking, processes and process soups), atomic co-creation of a protected location and its protecting capability (**let x = newhide e in s**) and for examining the top-level construct of a value (**destruct x = e as B in s or s′**) according to a pattern (**B ::= nat | pair**).

$$\frac{\mathbf{H} \triangleright \mathbf{e} \hookrightarrow\!\!\!\!\rightarrow \mathbf{n}}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{destruct\ x = e\ as\ nat\ in\ s\ or\ s'} \longrightarrow \mathbf{C}, \mathbf{H} \triangleright \mathbf{s}[\mathbf{n}\ /\ \mathbf{x}]} \text{(EL}^{\pi}\text{-destruct-nat)}$$

$$\frac{\mathbf{H} = \mathbf{H_1}; \mathbf{n} \mapsto (\mathbf{v}, \eta) \quad \mathbf{H} \triangleright \mathbf{e} \hookrightarrow\!\!\!\!\rightarrow \mathbf{v} \quad \mathbf{k} \notin \mathtt{dom}(\mathbf{H}) \quad \mathbf{s'} = \mathbf{s}[\langle \mathbf{n}+\mathbf{1}, \mathbf{k} \rangle\ /\ \mathbf{x}]}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{let\ x = newhide\ e\ in\ s} \longrightarrow \mathbf{C}, \mathbf{H}; \mathbf{n}+\mathbf{1} \mapsto \mathbf{v} : \mathbf{k}; \mathbf{k} \triangleright \mathbf{s'}} \text{(EL}^{\pi}\text{-new)}$$

Monitors are also updated to consider a fixed set of locations (a heap $\mathbf{H_0}$), so $\mathbf{M} ::= (\{\sigma \cdots\}, \rightsquigarrow, \sigma_0, \mathbf{H_0}, \sigma_c)$. The atomic creation of capabilities is provided to match modern security architectures such as Cheri [71] (which implement capabilities at the hardware level). This atomicity is not strictly necessary and we prove that *RSC* is attained both by a compiler relying on it and by one that allocates a location and then protects it non-atomically. The former compiler (with this atomicity in the target) is a bit easier to describe, so for space reasons, we only describe that here and defer the other one to the companion report [61].

4.3 Compiler from L^{τ} to L^{π}

The high-level structure of the compiler, $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathsf{L}^{\pi}}$, is similar to that of our earlier compiler $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathsf{L}^{\mathsf{P}}}$ (Sect. 3.3). However, $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathsf{L}^{\pi}}$ is defined by induction on the type derivation of the component to be compiled. The case for allocation (presented below) explicitly uses type information to achieve security efficiently, protecting only those locations whose type is not UN.

$$\left[\!\!\left[\dfrac{\Delta, \Gamma \vdash \mathsf{e} : \tau \quad \mathsf{C}, \Delta, \Gamma; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}}{\mathsf{C}, \Delta, \Gamma \vdash \mathsf{let\ x = new}_{\tau}\ \mathsf{e\ in\ s}} \right]\!\!\right]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \begin{cases} \mathbf{let\ xo\ = new}\ [\![\Delta, \Gamma \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \\ \quad \mathbf{in\ let\ x} = \langle \mathbf{xo}, \mathbf{0} \rangle \\ \quad\ \mathbf{in}\ [\![\mathsf{C}, \Delta, \Gamma; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} & \text{if } \tau = \mathsf{UN} \\ \\ \mathbf{let\ x\ = newhide}\ [\![\Delta, \Gamma \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \\ \quad \mathbf{in}\ [\![\mathsf{C}, \Delta, \Gamma; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} & \text{otherwise} \end{cases}$$

New Monitor Relation. As monitors have changed, we also need a new monitor relation $\mathsf{M} \approx \mathbf{M}$. Informally, a source and a target monitor are related if the target monitor can always step whenever the target heap satisfies the types specified in the source monitor (up to renaming by the partial bijection β).

We write $\vdash \mathsf{H} : \Delta$ to mean that for each location $\ell \in \Delta$, $\vdash \mathsf{H}(\ell) : \Delta(\ell)$. Given a partial bijection β from source to target locations, we say that a target monitor $\mathbf{M} = (\{\sigma \cdots\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{H_0}, \sigma_{\mathbf{c}})$ is good, written $\vdash \mathbf{M} : \beta, \Delta$, if for all $\sigma \in \{\sigma \cdots\}$ and all $\mathsf{H} \approx_{\beta} \mathbf{H}$ such that $\vdash \mathsf{H} : \Delta$, there is a σ' such that $(\sigma, \mathbf{H}, \sigma') \in \rightsquigarrow$. For a fixed partial bijection β_0 between the domains of Δ and $\mathbf{H_0}$, we say that the source monitor M and the target monitor $\mathbf{M}$ are related, written $\mathsf{M} \approx \mathbf{M}$, if $\vdash \mathbf{M} : \beta_0, \Delta$ for the Δ in M. With this setup, we define *RSC* as in Sect. 2.

Theorem 5 (Compiler $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ attains *RSC*). $\vdash [\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} : \mathit{RSC}$

To prove that $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ attains *RSC* we do not rely on a backtranslation. Here, we know statically which locations can be monitor-sensitive: they must all be trusted, i.e., must have a type τ satisfying $\tau \not\sim \circ$. Using this, we set up a simple cross-language relation and show it to be an invariant on runs of source and compiled target components. The relation captures the following:

- Heaps (both source and **target**) can be partitioned into two parts, a *trusted* part and an *untrusted* part;
- The trusted source heap contains only locations whose type is trusted ($\tau \not\sim \circ$);
- The trusted **target heap** contains only locations related to trusted source locations and these point to related values; more importantly, every **trusted target location** is protected by a capability;
- In the **target**, any capability protecting a trusted location does not occur in attacker code, nor is it stored in an untrusted heap location.

We need to prove that this relation is preserved by reductions both in compiled and in attacker code. The former follows from source robust safety (Theorem 4). The latter is simple since all trusted locations are protected with capabilities, attackers have no access to trusted locations, and capabilities are unforgeable and unguessable (by the semantics of $\mathbf{L}^{\pi}$). At this point, knowing that monitors are related, and that source traces are always accepted by source monitors, we can conclude that target traces are always accepted by target monitors too. Note that this kind of an argument requires all compilable source programs to be robustly safe and is, therefore, impossible for our first compiler $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$. Avoiding the backtranslation results in a proof much simpler than that of Sect. 3.

5 Fully Abstract Compilation

Our next goal is to compare *RSC* to *FAC* at an intuitive level. We first define fully abstract compilation or *FAC* (Sect. 5.1). Then, we present an example of how *FAC* may result in inefficient compiled code and use that to present in Sect. 5.2 what would be needed to write a fully abstract compiler from L^{U} to $\mathbf{L}^{\mathbf{P}}$ (the languages of our first compiler). We use this example to compare *RSC* and *FAC* concretely, showing that, at least on this example, *RSC* permits more efficient code and affords simpler proofs that *FAC*.

However, this does not imply that one should always prefer *RSC* to *FAC* blindly. In some cases, one may want to establish full abstraction for reasons other than security. Also, when the target language is typed [10,11,21,50] or has abstractions similar to those of the source, full abstraction may have no downsides (in terms of efficiency of compiled code and simplicity of proofs) relative to *RSC*. However, in many settings, including those we consider, target languages are not typed, and often differ significantly from the source in their abstractions. In such cases, *RSC* is a worthy alternative.

5.1 Formalising Fully Abstract Compilation

As stated in Sect. 1, *FAC* requires the preservation and reflection of observational equivalence, and most existing work instantiates observational equivalence with contextual equivalence ($\simeq_{ctx}$). Contextual equivalence and *FAC* are defined below. Informally, two components C_1 and C_2 are contextually equivalent if no context A interacting with them can tell them apart, i.e., they are *indistinguishable*. Contextual equivalence can encode security properties such as confidentiality, integrity, invariant maintenance and non-interference [6,9,53,60]. We do not explain this well-known observation here, but refer the interested reader to the survey of Patrignani *et al.* [54]. Informally, a compiler $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ is fully abstract if it translates (only) contextually-equivalent source components into contextually-equivalent target ones.

Definition 4 (Contextual equivalence and fully abstract compilation).

$C_1 \simeq_{ctx} C_2 \stackrel{\text{def}}{=} \forall A.A\,[C_1]\Uparrow \iff A\,[C_2]\Uparrow$, *where* $\Uparrow$ *means execution divergence*

$\vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : FAC \stackrel{\text{def}}{=} \forall \mathsf{C}_1, \mathsf{C}_2.\, \mathsf{C}_1 \simeq_{ctx} \mathsf{C}_2 \iff [\![\mathsf{C}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{C}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$

The security-relevant part of *FAC* is the $\Rightarrow$ implication [29]. This part is security-relevant because the proof thesis concerns target contextual equivalence ($\simeq_{ctx}$). Unfolding the definition of $\simeq_{ctx}$ on the right of the implication yields a universal quantification over all possible target contexts $\mathbf{A}$, which captures malicious attackers. In fact, there may be target contexts $\mathbf{A}$ that can interact with compiled code in ways that are impossible in the source language. Compilers that attain *FAC* with untyped target languages often insert checks in compiled code that detect such interactions and respond to them securely [60], often by halting the execution [6,9,29,37,39,42,53,54]. These checks are often inefficient, but must be performed even if the interactions are not security-relevant. We now present an example of this.

Example 5 (Wrappers for heap resources). Consider a password manager written in an object-oriented language that is compiled to an assembly-like language. The password manager defines a `private List` object where it stores the passwords locally. Shown below are two implementations of the `newList` method inside `List` which we call $\mathsf{C}_{\mathsf{one}}$ and $\mathsf{C}_{\mathsf{two}}$. The only difference between $\mathsf{C}_{\mathsf{one}}$ and $\mathsf{C}_{\mathsf{two}}$ is that $\mathsf{C}_{\mathsf{two}}$ allocates two lists internally; one of these (`shadow`) is used for internal purposes only.

```
1 public newList(): List{
2
3   ell = new List();
4   return ell;
5 }
```

```
1 public newList(): List{
2   shadow = new List(); // diff
3   ell = new List();
4   return ell;
5 }
```

$\mathsf{C}_{\mathsf{one}}$ and $\mathsf{C}_{\mathsf{two}}$ are equivalent in a source language that does not allow pointer comparison (like our source languages). To attain *FAC* when the target allows pointer comparisons (as in our target languages), the pointers returned by `newList` in the two implementations must be the same, but this is very difficult to ensure since the second implementation does more allocations. A simple solution to this problem is to wrap `ell` in a proxy object and return the proxy [9,47,53,59]. Compiled code needs to maintain a lookup table mapping the proxy to the original object and proxies must have allocation-independent addresses. Proxies work but they are inefficient due to the need to look up the table on every object access. ⊡

In this example, *FAC* forces all privately allocated locations to be wrapped in proxies. However, *RSC* does not require this. Our target languages $\mathbf{L}^{\mathbf{P}}$ and $\mathbf{L}^{\pi}$ support address comparison (addresses are natural numbers in their heaps) but $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ and $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ just use capabilities to attain security efficiently while $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\mathbf{I}}}$ relies on memory isolation. On the other hand, for attaining *FAC*, capabilities alone would be insufficient since they do not hide addresses. We explain this in detail in the next subsection.

Remarks. Our technical report lists many other cases of *FAC* forcing security-irrelevant inefficiency in compiled code [61]. All of these can be avoided by just replacing contextual equivalence with a different notion of equivalence in the statement of *FAC*. However, it is not clear how this can be done generally for any given kind of inefficiency, and what the security consequences of such instantiations of the statement of *FAC* are. On the other hand, *RSC* is *uniform* and it does not induce any of these inefficiencies.

A security issue that cannot be addressed just by tweaking equivalences is information leaks on side channels, as side channels are, by definition, not expressible in the language. Neither *FAC* nor *RSC* deals with side channels.

5.2 Towards a Fully Abstract Compiler from L^{U} to L^{P}

To further compare *FAC* and *RSC*, we now sketch what *would* be needed to construct a fully abstract compiler from L^{U} to L^{P}. In particular, this compiler should not suffer from the "attack" described in Example 5.

Inefficiency. We denote with $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathsf{L}^{\mathsf{P}}}$ a (hypothetical) new compiler from L^{U} to L^{P} that attains *FAC*. We describe informally what code generated by this compiler would have to do. We know that fully abstract compilation preserves *all* source abstractions in the target language. One abstraction that distinguishes L^{P} from L^{U} is that locations are abstract in L^{P}, but concrete natural numbers in L^{U}. Thus, locations allocated by compiled code must not be passed directly to the context as this would reveal the allocation order. Instead of passing the location $\langle \mathbf{n}, \mathbf{k} \rangle$ to the context, the compiler arranges for an opaque handle $\langle \mathbf{n}', \mathbf{k}_{\mathbf{com}} \rangle$ (that cannot be used to access any location directly) to be passed. Such an opaque handle is often called a *mask* or *seal* in the literature [66].

To ensure that masking is done properly, $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathsf{L}^{\mathsf{P}}}$ can insert code at entry and exit points of compiled code, *wrapping* the compiled code in a way that enforces masking [32,59]. The wrapper keeps a list $\overline{\mathbf{L}}$ of component-allocated locations that are shared with the context in order to know their masks. When a component-allocated location is shared, it is added to the list $\overline{\mathbf{L}}$. The mask of a location is its index in this list. If the same location is shared again it is not added again but its previous index is used. To implement lookup in $\overline{\mathbf{L}}$ we must compare capabilities too, so we need to add that expression to the target language. To ensure capabilities do not leak to the context, the second field of the pair is a constant capability $\mathbf{k}_{\mathbf{com}}$ which compiled code does not use otherwise. Clearly, this wrapping can increase the cost of all cross-component calls and returns.

However, this wrapping is not sufficient to attain *FAC*. A component-allocated location could be passed to the context on the heap, so before passing control to the context the compiled code needs to *scan the whole heap* where a location can be passed and mask all found component-allocated locations. Dually, when receiving control the compiled code must scan the heap to unmask any masked location so it can use the location. The problem now is determining what parts of the heap to scan and how. Specifically, the compiled code needs to

keep track of all the locations (and related capabilities) that are shared, i.e., (i) passed from the context to the component and (ii) passed from the component to the context. Both keeping track of these locations as well as scanning them on every cross-component control transfer is likely to be *very* expensive.

Finally, masked locations cannot be used directly by the context to be read and written. Thus, compiled code must provide a **read** and a **write** function that implement reading and writing to masked locations. The additional unmasking in these functions (as opposed to native reads and writes) adds to the inefficiency.

It should be clear as opposed to the *RSC* compiler $[\![\cdot]\!]^{\mathbf{L}^{\mathrm{U}}}_{\mathbf{L}^{\mathrm{P}}}$ (Sect. 3), the *FAC* compiler $[\![\boxed{\cdot}]\!]^{\mathbf{L}^{\mathrm{U}}}_{\mathbf{L}^{\mathrm{P}}}$ just sketched is likely to generate far more inefficient code.

Proof Difficulty. Proving that $[\![\boxed{\cdot}]\!]^{\mathbf{L}^{\mathrm{U}}}_{\mathbf{L}^{\mathrm{P}}}$ attains *FAC* can only be done by backtranslating *traces*, not contexts alone, since the newly-added target expressions cannot be directly backtranslated to valid source ones [7,9,59]. For this, we need a trace semantics that captures all information available to the context. This is often called a fully abstract trace semantics [38,55,56]. However, the trace semantics we defined for $\mathbf{L}^{\mathrm{P}}$ is not fully abstract, as its actions record the entire heap in every action, including private parts of the heap. Hence, we cannot use this trace semantics for proving *FAC* and so we design a new one. Building a fully abstract trace semantics for $\mathbf{L}^{\mathrm{P}}$ is challenging because we have to keep track of locations that have been shared with the context in the past. This substantially complicates both the definition of traces and the proofs that build on the definition.

Finally, the source context that the backtranslation constructs from a target trace must simulate the shared part of the heap at every context switch. Since locations in the target may be masked, the source context has to maintain a map from the source locations to the corresponding masked target ones, which complicates the backtranslation and the proof substantially.

To summarize, it should be clear that the proof of *FAC* for $[\![\boxed{\cdot}]\!]^{\mathbf{L}^{\mathrm{U}}}_{\mathbf{L}^{\mathrm{P}}}$ would be much harder than the proof of *RSC* for $[\![\cdot]\!]^{\mathbf{L}^{\mathrm{U}}}_{\mathbf{L}^{\mathrm{P}}}$, even though the source and target languages are the same and so is the broad proof technique (backtranslation).

6 Related Work

Recent work [8,33] presents new criteria for secure compilation that ensure preservation of subclasses of hyperproperties. Hyperproperties [25] are a formal representation of predicates on programs, i.e., they are predicates on sets of traces. Hyperproperties capture many security-relevant properties including not just conventional safety and liveness, which are predicates on traces, but also properties like non-interference, which is a predicate on pairs of traces. Modulo technical differences, our definition of *RSC* coincides with the criterion of "robust safety property preservation" in [8,33]. We show, through concrete instances, that this criterion can be easily realized by compilers, and develop two proof

techniques for establishing it. We further show that the criterion leads to more efficient compiled code than does *FAC*. Additionally, the criteria in [8,33] assume that behaviours in the source and target are represented using the same alphabet. Hence, the definitions (somewhat unrealistically or ideally) do not require a translation of source properties to target properties. In contrast, we consider differences in the representation of behaviour in the source and in the target and this is accounted for in our monitor relation $\mathsf{M} \approx \mathbf{M}$. A slightly different account of this difference is presented by Patrignani and Garg [60] in the context of reactive black-box programs.

Abate *et al.* [7] define a variant of robustly-safe compilation called RSCC specifically tailored to the case where (source) components can perform undefined behaviour. RSCC does not consider attacks from arbitrary target contexts but from compiled components that can become compromised and behave in arbitrary ways. To demonstrate RSCC, Abate *et al.* [7] rely on two backends for their compiler: software fault isolation and tag-based monitors. On the other hand, we rely on capability machines and memory isolation (the latter in the companion report). RSCC also preserves (a form of) safety properties and can be achieved by relying on a trace-based backtranslation; it is unclear whether proofs can be simplified when the source is verified and concurrent, as in our second compiler.

ASLR [6,37], protected module architectures [9,42,53,59], tagged architectures [39], capability machines [69] and cryptographic primitives [4,5,22,26] have been used as targets for *FAC*. We believe all of these can also be used as targets of *RSC*-attaining compilers. In fact, some targets such as capability machines seem to be better suited to *RSC* than *FAC*, as we demonstrated.

Ahmed *et al.* prove full abstraction for several compilers between typed languages [10,11,50]. As compiler intermediate languages are often typed, and as these types often serve as the basis for complex static analyses, full abstraction seems like a reasonable goal for (fully typed) intermediate compilation steps. In the last few steps of compilation, where the target languages are unlikely to be typed, one could establish robust safety preservation and combine the two properties (vertically) to get an end-to-end security guarantee.

There are three other criteria for secure compilation that we would like to mention: securely compartmentalised compilation (SCC) [39], trace-preserving compilation (TPC) [60] and non-interference-preserving compilation (NIPC) [12,15,16,27]. SCC is a re-statement of the "hard" part of full abstraction (the forward implication), but adapted to languages with undefined behaviour and a strict notion of components. Thus, SCC suffers from much of the same efficiency drawbacks as *FAC*. TPC is a stronger criterion than *FAC*, that most existing fully abstract compilers also attain. Again, compilers attaining TPC also suffer from the drawbacks of compilers attaining *FAC*.

NIPC preserves a single property: noninterference (NI). However, this line of work does not consider active target-level adversaries yet. Instead, the focus is on compiling whole programs. Since noninterference is not a safety property, it is difficult to compare NIPC to *RSC* directly. However, noninterference can also

be approximated as a safety property [20]. So, in principle, *RSC* (with adequate massaging of observations) can be applied to stronger end-goals than NIPC.

Swamy *et al.* [67] embed an F* model of a gradually and robustly typed variant of JavaScript into an F* model of JavaScript. Gradual typing supports constructs similar to our endorsement construct in $\lfloor \cdot \rceil^{\tau}$. Their type-directed compiler is proven to attain memory isolation as well as static and dynamic memory safety. However, they do not consider general safety properties, nor a specific, general criterion for compiler security.

Two of our target languages rely on capabilities for restricting access to sensitive locations from the context. Although capabilities are not mainstream in any processor, fully functional research prototypes such as Cheri exist [71]. Capability machines have previously been advocated as a target for efficient secure compilation [30] and preliminary work on compiling C-like languages to them exists, but the criterion applied is *FAC* [69].

7 Conclusion

This paper has examined robustly safe compilation (*RSC*), a soundness criterion for compilers with direct relevance to security. We have shown that the criterion is easily realizable and may lead to more efficient code than does fully abstract compilation wrt contextual equivalence. We have also presented two techniques for establishing that a compiler attains *RSC*. One is an adaptation of an existing technique, backtranslation, and the other is based on inductive invariants.

Acknowledgements. The authors would like to thank Dominique Devriese, Akram El-Korashy, Cătălin Hriţcu, Frank Piessens, David Swasey and the anonymous reviewers for useful feedback and discussions on an earlier draft.

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

References

1. Abadi, M.: Secrecy by typing in security protocols. In: Abadi, M., Ito, T. (eds.) TACS 1997. LNCS, vol. 1281, pp. 611–638. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0014571
2. Abadi, M.: Protection in programming-language translations. In: Vitek, J., Jensen, C.D. (eds.) Secure Internet Programming. LNCS, vol. 1603, pp. 19–34. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48749-2_2
3. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. **13**(1), 4:1–4:40 (2009)
4. Abadi, M., Fournet, C., Gonthier, G.: Authentication primitives and their compilation. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2000, pp. 302–315. ACM, New York (2000)

5. Abadi, M., Fournet, C., Gonthier, G.: Secure implementation of channel abstractions. Inf. Comput. **174**, 37–83 (2002)
6. Abadi, M., Plotkin, G.D.: On protection by layout randomization. ACM Trans. Inf. Syst. Secur. **15**, 8:1–8:29 (2012)
7. Abate, C., et al.: When good components go bad: formally secure compilation despite dynamic compromise. In: CCS 2018 (2018)
8. Abate, C., Blanco, R., Garg, D., Hriţcu, C., Patrignani, M., Thibault, J.: Journey beyond full abstraction: exploring robust property preservation for secure compilation. arXiv:1807.04603, July 2018
9. Agten, P., Strackx, R., Jacobs, B., Piessens, F.: Secure compilation to modern processors. In: 2012 IEEE 25th Computer Security Foundations Symposium, CSF 2012, pp. 171–185. IEEE (2012)
10. Ahmed, A., Blume, M.: Typed closure conversion preserves observational equivalence. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, pp. 157–168. ACM, New York (2008)
11. Ahmed, A., Blume, M.: An equivalence-preserving CPS translation via multi-language semantics. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, pp. 431–444. ACM, New York (2011)
12. Almeida, J.B., et al.: Jasmin: high-assurance and high-speed cryptography. In: ACM Conference on Computer and Communications Security, pp. 1807–1823. ACM (2017)
13. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
14. Backes, M., Hritcu, C., Maffei, M.: Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. J. Comput. Secur. **22**(2), 301–353 (2014)
15. Barthe, G., Grégoire, B., Laporte, V.: Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In: CSF 2018 (2018)
16. Barthe, G., Rezk, T., Basu, A.: Security types preserving compilation. Comput. Lang. Syst. Struct. **33**, 35–59 (2007)
17. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. ACM Trans. Program. Lang. Syst. **33**(2), 8:1–8:45 (2011)
18. Benton, N., Hur, C.-K.: Realizability and compositional compiler correctness for a polymorphic language. Technical report, MSR (2010)
19. Berry, G., Boudol, G.: The chemical abstract machine. Theor. Comput. Sci. **96**(1), 217–248 (1992)
20. Boudol, G.: Secure information flow as a safety property. In: Degano, P., Guttman, J., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 20–34. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01465-9_2
21. Bowman, W.J., Ahmed, A.: Noninterference for free. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015. ACM, New York (2015)
22. Bugliesi, M., Giunti, M.: Secure implementations of typed channel abstractions. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, pp. 251–262. ACM, New York (2007)
23. Carter, N.P., Keckler, S.W., Dally, W.J.: Hardware support for fast capability-based addressing. SIGPLAN Not. **29**, 319–327 (1994)
24. Chong, S.: Expressive and enforceable information security policies. Ph.D. thesis, Cornell University, August 2008

25. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
26. Corin, R., Deniélou, P.-M., Fournet, C., Bhargavan, K., Leifer, J.: A secure compiler for session abstractions. J. Comput. Secur. **16**, 573–636 (2008)
27. Costanzo, D., Shao, Z., Gu, R.: End-to-end verification of information-flow security for C and assembly programs. In: PLDI, pp. 648–664. ACM (2016)
28. Devriese, D., Patrignani, M., Keuchel, S., Piessens, F.: Modular, fully-abstract compilation by approximate back-translation. Log. Methods Comput. Sci. **13**(4) (2017). https://lmcs.episciences.org/4011
29. Devriese, D., Patrignani, M., Piessens, F.: Secure compilation by approximate back-translation. In: POPL 2016 (2016)
30. El-Korashy, A.: A formal model for capability machines - an illustrative case study towards secure compilation to CHERI. Master's thesis, Universitat des Saarlandes (2016)
31. Fournet, C., Gordon, A.D., Maffeis, S.: A type discipline for authorization policies. ACM Trans. Program. Lang. Syst. **29**(5), 141–156 (2007)
32. Fournet, C., Swamy, N., Chen, J., Dagand, P.-E., Strub, P.-Y., Livshits, B.: Fully abstract compilation to JavaScript. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, pp. 371–384. ACM, New York (2013)
33. Garg, D., Hritcu, C., Patrignani, M., Stronati, M., Swasey, D.: Robust hyperproperty preservation for secure compilation (extended abstract). ArXiv e-prints, October 2017
34. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. J. Comput. Secur. **11**(4), 451–519 (2003)
35. Gorla, D., Nestman, U.: Full abstraction for expressiveness: history, myths and facts. Math. Struct. Comput. Sci. **26**(4), 639–654 (2016)
36. Hur, C.-K., Dreyer, D.: A Kripke logical relation between ML and assembly. SIGPLAN Not. **46**, 133–146 (2011)
37. Jagadeesan, R., Pitcher, C., Rathke, J., Riely, J.: Local memory via layout randomization. In: Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium, CSF 2011, Washington, DC, USA, pp. 161–174. IEEE Computer Society (2011)
38. Jeffrey, A., Rathke, J.: Java JR: fully abstract trace semantics for a core Java language. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 423–438. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_29
39. Juglaret, Y., Hriţcu, C., de Amorim, A.A., Pierce, B.C.: Beyond good and evil: formalizing the security guarantees of compartmentalizing compilation. In: 29th IEEE Symposium on Computer Security Foundations (CSF). IEEE Computer Society Press, July 2016. To appear
40. Kang, J., Kim, Y., Hur, C.-K., Dreyer, D., Vafeiadis, V.: Lightweight verification of separate compilation. In: POPL 2016, pp. 178–190 (2016)
41. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-pointer integrity. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI 2014, Berkeley, CA, USA, pp. 147–163. USENIX Association (2014)
42. Larmuseau, A., Patrignani, M., Clarke, D.: A secure compiler for ML modules. In: Feng, X., Park, S. (eds.) APLAS 2015. LNCS, vol. 9458, pp. 29–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26529-2_3
43. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL, pp. 42–54 (2006)

44. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. **43**(4), 363–446 (2009)
45. Maffeis, S., Abadi, M., Fournet, C., Gordon, A.D.: Code-carrying authorization. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 563–579. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88313-5_36
46. McKeen, F., et al.: Innovative instructions and software model for isolated execution. In: HASP 2013, pp. 10:1–10:1. ACM (2013)
47. Morris Jr., J.H.: Protection in programming languages. Commun. ACM **16**, 15–21 (1973)
48. Neis, G., Dreyer, D., Rossberg, A.: Non-parametric parametricity. SIGPLAN Not. **44**(9), 135–148 (2009)
49. Neis, G., Hur, C.-K., Kaiser, J.-O., McLaughlin, C., Dreyer, D., Vafeiadis, V.: Pilsner: a compositionally verified compiler for a higher-order imperative language. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, pp. 166–178. ACM (2015)
50. New, M.S., Bowman, W.J., Ahmed, A.: Fully abstract compilation via universal embedding. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pp. 103–116. ACM, New York (2016)
51. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, New York (1999). https://doi.org/10.1007/978-3-662-03811-6
52. Parrow, J.: General conditions for full abstraction. Math. Struct. Comput. Sci. **26**(4), 655–657 (2014)
53. Patrignani, M., Agten, P., Strackx, R., Jacobs, B., Clarke, D., Piessens, F.: Secure compilation to protected module architectures. ACM Trans. Program. Lang. Syst. **37**, 6:1–6:50 (2015)
54. Patrignani, M., Ahmed, A., Clarke, D.: Formal approaches to secure compilation a survey of fully abstract compilation and related work. ACM Comput. Surv. **51**(6), 125:1–125:36 (2019)
55. Patrignani, M., Clarke, D.: Fully abstract trace semantics of low-level isolation mechanisms. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC 2014, pp. 1562–1569. ACM (2014)
56. Patrignani, M., Clarke, D.: Fully abstract trace semantics for protected module architectures. Comput. Lang. Syst. Struct. **42**(0), 22–45 (2015)
57. Patrignani, M., Clarke, D., Piessens, F.: Secure compilation of object-oriented components to protected module architectures. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 176–191. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03542-0_13
58. Patrignani, M., Clarke, D., Sangiorgi, D.: Ownership types for the join calculus. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE -2011. LNCS, vol. 6722, pp. 289–303. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21461-5_19
59. Patrignani, M., Devriese, D., Piessens, F.: On modular and fully abstract compilation. In: Proceedings of the 29th IEEE Computer Security Foundations Symposium, CSF 2016 (2016)
60. Patrignani, M., Garg, D.: Secure compilation and hyperproperties preservation. In: Proceedings of the 30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, USA (2017)
61. Patrignani, M., Garg, D.: Robustly safe compilation or, efficient, provably secure compilation. CoRR, abs/1804.00489 (2018)
62. Sabelfeld, A., Sands, D.: Declassification: dimensions and principles. J. Comput. Secur. **17**(5), 517–548 (2009)

63. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000)
64. Stark, I.: Names and higher-order functions. Ph.D. thesis, University of Cambridge, December 1994. Also available as Technical Report 363, University of Cambridge Computer Laboratory
65. Stewart, G., Beringer, L., Cuellar, S., Appel, A.W.: Compositional compcert. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, pp. 275–287. ACM, New York (2015)
66. Sumii, E., Pierce, B.C.: A bisimulation for dynamic sealing. In: Principles of Programming Languages, pp. 161–172 (2004)
67. Swamy, N., Fournet, C., Rastogi, A., Bhargavan, K., Chen, J., Strub, P.-Y., Bierman, G.: Gradual typing embedded securely in Javascript. SIGPLAN Not. **49**(1), 425–437 (2014)
68. Swasey, D., Garg, D., Dreyer, D.: Robust and compositional verification of object capability patterns. In: Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017, 22–27 October 2017 (2017)
69. Tsampas, S., El-Korashy, A., Patrignani, M., Devriese, D., Garg, D., Piessens, F.: Towards automatic compartmentalization of C programs on capability machines. In: 2017 Workshop on Foundations of Computer Security, FCS 2017, 21 August 2017 (2017)
70. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**, 167–187 (1996)
71. Woodruff, J., et al.: The CHERI capability model: revisiting RISC in an age of risk. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture, ISCA 2014, Piscataway, NJ, USA, pp. 457–468. IEEE Press (2014)
72. Zdancewic, S.A.: Programming languages for information security. Ph.D. thesis, Cornell University (2002)

Compiling Sandboxes: Formally Verified Software Fault Isolation

Frédéric Besson[1(✉)], Sandrine Blazy[1], Alexandre Dang[1], Thomas Jensen[1], and Pierre Wilke[2]

[1] Inria, Univ Rennes, CNRS, IRISA, Rennes, France
frederic.besson@inria.fr
[2] CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA, Rennes, France

Abstract. Software Fault Isolation (SFI) is a security-enhancing program transformation for instrumenting an untrusted binary module so that it runs inside a dedicated isolated address space, called a sandbox. To ensure that the untrusted module cannot escape its sandbox, existing approaches such as Google's Native Client rely on a binary verifier to check that all memory accesses are within the sandbox. Instead of relying on *a posteriori* verification, we design, implement and prove correct a program instrumentation phase as part of the formally verified compiler CompCert that enforces a sandboxing security property *a priori*. This eliminates the need for a binary verifier and, instead, leverages the soundness proof of the compiler to prove the security of the sandboxing transformation. The technical contributions are a novel sandboxing transformation that has a well-defined C semantics and which supports arbitrary function pointers, and a formally verified C compiler that implements SFI. Experiments show that our formally verified technique is a competitive way of implementing SFI.

1 Introduction

Isolating programs with various levels of trustworthiness is a fundamental security concern, be it on a cloud computing platform running untrusted code provided by customers, or in a web browser running untrusted code coming from different origins. In these contexts, it is of the utmost importance to provide adequate isolation mechanisms so that a faulty or malicious computation cannot compromise the host or neighbouring computations.

There exists a number of mechanisms for enforcing isolation that intervene at various levels, from the hardware up to the operating system. Hypervisors [10], virtual machines [2] but also system processes [17] can ensure strong isolation properties, at the expense of costly context switches and limited flexibility in the interaction between components. Language-based techniques such as strong typing offer alternative techniques for ensuring memory safety, upon which access control policies and isolation can be implemented. This approach is implemented e.g. by the Java language for which it provides isolation guarantees, as proved by Leroy and Rouaix [21]. The isolation is fined-grained and very flexible but

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 499–524, 2019.
https://doi.org/10.1007/978-3-030-17184-1_18

the security mechanisms, e.g. stack inspection, may be hard to reason about [7]. In the web browser realm, JavaScript is dynamically typed and also ensures memory safety upon which access control can be implemented [29].

1.1 Software Fault Isolation

Software Fault Isolation (SFI) is an alternative for unsafe languages, e.g. C, where memory safety is not granted but needs to be enforced at runtime by program instrumentation. Pioneered by Wahbe *et al.* [35] and popularised by Google's Native Client [30,37,38], SFI is a program transformation which confines a software component to a memory sandbox. This is done by pre-fixing every memory access with a carefully designed code sequence which efficiently ensures that the memory access occurs within the sandbox. In practice, the sandbox is aligned and the sandbox addresses are thus of the form $0xYZ$ where Y is a fixed bit-pattern and Z is an arbitrary bit-pattern *i.e.*, $Z \in [0x0 \ldots 0, 0xF \ldots F]$. Hence, enforcing that memory accesses are within the sandbox range of addresses can be efficiently implemented by a *masking* operation which exploits the binary representation of pointers: it retains the lowest bits Z and sets the highest bits to the bit-pattern Y.

Traditionally, the SFI transformation is performed at the binary level and is followed by an *a posteriori* verification by a trusted SFI verifier [23,31,35]. Because the verifier can assume that the code has undergone the SFI transformation, it can be kept simple (almost syntactic), thereby reducing both verification time and the Trusted Computing Base (TCB). This approach to SFI can be viewed as a simple instance of Proof Carrying Code [25] where the compiler is untrusted and the binary verifier is either trusted or verified.

Traditional SFI is well suited for executing binary code from an untrusted origin that must, for an adequate user experience, start running as soon as possible. Google's Native Client [30,37] is a state-of-the-art SFI implementation which has been deployed in the Chrome web browser for isolating binary code in untrusted pages. ARMor [39] features the first fully verified SFI implementation where the TCB is reduced to the formal ARM semantics in the HOL proof-assistant [9]. RockSalt [24] is a formally verified implementation of an SFI verifier for the x86 architecture, demonstrating that an efficient binary verifier can be obtained from a machine-checked specification.

1.2 Software Fault Isolation Through Compilation

A downside of the traditional SFI approach is that it hinders most compiler optimisations because the optimised code no longer respects the simple properties that the SFI verifier is capable of checking. For example, the SFI verifier expects that every memory access is immediately preceded by a specific syntactic code pattern that implements the sandboxing operation. A semantically equivalent but syntactically different code sequence would be rejected. An alternative to the *a posteriori* binary verifier approach is Portable Software Fault Isolation (PSFI), proposed by Kroll *et al.* [16]. In this methodology, there is no verifier

to trust. Instead isolation is obtained by compilation with a machine-checked compiler, such as CompCert [18]. Portability comes from the fact that PSFI can reuse existing compiler back-ends and therefore target all the architectures supported by the compiler without additional effort.

PSFI is applicable in scenarios where the source code is available or the binary code is provided by a trusted third-party that controls the build process. For example, the original motivation for Proof Carrying Code [25] was to provide safe kernel extensions [26] as binary code to replace scripts written in an interpreted language. This falls within the scope of PSFI. Another PSFI scenario is when the binary code is produced in a controlled environment and/or by a trusted party. In this case, the primary goal is not to protect against an attacker trying to insert malicious code but to prevent honest parties from exposing a host platform to exploitable bugs. This is the case *e.g.* in the avionics industry, where software from different third-parties is integrated on the same host that needs to ensure strong isolation properties between tasks whose levels of criticality differ. In those cases, PSFI can deliver both security and a performance advantage. In Sect. 8, we provide experimental evidence that PSFI is competitive and sometimes outperforms SFI in terms of efficiency of the binary code.

1.3 Challenges in Formally Verified SFI

PSFI inserts the masking operations during compilation and does away with the *a posteriori* SFI verifier. The challenge is then to ensure that the security, enforced at an intermediate representation of the code, still holds for the running code. Indeed, compiler optimisation often breaks such security [33]. The insight of Kroll *et al.* is that a safety theorem of the compiled code (i.e., that its behaviour is well-defined) can be exploited to obtain a security theorem for that same compiled code, guaranteeing that it makes no memory accesses outside its sandbox. We explain this in more detail in Sect. 2.2.

One challenge we face with this approach is that it is far from evident that the sandboxing operations and hence the transformed program have well-defined behaviour. An unsafe language such as C admits undefined behaviours (e.g. bitwise operations on pointers), which means that it is possible for the observational behaviour of a program to differ depending on the level of optimisation. This is not a compiler bug: compilers only guarantee semantics preservation *if* the code to compile has a well-defined semantics [36]. Therefore, our SFI transformation must turn any program into a program with a well-defined semantics.

The seminal paper of Kroll *et al.* emphasises that the absence of undefined behaviour is a prerequisite but they do not provide a transformation that enforces this property. More precisely, their transformation may produce a program with undefined behaviours (*e.g.* because the input program had undefined behaviours). This fact was one of the motivation for the present work, and explains the need for a new PSFI technique. One difficulty is to remove undefined behaviours due to restrictions on pointer arithmetic. For example, bitwise operators on pointers have undefined C semantics, but traditional masking operations of SFI rely heavily on these operators. Another difficulty is to deal with

indirect function calls and ensure that, as prescribed by the C standard, they are resolved to valid function pointers. To tackle these problems, we propose an original sandboxing transformation which unlike previous proposals is compliant with the C standard [13] and therefore has well-defined behaviour.

1.4 Contributions

We have developed and proved correct CompCertSfi, the first full-fledged, fully verified implementation of SFI inside a C compiler. The SFI transformation is performed early in the compilation chain, thereby permitting the generated code to benefit from existing optimisations that are performed by the back-end. The technical contributions behind CompCertSfi can be summarised as follows.

- An original design and implementation of the SFI transformation based on well-defined pointer arithmetic and which supports function pointers. This novel design of the SFI transformation is necessary for the safety proof.
- A machine-checked proof of the **security** and **safety** of the SFI transformation. Our formal development is available online [1].
- A small, lightweight runtime system for managing the sandbox, built using a standard program loader and configured by compiler-generated information.
- Experimental evidence demonstrating that the portable SFI approach is competitive and sometimes even outperforms traditional SFI, in particular state-of-the-art implementations of (P)Native Client.

The rest of the paper is organised as follows. In Sect. 2, we present background information about the CompCert compiler (Sect. 2.1) and the PSFI approach (Sect. 2.2). Section 3 provides an overview of the layout of the sandbox and the masking operations implementing our SFI. In Sect. 4 we explain how to overcome the problem with undefined pointer arithmetic and define masking operations with a well-defined C semantics. Section 5 describes how control-flow integrity in the presence of function pointers can be achieved by a sligthly more flexible SFI policy which allows reads in well-defined areas outside the sandbox. Section 6 specifies the SFI policy in more detail, and describes the formal Coq proofs of safety and security. Section 7 presents the design of our runtime library and how it exploits compiler support. Experimental results are detailed in Sect. 8. Section 9 presents related work and Sect. 10 concludes.

2 Background

This section presents background information about the CompCert compiler [18] and the Portable Software Fault Isolation proposed by Kroll *et al.* [16].

2.1 CompCert

The CompCert compiler [18] is a machine-checked compiler programmed and proved correct using the Coq proof-assistant [22]. It compiles C programs down

$$
\begin{aligned}
\mathit{constant} \ni c &::= \mathit{i32} \mid \mathit{i64} \mid \mathit{f32} \mid \mathit{f64} \mid \&\mathit{gl} \mid \&\mathit{stk} \\
\mathit{chunk} \ni \kappa &::= \mathit{is}_8 \mid \mathit{iu}_8 \mid \mathit{is}_{16} \mid \mathit{iu}_{16} \mid \mathit{i}_{32} \mid \mathit{i}_{64} \mid \mathit{f}_{32} \mid \mathit{f}_{64} \\
\mathit{expr} \ni e &::= x \mid c \mid \triangleright e \mid e_1 \square e_2 \mid [e]_\kappa \\
\mathit{stmt} \ni s &::= \mathbf{skip} \mid x := e \mid [e_1]_\kappa := e_2 \mid \mathbf{return}\ e \mid x := e(e_1 \ldots, e_n)_\sigma \\
&\quad \mid \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid s_1; s_2 \mid \mathbf{loop}\ s \mid \{s\} \mid \mathbf{exit}\ n \mid \mathbf{goto}\ \mathit{lb}
\end{aligned}
$$

Fig. 1. CMINOR syntax

to assembly code through a succession of compiler passes which are shown to be semantics preserving. COMPCERT features an architecture independent front-end. The back-end supports four main architectures: x86, ARM, PowerPC and RiscV. To target all the back-ends without additional effort, our secure transformation is performed in the compiler front-end, at the level of the CMINOR language that is the last architecture-independent language of the COMPCERT compiler chain. Our transformation can obviously be applied on C programs by first compiling them into CMINOR, and then applying the transformation itself.

The CMINOR language is a minimal imperative language with explicit stack allocation of certain local variables [19]. Its syntax is given in Fig. 1. Constants range over 32-bit and 64-bit integers but also IEEE floating-point numbers. It is possible to get the address of a global variable *gl* or the address of the stack allocated local variables (i.e., *stk* denotes the address of the current stack frame). In COMPCERT parlance, a memory chunk κ specifies how many bytes need to be read (resp. written) from (resp. to) memory and whether the result should be interpreted as a signed or unsigned quantity. For instance, the memory chunk is_{16} denotes a 16-bit signed integer and f_{64} denotes a 64-bit floating-point number. In CMINOR, memory accesses, written $[e]_\kappa$, are annotated with the relevant memory chunk κ. Expressions are built from pseudo-registers, constants, unary ($\triangleright$) and binary ($\square$) operators. COMPCERT features the relevant unary and binary operators needed to encode the semantics of C. Expressions are side-effect free but may contain memory reads.

Instructions are fairly standard. Similarly to a memory read, a memory store $[e_1]_\kappa = e_2$ is annotated by a memory chunk κ. In CMINOR, a function call such as $e(e_1 \ldots, e_n)_\sigma$ represents an indirect function call through a function pointer denoted by the expression e, σ is the signature of the function and $e_1 \ldots, e_n$ are the arguments. A direct call is a special case where the expression e is a constant (function) pointer. CMINOR is a structured language and features a conditional, a block construct $\{s\}$ and an infinite loop **loop** s. Exiting the n^{th} enclosing loop or block can be done using an **exit** n instruction. CMINOR is structured but **goto**s towards a symbolic label *lb* are also possible. Returning from a function is done by a return instruction. CMINOR is equipped with a small-step operational semantics. The intra-procedural and inter-procedural control flows are modelled using an explicit continuation which therefore contains a call stack.

CompCert Soundness Theorem. Each compiler pass is proved to be semantics preserving using a simulation argument. Theorem 1 states semantics preservation.

Theorem 1 (Semantics Preservation). *If the compilation of program p succeeds and generates a target program tp, then for any behaviour beh of program tp there exists a behaviour of p, beh′, such that beh* improves *beh′.*

In this statement, a behaviour is a trace of observable events that are typically generated when performing external function calls. CompCert classifies behaviours depending on whether the program terminates normally, diverges or goes wrong. A *goes wrong* behaviour corresponds to a situation where the program semantics gets stuck (i.e., has an undefined behaviour). In this situation, the compiler has the liberty to generate a program with an *improved* behaviour i.e., the semantics of the transformed program may be more defined (i.e., it may not get stuck at all or may get stuck later on).

The consequence is that Theorem 1 is not sufficient to preserve a safety property because the target program *tp* may have behaviours that are not accounted for in the program *p* and could therefore violate the property. Corollary 1 states that in the absence of going-wrong behaviour, the behaviours of the target program are a subset of the behaviours of the source program.

Corollary 1 (Safety preservation). *Let p be a program and tp be a target program. Consider that none of the behaviours of p is a going-wrong behaviour. If the compilation of p succeeds and generates a target program tp, then any behaviour of program tp is a behaviour of p.*

As a consequence, any (safety) property of the behaviours of p is preserved by the target program *tp*. In Sect. 2.2, we show how the PSFI approach leverages Corollary 1 to transfer an isolation property obtained at the Cminor level to the assembly code.

Going-wrong behaviours in CompCert. As safety is an essential property of our PSFI transformation, we give below a detailed account of the going-wrong behaviours of the CompCert languages with a focus on Cminor.

Undefined evaluation of expressions. CompCert's runtime values are dynamically typed and defined below:

$$\mathit{values} \ni v ::= \mathbf{undef} \mid \mathbf{int}(i_{32}) \mid \mathbf{long}(i_{64}) \mid \mathbf{single}(f_{32}) \mid \mathbf{float}(f_{64}) \mid \mathbf{ptr}(b, o)$$

Values are built from numeric values (32-bit and 64-bit integers and floating point numbers), the **undef** value representing an indeterminate value, and pointer values made of a pair (b, o) where b is a memory block identifier and o is an offset which, depending on the architecture, is either a 32-bit or a 64-bit integer.

For Cminor, like all languages of CompCert, the unary ($\triangleright$) and binary ($\square$) operators are not total. They may directly produce going-wrong behaviours *e.g.* in case of division by $\mathbf{int}(0)$. They may also return **undef** if (i) the arguments are not in the right range *e.g.* the left-shift $\mathbf{int}(i) << \mathbf{int}(32)$; or (ii) the arguments are not well-typed *e.g.* $\mathbf{int}(i) +_{int} \mathbf{float}(f)$. Pointer arithmetic is strictly conforming to the C standard [13] and any pointer operation that is implementation-defined according to the standard returns **undef**.

$$
\begin{array}{ll}
\mathbf{ptr}(b,o) \pm \mathbf{long}(l) & = \mathbf{ptr}(b, o \pm l) \\
\mathbf{ptr}(b,o) - \mathbf{ptr}(b,o') & = \mathbf{long}(o - o') \\
\mathbf{ptr}(b,o)\,!{=}\mathbf{long}(0) & = \mathbf{tt} \quad \text{if } W(b,o) \\
\mathbf{ptr}(b,o) == \mathbf{long}(0) & = \mathbf{ff} \quad \text{if } W(b,o) \\
\mathbf{ptr}(b,o) \star \mathbf{ptr}(b,o') & = o \star o' \quad \text{if } W(b,o) \wedge W(b,o') \\
\mathbf{ptr}(b,o) == \mathbf{ptr}(b',o') & = \mathbf{ff} \quad \text{if } b \neq b' \wedge V(b,o) \wedge V(b',o') \\
\mathbf{ptr}(b,o)\,!{=}\mathbf{ptr}(b',o') & = \mathbf{tt} \quad \text{if } b \neq b' \wedge V(b,o) \wedge V(b',o') \\
\multicolumn{2}{c}{\text{where } \star \in \{<, \leq, ==, \geq, >, \,!{=}\}}
\end{array}
$$

Fig. 2. Pointer arithmetic in CompCert

The precise semantics of pointer operations is given in Fig. 2. For simplicity, we provide the semantics for a 64-bit architecture. Pointer operations are often only defined provided that the pointers are valid, written V, or weakly valid, written W. This validity condition requires that the offset o of a pointer $\mathbf{ptr}(b,o)$ is strictly within the bounds of the block b. The weakly valid condition refers to a pointer whose offset is either valid or one-past-the-end of the block b. Any pointer arithmetic operation that is not listed in Fig. 2 returns **undef**. This is in particular the case for bitwise operations which are typically used for the masking operation needed to implement SFI.

The indeterminate value **undef** is not *per se* a going-wrong behaviour. Yet, branching over a test evaluating to **undef**, performing a memory access over an **undef** address and returning **undef** from the `main` function are going-wrong behaviours.

Memory accesses are ruled by a unified memory model [20] that is used throughout the whole compiler. The memory is made of a collection of separated blocks. For a given block, each offset o below the block size is given a permission $p \in \{\mathbf{r}, \mathbf{w}, \dots\}$ and contains a memory value

$$mval \ni mv ::= \mathbf{undef} \mid \mathbf{byte}(b) \mid [\mathbf{ptr}(b,o)]_n$$

where b is a concrete byte value and $[\mathbf{ptr}(b,o)]_n$ represents the n^{th} byte of the pointer $\mathbf{ptr}(b,o)$ for $n \in \{1 \dots 8\}$. A memory write $storev(\kappa, m, a, v)$ is only defined if the address a is a pointer $\mathbf{ptr}(b,o)$ to an existing block b such that the memory locations $(b,o), \dots, (b, o+ \mid \kappa \mid -1)$ have the permission $\mathbf{w}$ and the offset o satisfies the alignment constraint of κ. A memory read $loadv(\kappa, m, a)$ is only defined under similar conditions with the additional restriction that not reading all the consecutive fragments of a pointer returns **undef**.

Control-flow transfers may go-wrong if the target of the control-flow transfer is not well-defined. Hence, a **goto** lb instruction goes wrong if, in the current function, there is no statement labelled by lb; and an **exit** n instruction goes wrong if there are less than n enclosing blocks around the statement containing the exit instruction. A conditional **if** e **then** s_1 **else** s_2 goes wrong if the expression e does not evaluate to $\mathbf{int}(i)$ for some i. Also, the execution goes wrong if the

last statement of a function is not a **return** instruction. Last but not least, a function call $x := e(e_1 \ldots, e_n)_\sigma$ goes wrong if the expression e does not evaluate to a pointer $\mathbf{ptr}(b, 0)$ where b is a function pointer with signature σ.

We show in Sect. 4 how our transformation ensures that pointer arithmetic and memory accesses are always well-defined. Section 5 shows how we make sure indirect calls are always correctly resolved. Section 6 shows that, together with other statically checkable verifications, our PSFI transformation rules out all possible going-wrong behaviours.

2.2 Portable Software Fault Isolation

Kroll, Stewart and Appel have pioneered the concept of Portable Software Fault Isolation (PSFI) [16] whereby SFI is enforced by a pass of the compiler front-end that is architecture independent. The main expected advantage is that isolation is implemented, once and for all, for any target architecture. Moreover, the generated code is optimised by the back-end passes of the compiler. Compared to traditional SFI, there is no architecture-specific binary verifier but instead the compiler enters the TCB. The key insight of Kroll *et al.* is to leverage a formally verified compiler, namely CompCert, to transfer a security proof of isolation obtained at the Cminor level through the compiler back-end, with minimal proof effort. In the following, we recall the only basic properties that a Cminor SFI transformation needs to satisfy so that isolation holds at assembly level.

In CompCert's terms, the sandbox is identified by a dedicated memory block *sb*. A Cminor program is secure (Property 1) under the condition that all its memory accesses are performed within the sandbox.

Property 1 (Program security). A Cminor program p is secure if all its memory accesses are within the sandbox block *sb*.

After compilation, the assembly code is secure if its observable behaviours are the same as the observable behaviours of the Cminor program. In order to apply CompCert's semantics preservation theorem (more precisely Corollary 1), it remains to ensure that the Cminor program has a well-defined semantics (Property 2).

Property 2 (Program safety). A Cminor program p is safe if all its behaviours are well-defined, i.e., not wrong.

Kroll *et al.* state Property 1 by means of an instrumented Cminor semantics which gets stuck in case of memory accesses outside the sandbox. They prove formally that the additional semantic safeguards are never triggered for a transformed program.

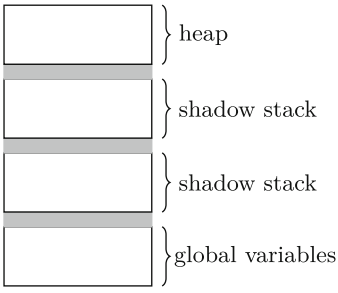
Kroll *et al.* also sketch some necessary steps to prove the Property 2 of safety but do not propose a formal proof. This leaves open a number of challenging issues such as whether it is feasible to define a masking operation that has a defined Cminor semantics and how to deal with indirect function calls through function pointers, More generally, the work leaves open whether a formal proof

of Property 2 on safety is possible given the restrictions of CompCert's semantics (notably pointer arithmetic) and without relying on axioms asserting properties of an external masking primitive. One of the central contributions of this work is to provide a positive answer to this question and propose solutions to these issues where neither the sandboxing of memory accesses nor the sandboxing of function pointers is part of a TCB. The transformation that circumvents the limitations imposed by pointer arithmetic is original and, we surmise, is a necessary component to transfer security down to assembly. For a precise comparison with Kroll *et al.* see Sect. 9).

3 A Thread-Aware Sandbox

The memory address space of a C program is partitioned into a runtime stack of frames, a heap and a dedicated space for global variables. The address space of a sandboxed program is re-organised to fit into a single global variable, *sb*, where the global variables, the heap and the stack frames are relocated. Figure 3a depicts the memory layout of the program after our SFI transformation. Each global variable is relocated and allocated in the sandbox at a given offset, and each global memory access of the program is translated into a memory access in the sandbox. For managing the heap it suffices to use a sandbox-aware `malloc` implementation that allocates memory inside the sandbox.

To prevent buffer overflows, a standard approach consists in introducing a so-called *shadow stack* that is used to store the function stack frames. Our implementation supports multi-threaded applications and therefore there are as many shadow stacks as there are threads. Upon thread creation, we allocate a novel shadow stack in the sandbox. The shadow-stack pointer is passed as an additional argument to each function call. This is efficient when arguments are passed by register, with the only drawback of reserving an additional register. Frames are allocated by incrementing the shadow-stack pointer at function entry. All accesses to the original stack are then translated into accesses to the sandbox shadow stack. The following Example 1 and the code snippet in Fig. 3 illustrate the essence of the transformation.



(a) Layout of memory

```
g = {long(5)};

long foo(){
  stk[8];
  bar(g, &stk);
  return([&stk]);
}
```

(b) Original Cminor

```
sb[2^k]= {long(5);...};

long foo(sp){
  sp1=sp + 8 ;
  bar(sp1,[&sb],sp);
  return([sp]);
}
```

(c) Sandboxed Cminor

Fig. 3. Sandbox transformation

Example 1. The Cminor program of Fig. 3b declares a global variable `g` initialised to the 64-bit integer 5. The function `foo` allocates a stack frame of 8 bytes that will be used to store a 64-bit local variable. By convention, the current stack frame is called `stk`. The function `foo` calls the function `bar` with as arguments the value of `g` and the address of the local variable `stk`; and returns the value, presumably updated by `bar`, of the local variable.

Syntactically, the program of Fig. 3c only performs memory accesses on the global sandbox `sb` variable. The size of `sb` variable is 2^k for some predefined k. At thread creation, a shadow stack is allocated by our sandbox-aware `malloc` in the sandbox after the statically allocated global variables. For our program, the unique global variable `g` is stored at offset 0 and spans over 8 bytes. Therefore, the initial value of the shadow-stack pointer `sp` is 8. After the transformation, the function `foo` reserves the space for the local variable `stk` by incrementing the pseudo-register `sp`. The function `bar` is called with the incremented shadow-stack pointer `sp1`, the value stored at offset 0 in the sandbox (i.e., the value of the global variable `g`) and the address of the local variable `stk` which is given by the value of the stack pointer `sp`. At function exit, the value of the local variable `stk` is returned by dereferencing the shadow-stack pointer `sp`.

Our SFI transformation enforces the isolation security policy stipulating that all memory accesses are performed within the sandbox *sb*—at the Cminor level. However, this holds because the semantics gets stuck (i.e., the semantics *goes wrong*) whenever the program performs an access outside the bounds of the sandbox. As explained earlier, the compiler is free to translate this into an insecure program that would escape the sandbox at runtime. To get a formal security guarantee, it is necessary to transform further the Cminor program to rule out any behaviour that *goes wrong* i.e., ensure Property 2. Given the numerous undefined behaviours of the C language, ruling out any *going-wrong* behaviour may seem a daunting task. In general, this requires to ensure both memory safety and control-flow integrity. The following two sections describe how we can exploit the SFI transformation and the knowledge that all memory accesses are inside the sandbox to ensure both memory safety and control-flow integrity.

4 Memory-Safe Masking

For SFI, memory safety is obtained by making sure that every memory access is performed inside the sandbox. Starting from an analysis of the standard SFI solution, we present our own design which satisfies the additional requirements of being compliant with the semantic restrictions of CompCert and with a strict interpretation of the C standard.

4.1 Standard SFI Masking of Addresses

Standard SFI transformations ensure memory safety by masking memory accesses. The gist of it is to allocate a sandbox *sb* of size 2^k at a 2^k aligned memory address, say $\&sb = tag \times 2^k$. Under those constraints, enforcing that an address A is within the bounds of the sandbox can essentially be done by replacing the high-address bits by those of *tag*. Using bitwise operations, this can be done by the expression $(A\&(2^k-1))|tag \times 2^k$, where & is the bitwise *and* and | is the bitwise *or*. More visually, this can be written $(A\&\underbrace{1\cdots1}_{k})|tag\,\underbrace{0\cdots0}_{k}$.

At binary level, this masking transformation is defined and the cost is modest: two bitwise operations. However, this masking operation has no well-defined C semantics. This is also the case for the semantics of CompCert and in particular for the Cminor language. The reason is twofold: bitwise operations over pointer values return **undef** and concrete addresses (e.g. $tag \times 2^k$) are not pointers for CompCert where they are represented by a block and an offset (see Fig. 2).

4.2 Specialised Masking for 32-Bit Sandboxes

For 32-bit sandboxes, there exists a variant of the sandboxing primitive which has the advantages (1) that the sandbox address does not need to be aligned; (2) that the cost of masking may be reduced to a single instruction. In its simplest form, the masking primitive is defined by

$$\&sb + (A - \&sb)_{64\to32\to64}$$

where $\&sb$ is the symbolic address of the sandbox. The subtraction of $\&sb$ extracts the offset of the pointer and the double (unsigned) cast $64 \to 32 \to 64$ has the effect of truncating the offset to a 32-bit quantity that is therefore within the bounds of a 32-bit sandbox. At first sight, this masking is less efficient than the standard masking but it is efficient for typical address computations which require both displacement and scaling (e.g. $A = t + k + k' * i_{32\to64}$ where t is a 64-bit address, k and k' are constants and i is a 32-bit integer). Assuming that each cast or arithmetic operation is mapped to a single instruction[1], the masked address A can be computed using 8 instructions: 4 instructions for computing the address A and 4 more for the sandboxing primitive. Using simple properties of modular arithmetic, it is possible to distribute the $64 \to 32$ cast over addition and multiplication to obtain the following equivalent formulation of the sandboxed address:

$$\&sb + A'_{32\to64} \quad with \quad A' = t_{64\to32} + c_1 + c_2 * i$$

where c_1 and c_2 are compile-time constants: $c_1 = (k - \&sb)_{64\to32}$ and $c_2 = k'_{64\to32}$. Using this formulation, the address A' still requires 4 instructions but the cost of the sandboxing is reduced to 2 instructions making it on par with the standard sandboxing. On x86, 32-bit registers are just zero-extended 64-bit registers. Therefore, the cast $A'_{32\to64}$ is actually redundant and the overhead induced by the sandboxing is reduced to a single instruction. Our experiments (see Sect. 8.2) validate the practical advantage of this encoding.

Still, as for the standard sandboxing, this sanboxing primitive has no semantics in CompCert due to the limitations of pointer arithmetic. As a consequence, the solution of Kroll *et al.* [16] does not give actual code for the masking primitive, but rather axiomatise its behaviour as an external function. This prevents optimisations such as common subexpression elimination or function inlining from happening and induces the cost of a function call for each memory access.

4.3 Towards Well-Defined Pointer Arithmetic

To illustrate the limitations of pointer arithmetic, we examine the semantic behaviour of the standard sandboxing primitive (the specialised sandboxing primitive has similar

[1] Some architecture have rich addressing modes allowing for more compact encodings.

issues). The standard sandboxing primitive can be written $(A\&(2^k-1)) \mid \&sb$ where $\&sb$ is the address of the sandbox variable. If sb is allocated at runtime at address $tag \times 2^k$ for some tag, this formulation is equivalent at binary level. Again, this heavily relies on pointer arithmetic that is undefined and on information about where the sandbox is linked at runtime.

Consider the alternative formulation $(A\&(2^k-1)) + \&sb$ where the bitwise $\mid$ is replaced by a $+$. This formulation has the advantage that incrementing a pointer, here sb, is well-defined (see Fig. 2). As on modern hardware, both addition and bitwise operations take a single cycle, the difference in efficiency should be negligible. Moreover, at least for x86, the addition can be compiled into the addressing mode.

Still, this does not solve our issue. To understand this, suppose that A is a pointer. In this case, the bitwise $\&$, whose purpose is to extract the pointer offset, is still undefined. Therefore, the whole expression $(A\&(2^k-1)) + \&sb$ is undefined. Because dereferencing an undefined expression is a *going-wrong* behaviour, the compiled program may have an arbitrary runtime behaviour and escape the sandbox. A prerequisite for our masking primitive is therefore to ensure that the evaluation is defined i.e., different from **undef**. As all the semantic operators of CompCert are strict in **undef** (if any argument is **undef**, so is the result), a necessary condition is that A is not **undef**. As A can be obtained from any expression, a challenge is to ensure that every expression evaluates to a defined value. A particular difficulty is that the many undefined pointer operations (see Fig. 2) cannot be detected by runtime checks.

4.4 Arithmetisation of the Heap

To tackle this challenge and ensure that every computation is defined, we propose an original and radical approach which ensures syntactically that pointers are neither stored in memory nor in local variables. As a result, the program is only manipulating integer values and memory addresses are only constructed by the sandboxing primitives. This approach implies, as a side-effect, that our previously undefined masking primitives are defined. Let asb be the runtime address of the symbolic address $\&sb$ of the sandbox. The masking of an address A can be written

$$A' + \&sb$$

where A' is either defined by $A' = A\&(2^k-1)$ or $A' = (A - asb)_{64\to 32\to 64}$. As A is necessarily an integer, A' is necessarily a defined integer and therefore $A' + \&sb$ returns a defined pointer $\mathbf{ptr}(sb, o)$ that is necessarily inside the sandbox.

An additional subtlety is that memory accesses are indexed by a memory chunk κ which mandates an alignment constraint (e.g. the chunk i_{64} mandates an 8-byte aligned address). As a result, the masking primitive is parameterised by the chunk κ and the masking primitive for i_{64} is $A'\&msk_{i_{64}} + \&sb$ where $msk_{i_{64}} = (2^{k-3}-1) \times 2^3$.

Only computing over numeric values is facilitated by the fact that the sandboxed program is only manipulating pointers relative to a single object, the sandbox. Therefore, a solution could be to only compute with pointer offsets. This is not totally satisfactory because the null pointer (i.e., 0) would be undistinguishable from the base pointer $\mathbf{ptr}(sb, 0)$. Instead, we use the integer asb that is the integer runtime address of the sandbox (i.e., we have $asb = \&sb$) and perform the following transformation t over program expressions.

$$\begin{array}{ll} t(\&sb) & = asb \\ t(c) & = c \text{ for } c \in \{i32, i64, f32, f64\} \\ t(\triangleright e) & = \blacktriangleright\, t(e) \\ t(e_1 \square e_2) & = t(e_1) \blacksquare t(e_2) \\ t([e]_\kappa) & = [msk_\kappa(t(e))] \end{array}$$

The operators ▶ and ■ ensure that, if the expressions are well-typed, they never return the **undef** value. Typical examples include division, modulus, and bitwise shifts. We transform expressions so that they evaluate to an arbitrary value when their original semantics is undefined. For example, we transform the left-shift operations on 32-bit integers so that the resulting expression always has a shift amount less than 32:

$$\texttt{a≪b} \quad \rightsquigarrow \quad \texttt{a ≪ (b \& 31).}$$

Similarly, we transform divisions and modulus in the following way, to rule out the undefined cases of division by zero and signed division of `MIN_SIGNED` by `-1`:

$$\texttt{a/b} \rightsquigarrow \texttt{(a+(a==MIN\_SIGNED \& b==-1))/(b+(b==0)).}$$

We can prove that the resulting division expression is always defined. Most of the other expressions are always defined and do not need further transformations.

5 Enforcement of Control-Flow Integrity

Correct sandboxing of code requires some degree of control-flow integrity. Existing SFI implementations enforce a weak form of control-flow integrity which only ensures that jumps are aligned and within a sandbox of code. This is achieved by inserting a masking operation before indirect jumps, that will mask the target address to ensure that the jump is within the sandbox. Additional padding with no-ops is inserted to ensure that all the instructions are indeed aligned [30,37,38]. We enforce a stronger, more traditional, form of control-flow integrity where any control-flow transfer has a well-defined Cminor semantics.

5.1 Relaxation of the Cminor SFI Property

Intraprocedural control-flow integrity is ensured by simple syntactic checks. For instance, they ensure that a **goto** *lb* has a corresponding label *lb* and that an **exit** *n* has at least *n* enclosing blocks. The semantics of Cminor prescribes that function calls and returns necessarily match. For this to still hold at the assembly level where the return address is explicitly stored in the stack frame, it is sufficient to prove that the Cminor program has no *going-wrong* behaviour. To ensure control-flow integrity, the only remaining issue is due to indirect calls through function pointers. Our control-flow integrity counter-measure implements software trampolines and ensures that an indirect call with signature σ can only be resolved by a function pointer towards a function with signature σ.

For this purpose, the existing Cminor SFI security policy i.e., Property 1, which rules out any memory access outside the sandbox is too restrictive. As we shall see, the implementation of trampolines necessitates controlled memory reads, outside the sandbox, within compiler-generated variables. To accommodate for this extension, we propose a slightly relaxed SFI security property which, in addition to memory accesses inside the sandbox, authorises other memory reads in read-only regions.

Property 3. A CMINOR program is secure if all its memory accesses are within either the sandbox block *sb* or some read-only memory.

This relaxed property still ensures the integrity of the runtime because all memory writes are confined to the sandbox. Note that Property 3 and Property 1 are equivalent if the trusted runtime library has no read-only memory. This can be achieved at modest cost by modifying slightly the source code and remove the C type qualifier `const` which instructs the compiler that the memory is read-only.

5.2 Control-Flow Integrity of Indirect Calls

In Sect. 4, we have eluded the presence of function pointers. They actually perfectly fit our strategy of encoding pointers by integers. In this case, each function pointer is encoded as an index and the trampoline code translates the index into a valid function pointer.

Consider a function f of signature σ and suppose that the function pointer $\&f$ is compiled into the index i. The reverse mapping from indexes to function pointers is obtained from a compiler-generated array variable A_σ such that $A_\sigma[i] = \&f$. The array variable A_σ is made of all the function pointers with signature σ. The array variable is also padded with a default function pointer such that its length is a power of two. At the call site, the instruction $e(e_1 \ldots, e_n)_\sigma$ is transformed into $[te\&msk_\sigma + \&A_\sigma](te_1, \ldots, te_n)_\sigma$ where $te, te_1 \ldots, te_n$ are transformed expressions such that all memory accesses are masked and msk_σ is the binary mask ensuring that the index te is within the bounds of the variable A_σ. In our actual implementation, we optimise direct calls and in this case bypass the trampoline. Therefore, when the expression e is a constant pointer $\&f$ to an existing function with signature σ, we generate directly $(\&f)(te_1 \ldots, te_n)$. As a result, only C code using indirect calls goes through the trampoline code.

Though our implementation only exploits the relaxation of Property 3 for the sake of trampolines, a more aggressive implementation could sometimes avoid to relocate read-only memory inside the sandbox. This could have a positive impact on optimisations which exploit the immutability of read-only memory.

6 Safety and Security Proofs

We next give an overview of our fully verified Coq proof of security and safety.

6.1 Security Proof

Property 3 is an informal formulation of our security property that is formally stated as a CMINOR instrumented semantics. This semantics mimics the CMINOR semantics with the exception that memory accesses are restricted: a memory read is either performed within the sandbox or in a read-only memory region; a memory write is necessarily performed within the sandbox.

The goal of the security proof is to show that all the memory accesses abide by the restrictions of the instrumented semantics. This is stated by Theorem 2 which establishes that for a transformed program *tp*, no behaviour of the standard CMINOR semantics gets stuck for the instrumented CMINOR semantics.

Theorem 2 (Security). *For any transformed program tp, every behaviour of tp in the standard semantics of* Cminor *is also a behaviour of tp in the instrumented semantics.*

The proof is based on the standard technique of forward simulation that is used in CompCert to ensure the preservation of semantics by compiler passes. Here, the forward simulation has the distinctive feature of relating the same (transformed) program equipped with a standard and an instrumented semantics. Since the only difference between the two semantics is that memory accesses must be secure, the crux of the proof lies in the correctness of the masking primitive, as stated in the following lemma.

Lemma 1. *For any masked expression e, if e evaluates to some pointer* $\mathbf{ptr}(b, o)$*, then b is the block of the sandbox i.e., sb.*

The proof relies on the definition of the masking primitive: a masked expression e is of the form $e' + \&sb$. Since $\&sb$ evaluates to the pointer $\mathbf{ptr}(sb, 0)$, then if the whole expression evaluates to a pointer $\mathbf{ptr}(b, o)$, necessarily $b = sb$.

6.2 Safety Proof

In order to benefit from CompCert's semantic preservation theorem and transport our security proof to the compiled assembly program, we must also prove that the sandboxed program is safe, i.e., it never gets *stuck*. We address all the going-wrong behaviours that we enumerated in Sect. 2.1. The well-formedness properties of a program (calling only defined functions, accessing only defined variables, jumping only to defined labels, exiting from no more blocks than currently enclosed in) are checked statically and make the transformation fail if they are violated. Next, the memory accesses require the addresses to be valid and adequately aligned: our masking operation ensures that this is always the case. Then, the evaluation of expressions must always be defined: this has mostly been dealt with the arithmetisation of the memory (Sect. 4.4). Finally, function calls should always be performed with the appropriate number of well-typed arguments. This is easy to check statically for direct function calls, but requires trampolines (as described in Sect. 5.2) for indirect function calls. The following sandbox invariant encapsulates all these conditions.

Definition 1 (Sandbox Invariant). *A state S of program P satisfies the sandbox invariant if the following conditions are satisfied:*

1. *indirect control-flow transfers are well-defined in P (e.g.* `goto` *instructions in the functions of P only jump to defined labels);*
2. *every function of P ends with an explicit return;*
3. *every function of P is well-typed;*
4. *every function of P starts by explicitly initialising its local variables;*
5. *the global array* A_σ *for signature* σ *contains function pointers to functions of signature* σ*;*
6. *the environment for local variables and the memory in S only contain properly initialised, numerical values.*

Properties 1, 2, 3 are ensured by a set of syntactic checks over the bodies of all the functions of the program. Property 4 is enforced by our function transformation which inserts assignments that explicitly initialise all declared local variables. Property 5 is ensured by construction of the arrays for function pointers. All these properties can be established solely on the program body and do not change during the execution of the program. By contrast, Property 6 cannot be checked statically and depends on the state of the program at each point.

Safe Evaluation of Expressions. A necessary condition for the safe evaluation of expressions is that the program is well typed. CompCert does not generate these type guarantees so we have integrated a verified (simple) type-inference algorithm for Cminor programs. Type-checking alone is not sufficient to rule out undefined behaviours of C operators, but together with the transformations explained in Sect. 4.4, we prove the following lemma about the evaluation of transformed expressions.

Lemma 2 (Safe evaluation of expressions). *In a memory state and a well-typed environment for local variables containing only defined numerical values, the transformation of any well-typed expression e evaluates to a defined numerical value.*

Lemma 2 follows directly from the properties of our expression transformation.

Safety of Calls through Trampolines. As mentioned in Sect. 5, we implement software trampolines to secure function calls through function pointers. To ensure the safety of indirect function calls, we maintain a map *smap* from function signatures to the corresponding array identifier and the length of this array. The proof of safety relies on the fact that for every function f of signature σ present in a program, we have $smap(\sigma) = (A_\sigma, l_\sigma)$ such that all offsets lower than l_σ in A_σ contain a pointer to a function of signature σ. The safety proof of indirect calls itself is not hard, but we need to set up this signature map and establish invariants relating it to the global environment of the program.

Safety Theorem. Considering the invariants defined in Definition 1, we prove Lemma 3 which is our main technical result.

Lemma 3 (Safety). *For any* Cminor *program state S that satisfies the invariants, either S is a final state or there exists a sequence of steps from S to some S' such that S' also satisfies the invariants.*

A subtlety of the proof is that at function entry, the local variables carry the value **undef** and therefore the sandbox invariant only holds after they have been initialised by a sequence of assignments (see Property 4 of Definition 1).

Using Lemma 3, we can show Property 2, in the form of Theorem 3.

Theorem 3 (Safety of the transformation). *All behaviours of the transformed program are well-defined, i.e., not wrong.*

Proof. A going-wrong behaviour occurs precisely when a state is reached, from which no further step can be taken, though it is not a final state. Lemma 3, together with a proof that the initial state of the transformed prorgam satisfies the invariants, tells us that no such reachable state exists, concluding the proof. □

As a result, we benefit from CompCert's semantic preservation theorem and can transport the security proof down to the assembly program.

Theorem 4 (Security of the compiled program). *Let p be a transformed* Cminor *program. If p compiles into the assembly program tp, then tp is secure.*

The proof uses Corollary 1 and Theorem 2 to conclude that the behaviours of tp are the same as those of p, and hence secure.

7 SFI Runtime and Library

Our modified CompCert compiler, CompCertSfi, takes as input a C program unit in the form of a list of C files. Each C file is first compiled down to the Cminor language using the existing passes of the CompCert compiler. Then, all the Cminor programs are syntactically linked [14] together to form the program unit to be isolated inside the sandbox. CompCertSfi comes with a lightweight runtime and a generic support for interfacing with a trusted library (e.g. a libC). An originality of our approach is that the runtime is using a standard program loader. Moreover, the runtime gets some of its configuration through compiler-generated variables.

7.1 Loading the SFI Application

The sandboxed code is linked with our runtime library by a linker script which specifies where to load at runtime the *sb* variable, viewed as the data segment. The compiler also emits a sandbox configuration map which contains the symbolic address of the sandbox, its numeric value at runtime, the total size of the sandbox and the range of addresses reserved for global variables.

Our runtime code is executed before starting the sandboxed `main` function. It first checks that the sandbox is properly linked according to the sandbox configuration map, sets the shadow-stack pointer and initialises the sandbox heap using our sandbox-aware implementation of `malloc` based on `ptmalloc3`[2].

By construction, our runtime stack is free of buffer overruns. Yet, if the recursion is too deep, the stack may overflow. Therefore, the runtime inserts an unmapped page guard at the bottom of the stack and intercepts the segmentation fault. This protection suffices provided that the size of each function stack frame does not exceed a page; which can be checked at compile-time. Eventually, after copying its arguments inside the sandbox, the runtime calls the `main` function of the sandboxed application.

7.2 Monitoring Calls to the Runtime Library

The runtime library is trusted and therefore part of the TCB. To ensure isolation, each call towards the runtime library is monitored to check the validity of the arguments. For this purpose, a call to a library function, say `foo`, is renamed in the object file into a call to a function `sb_foo` which sanitises its arguments before really calling the function `foo`. The verifications are library specific but usually straightforward to implement. For `stdio`, the `FILE` structures are allocated by the runtime outside of the sandbox. Hence, the returned `FILE*` cannot be dereferenced to corrupt the `FILE` structure. To prevent the sandboxed program to forge `FILE*` pointers, the runtime maintains at all time the set of valid `FILE*`. For variadic functions *e.g.*, `printf`, we statically compile the format into a sequence of safe primitive calls. (We reject programs using formats computed at runtime). For functions in `string`, we check beforehand that the range of memory accesses is within the range of the sandbox. We also allow callbacks and therefore a runtime function may take a function pointer as argument. To ensure that the function is valid, the runtime is using the trampoline programming pattern presented in Sect. 5.2.

[2] http://www.malloc.de/malloc/ptmalloc3-current.tar.gz.

7.3 Communication via Global Variables

Programs may not only communicate *via* function calls but also directly *via* global variables. For the libC, this includes e.g. `stdout` or `errno`. To ensure isolation, CompCertSfi relocates those variables inside the sandbox but also generates a global variable map which is an array variable of the form

$$\{\&n_1, o_1, \ldots, \&n_i, o_i, \ldots, \&n_m, o_m\}$$

where $\&n_i$ is the symbolic address of a global variable and o_i is its offset in the sandbox. Using this information, the runtime has the ability to synchronise the values of the variables inside and outside the sandbox. For example, at program startup, the value of `stdout` (a `stream` pointer) is copied inside the sandbox at the relevant offset. This allows the sandboxed program to call `stdio` functions but protects the integrity of the stream. For `errno`, it is the responsibility of each runtime library call to synchronise the value of `errno` in the sandbox.

8 Experiments

We have evaluated our PSFI approach over the CompCert benchmark suite and a port of Quake. All the experiments have been carried over a quad-core Intel 6600U laptop at 2.6 GHz with 16 GB of RAM running Linux Fedora 27. For Quake, we explain how to adapt the code to our runtime library and verify the absence of noticeable slowdown. For the other benchmarks, we make a more detailed performance evaluation and compare CompCertSfi with CompCert, gcc, clang but also the state-of-the-art (P)NaCl implementation of SFI. In our experiments, all the benchmarks are ordered by increasing running time. Moreover, for computing a runtime overhead, the running time is obtained by taking the harmonic mean of 3 consecutive runs.

8.1 Porting Quake

Quake engines come in various flavours and we use the tyr-quake[3] implementation linking with Xlib. The port requires the addition of several functions to our runtime library from Xlib and the libC. Most of them are not problematic and require no or little modification. For instance, the `getopt` function which is used to parse command-line options is using the global variables `optarg`, `optind`, `opterr`, and `optopt`. As explained in Sect. 7.3, the runtime library copies the values of these variables at reserved places inside the sandbox.

Other functions, *e.g.* `gethostbyname`, allocate memory on their own and return a pointer to this piece of data which is therefore not accessible to the sandboxed code. For the specific case of `gethostbyname`, the library provides the function `gethostbyname_r` which, instead of allocating memory, takes as argument a data-structure that is filled by the function. In our case, we pass as argument a sandbox allocated piece of memory. This does not solve our problem entirely as inner pointers may still point outside the sandbox. To cope with this issue, we perform a deep copy of the relevant piece of data inside the sandbox.

A last issue is that the video memory is shared between the application and the X server using the system call `shmat`. Fortunately, the libC provides the relevant flags to

[3] https://disenchant.net/git/tyrquake.git.

bind shared memory at a specific address. Hence, we were able to allocate it inside the sandbox thus allowing a seamless communication with the X server. After these modifications, the sandboxed Quake runs without noticeable slowdown which is encouraging and an indication of the good overall performance of our sandboxing technique. In the following, we complement this with a more precise runtime evaluation for the CompCert benchmarks.

8.2 PSFI Overhead: Impact of Sandboxing Primitives

Next, we compare the efficiency of a standard masking primitive (Sect. 4.1) with a specialised version for 32-bit sandboxes (Sect. 4.2).

Figure 4 shows the overhead of the standard sandboxing primitive with respect to the specialised sandboxing primitive. There are 6 benchmarks for which the overhead incurred by the standard sandboxing is above 10% reaching 40% for 2 benchmarks. These cases illustrate the significant performance advantage that is sometime obtained by the specialised sandboxing. For some benchmarks, the standard sandboxing outperforms our optimised sandboxing. Yet when it does it is by a very small margin (below 3%). Overall, for the vast majority of our benchmarks, the specialised sandboxing primitive is very competitive.

In Sect. 4.1, we gave theoretical arguments for the advantage of the specialised sandboxing. Another argument comes from the fact that the specialised sandboxing is easier to optimise. First, note that the standard and the specialised sandboxing primitives are both using a bitwise mask but for different purposes. For the standard primitive, it is used to enforce that the pointer is within the sandbox bounds but also to enforce alignment constraints. For the specialised primitive, it is only used to enforce alignment constraints. Using the existing CompCert dataflow framework, we have implemented an alignment analysis that is quite effective at removing redundant alignment masks. To enable more optimisations, we explicit alignment constraints in the Cminor code program (e.g. by specifying that function arguments of a pointer type are necessarily aligned). Thus, our experimental results are explained by both the theoretical advantages given in Sect. 4.2 and the effectiveness of our alignment analysis.

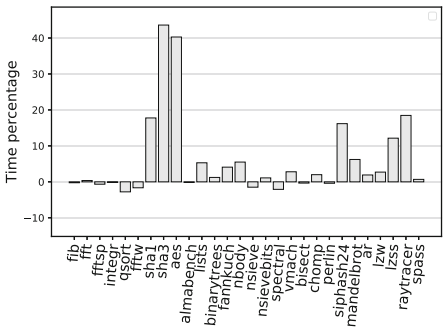


Fig. 4. Overhead of standard w.r.t specialised sandboxing

8.3 PSFI Overhead: Impact of Compiler Back-End

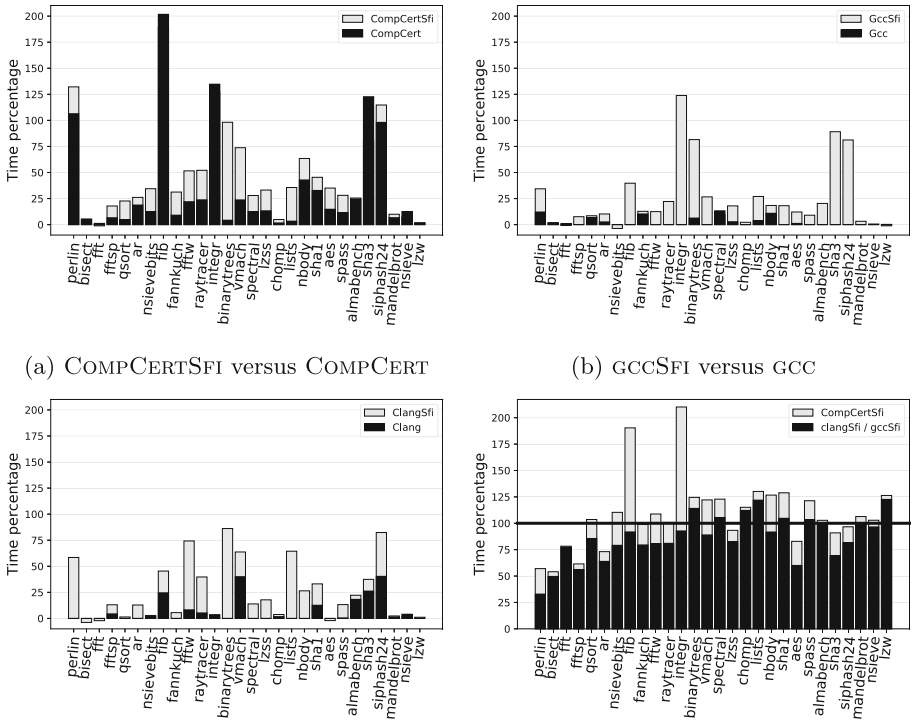
As a second experiment, we evaluate the overhead of our PSFI transformation for various compilers: CompCert, gcc and clang. CompCert is a *moderately optimising compiler* and the benchmarks run significantly faster using gcc and clang. In Fig. 5, the baseline is given by the minimum of the execution times of the three compilers without PSFI instrumentation. The black bar is the overhead of a compiler (e.g. CompCert), with respect to the baseline and the grey bar is the overhead of the same compiler but with the PSFI transformation (e.g. CompCertSfi). In order to use gcc and clang, we implement a trusted decompiler from our secured Cminor programs to Clight, a subset of C in CompCert. These Clight programs are then compiled with gcc or clang.

For a fair comparison, we should compare programs for which we actually have a reasonable security guarantee. We have a formal proof of security and safety (see Sect. 6) for the sandboxed Cminor program, and we are confident that our syntax-directed decompiler preserves this property. For CompCert, this would suffice to preserve the security of the compiled Clight code, but this is not the case for gcc and clang because of semantic discrepancies between the compilers. To limit this risk, we have set the compiler flags to instruct gcc and clang to adhere to the specificity of CompCert semantics: signed integer arithmetic is defined and so are wraps around (flag `-fwrapv`), strict aliasing is irrelevant (flag `-fno-strict-aliasing`), and floating-point arithmetic is strictly IEEE 754 compliant (flags `-frounding-math` and `-fsignaling-nans`). We also instruct the compilers to ignore any knowledge about the C library (`-fno-builtin`).

Our experimental results are shown in Fig. 5. In Fig. 5a, we have the overhead of CompCert and CompCertSfi. The overhead of CompCert over gcc and clang is expected and corroborates existing results[4]. For 10% of the benchmarks, the overhead CompCertSfi over CompCert is negligible and sometimes the PSFI transformation even improves performance. Those are programs for which the PSFI transformation introduces few masking operations, if any. For 41% of the benchmarks, the overhead is below 10% and can be considered, for most applications, a reasonable efficiency/security trade-off. For all the other benchmarks except `binarytrees` and `vmach`, the overhead is below 25%. The two remaining benchmarks have a significant overhead reaching 82% for `binarytrees`. This corresponds to programs which are memory intensive and where sandboxing cannot be optimised.

In Fig. 5b and c, we perform the same experiments but with gcc and clang. The results have some similarities but also have visible differences. For about 60% of the benchmarks the overhead is below 20%. Moreover, for both compilers, the average overhead is similar: 22% for gccSfi and 24% for clangSfi. Yet, on average gccSfi makes a better job at optimising our benchmarks and best clangSfi for about 75% of the benchmarks. For the rest of the benchmarks, we observe a significant overhead, up to 20%, indicating that the PSFI transformation hinders certain aggressive optimisations. The results also seem to indicate that optimisations are fragile as the overhead is not always consistent across compilers. The case of the `integr` benchmark is particularly striking because it runs with negligible overhead for clangSfi but exhibits the worst case overhead for gccSfi. The `integr` program is using a function pointer inside a loop and we suspect that gccSfi, unlike clangSfi, fails to optimise the program due to the inserted trampoline code. Though less striking, the benchmarks `fftw` and `raytracer` follow the opposite trend; these are programs where the overhead of clangSfi is much higher than gccSfi.

[4] http://compcert.inria.fr/compcert-C.html#perfs.



(a) CompCertSfi versus CompCert

(b) gccSfi versus gcc

(c) clangSfi versus clang

(d) PSFI versus (P)NaCl

Fig. 5. Overhead of PSFI:CompCert, clang, gcc, (P)NaCl

8.4 PSFI Versus (P)NaCl

We also compare our compiler-based SFI approach with (P)NaCl [30], which to our knowledge is one of the most mature implementations of SFI. Figure 5d shows the overhead of CompCertSfi, gccSfi, clangSfi with respect to (P)NaCl. The baseline is given by the best among NaCl and PNaCl. The best of clangSfi and gccSfi is given in dark gray and CompCertSfi is given in light grey.

We first analyse the results of CompCertSfi. Our benchmarks are ordered by increasing runtime. The first 5 benchmarks have a runtime below one second. They are not representative of the performance of both approaches but only illustrate the fact that (P)NaCl has a startup penalty due to the verification of the binary and the setup of the sandbox. The overhead peaks above 75% for two programs (i.e., `fib` and `integr`). As the PSFI transformation keeps `fib` unmodified and only inserts a trampoline call in `integr`, these programs only highlight the limited optimisations performed by CompCert. Of the remaining benchmarks, 40% of them run faster or have similar speed with CompCertSfi. For those benchmarks, the average overhead of CompCertSfi w.r.t (P)NaCl is around 9%. Except for a few programs whose overhead skyrockets due to CompCert not being specialised for speed, we can say that CompCertSfi performance is comparable to (P)NaCl, having programs with better speed in both sides and a large number having similar results.

We also matched gccSfi/clangSfi against (P)NaCl to compare the impact on performance of more aggressive optimisations. Here 60% of the programs are faster with gccSfi/clangSfi. Among the remaining programs, `lzw` and `chomp` are programs for which the (P)NaCl code runs faster than the optimised gcc clang code without the PSFI transformation. As (P)NaCl is based on clang, more investigation is needed to understand this paradox that may be explained by code running outside the sandbox *i.e.* the trusted runtime library. Among the remaining benchmarks, `binarytrees` and `lists` still show a noticeable overhead. Those are recursive micro-benchmarks for which our PSFI is costly (see Fig. 5). For `lists`, 99% of the time is spent in a tight loop where only a single address is masked. For `binarytrees`, 70% of the time is spent in the runtime code of `malloc` and `free` and therefore this highlights the fact that our implementation is less efficient than the (P)NaCl counterpart. Overall these results indicate that our implementation of SFI is competitive with (P)NaCl, given similar compilers. Furthermore speed can be improved with more sandbox-dedicated optimisations; these would be harder for (P)NaCl to check.

9 Related Work

Since Wahbe *et al.* [35] proposed their initial technique for SFI, there has been a number of proposals for efficiently confining untrusted software to a memory sandbox (see [23, 24,31,32,34,37,39]). One of the most prominent is Google's Native Client (NaCl) [37], which provides an infrastructure for executing untrusted native code in a web browser. NaCl was specifically targeted at executing computation-intensive applications without incurring a performance penalty. Certain features (in particular self-modifying code) were ruled out. These restrictions were addressed in a subsequent work [3].

RockSalt [24] is an SFI verifier for x86 code which has been developed and formally verified with the proof assistant Coq. The major contribution of RockSalt is to provide a formal model of the x86 architecture, from which it is possible to extract a decoder for a subset of the very rich set of x86 instructions, and build a verifier for the NaCl sandbox policy. Their experiments show that the formally verified checker performs marginally better than the NaCl verifier. In comparison, our approach avoids the complexities of the x86 instruction set by relying on the CompCert compiler back-end to produce binaries whose adherence to the sandbox policy is guaranteed by a combination of a sandbox verification at a higher level (Cminor) and the CompCert's correctness theorem.

ARMor [39] is using the binary rewriter Diablo [28] to implement SFI for ARM processors. Using an untrusted program analysis, a proof of SFI safety is automatically constructed using the HOL theorem prover. ARMor was tested with some programs of the MiBench benchmark [11], namely BitCount and StringSearch. These programs required 2.5 and 8 h respectively to prove the memory safety and control-flow integrity of the executables, which means that the approach is not practically viable as it is.

Kroll *et al.* [16] proposed PSFI as an alternative methodology to the standard, verification-based SFI. In PSFI, the sandbox is built by inserting the necessary masking instructions during compilation. This means that the correctness of the transformation can be argued at an intermediate stage in the compilation where the program representation retains a high-level structure. Our work extends the seminal proposal in a number of ways that we detail below. Unlike Kroll *et al.*, we exclude from the TCB the masking primitive and the trampoline mechanism for calling external functions. In our implementation, these crucial components are written entirely in Cminor and

proved correct without introducing trusted, unproved, code. Kroll *et al.* sketch a proof of safety but do not identify the issue of pointer arithmetic. To sidestep the semantics limitation of pointer arithmetic, we introduce a compile-time encoding of pointer as integers. This transformation is instrumental for our Coq verified proof of safety, which itself is mandatory to transfer security down to assembly.

Since the seminal work of Norrish [27], several works propose formal semantics of the C language [8,12,15]. All these share the limitations of CompCert with respect to pointer arithmetic. Recent works specifically aim at providing a more defined semantics for pointers. The proposal of Besson *et al.* [4] is able to cope with most existing low-level pointer manipulations and has been ported to CompCert [5,6]. Yet, it has nonetheless limitations and the design of our PSFI transformation would not benefit from the increased expressiveness. The semantics of Kang *et al.* [14] is more permissive because, after a cast, a pointer is indistinguishable from an integer value. To our knowledge, their semantics has not been ported to the CompCert compiler. Our SFI transformation has the advantage of being compatible with the existing semantics of CompCert with the caveat that pointers needs to be explicitly compiled into integers.

10 Conclusion

We have presented CompCertSfi, a formally verified implementation of Software Fault Isolation based on the CompCert compiler. Our approach provides security guarantees at runtime when the source code may be malicious or has security vulnerabilities but the build process is trusted. This is typically the case when a final product is built using code originating from multiple third parties. Our work shows that it is possible to perform security-enhancing compilation that is both formally verified and competitive with existing approaches in terms of efficiency. CompCertSfi does not rely on *a posteriori* binary verification for guaranteeing security, and hence has a reduced TCB compared to traditional SFI solutions. The reduction in TCB is obtained through a formal, machine-checked proof of the fact that the security guaranteed by our SFI transformation in the compiler front-end, still holds at the assembly level. Key to achieving this property has been to fine-tune the transformation (and in particular its pointer manipulations) to ensure that the secured program has a well-defined semantics.

The impact of SFI has been evaluated on a series of benchmarks, showing that the transformed code can in a few cases be more efficient, and that the average runtime overhead incurred is about 9%. We have evaluated the impact of back-end optimisation on the transformed code on three different compilers. The gains vary, with clang being more efficient than CompCert and gcc, and CompCert being slightly more efficient than gcc. The experiments show that CompCertSfi combined with an aggressive back-end optimiser can sometimes achieve performances superior to Native Client implementations. In addition, there is still room for further optimisation of the generated code. We have observed that existing optimisations are sometimes hindered by our SFI transformation, so we gain by having more optimisation before the SFI transformation. We also intend to investigate optimisations for removing redundant sandboxing operations and in particular hoisting sandboxing outside loops.

References

1. Supplementary material. https://www.irisa.fr/celtique/ext/compcertsfi
2. Andronick, J., Chetali, B., Ly, O.: Using Coq to verify Java Card^{TM} applet isolation properties. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 335–351. Springer, Heidelberg (2003). https://doi.org/10.1007/10930755_22
3. Ansel, J., et al.: Language-independent sandboxing of just-in-time compilation and self-modifying code. In: PLDI, pp. 355–366 (2011)
4. Besson, F., Blazy, S., Wilke, P.: A precise and abstract memory model for C using symbolic values. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 449–468. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12736-1_24
5. Besson, F., Blazy, S., Wilke, P.: CompCertS: a memory-aware verified C compiler using pointer as integer semantics. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 81–97. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_6
6. Besson, F., Blazy, S., Wilke, P.: A verified CompCert front-end for a memory model supporting pointer arithmetic and uninitialised data. J. Autom. Reasoning (2018, accepted for publication)
7. Besson, F., de Grenier de Latour, T., Jensen, T.P.: Interfaces for stack inspection. J. Funct. Program. **15**(2), 179–217 (2005)
8. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: POPL. ACM (2012)
9. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 243–258. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_18
10. Guanciale, R., Nemati, H., Dam, M., Baumann, C.: Provably secure memory isolation for Linux on ARM. J. Comput. Secur. **24**(6), 793–837 (2016)
11. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: MiBench: a free, commercially representative embedded benchmark suite, pp. 3–14. Institute of Electrical and Electronics Engineers Inc., United States (2001)
12. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of C. In: PLDI, pp. 336–345. ACM, June 2015
13. ISO: ISO C Standard 1999. Technical report (1999)
14. Kang, J., Kim, Y., Hur, C., Dreyer, D., Vafeiadis, V.: Lightweight verification of separate compilation. In: POPL, pp. 178–190. ACM (2016)
15. Krebbers, R.: An operational and axiomatic semantics for non-determinism and sequence points in C. In: POPL. ACM (2014)
16. Kroll, J.A., Stewart, G., Appel, A.W.: Portable software fault isolation. In: CSF, pp. 18–32. IEEE (2014)
17. Larus, J.R., Hunt, G.C.: The singularity system. Commun. ACM **53**(8), 72–79 (2010)
18. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
19. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. **43**(4), 363–446 (2009)
20. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert memory model. In: Program Logics for Certified Compilers. Cambridge University Press (2014)

21. Leroy, X., Rouaix, F.: Security properties of typed applets. In: Vitek, J., Jensen, C.D. (eds.) Secure Internet Programming, Security Issues for Mobile and Distributed Objects. LNCS, vol. 1603, pp. 147–182. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48749-2_7
22. The Coq development team: The Coq proof assistant reference manual (2017). http://coq.inria.fr, version 8.7
23. McCamant, S., Morrisett, G.: Evaluating SFI for a CISC architecture. In: Proceedings of the 15th Conference on USENIX Security Symposium, USENIX-SS 2006, vol. 15. USENIX Association (2006)
24. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: RockSalt: better, faster, stronger SFI for the x86. In: PLDI, pp. 395–404. ACM (2012)
25. Necula, G.C.: Proof-carrying code. In: POPL, pp. 106–119. ACM Press (1997)
26. Necula, G.C., Lee, P.: Safe kernel extensions without run-time checking. In: OSDI, pp. 229–243. ACM (1996)
27. Norrish, M.: C formalised in HOL. Ph.D. thesis, University of Cambridge (1998)
28. Put, L.V., Chanet, D., Bus, B.D., Sutter, B.D., Bosschere, K.D.: DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In: In IEEE International Symposium On Signal Processing And Information Technology (2005)
29. Richards, G., Hammer, C., Nardelli, F.Z., Jagannathan, S., Vitek, J.: Flexible access control for JavaScript. In: OOPSLA, pp. 305–322. ACM (2013)
30. Sehr, D., et al.: Adapting software fault isolation to contemporary CPU architectures. In: 19th USENIX Security Symposium, pp. 1–12. USENIX Association (2010)
31. Sehr, D., et al.: Adapting software fault isolation to contemporary CPU architectures. In: Proceedings of the 19th USENIX Conference on Security, USENIX Security 2010, p. 1. USENIX Association (2010)
32. Shu, R., et al.: A study of security isolation techniques. ACM Comput. Surv. **49**(3), 50:1–50:37 (2016)
33. Simon, L., Chisnall, D., Anderson, R.J.: What you get is what you C: controlling side effects in mainstream C compilers. In: EuroS&P, pp. 1–15. IEEE (2018)
34. Sinha, R., et al.: A design and verification methodology for secure isolated regions. In: PLDI, pp. 665–681. ACM (2016)
35. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: SOSP, pp. 203–216. ACM (1993)
36. Wang, X., Chen, H., Cheung, A., Jia, Z., Zeldovich, N., Kaashoek, M.: Undefined behavior: what happened to my code? In: APSYS (2012)
37. Yee, B., et al.: Native client: a sandbox for portable, untrusted x86 native code. In: S&P, pp. 79–93. IEEE (2009)
38. Yee, B., et al.: Native client: a sandbox for portable, untrusted x86 native code. Commun. ACM **53**(1), 91–99 (2010)
39. Zhao, L., Li, G., Sutter, B.D., Regehr, J.: ARMor: fully verified software fault isolation. In: EMSOFT, pp. 289–298. ACM (2011)

Fixing Incremental Computation

Derivatives of Fixpoints, and the Recursive Semantics of Datalog

Mario Alvarez-Picallo[1(✉)], Alex Eyers-Taylor[2], Michael Peyton Jones[2(✉)], and C.-H. Luke Ong[1]

[1] University of Oxford, Oxford, UK
{mario.alvarez-picallo,luke.ong}@cs.ox.ac.uk
[2] Semmle Ltd., Oxford, UK
alexet@semmle.com, me@michaelpj.com

Abstract. Incremental computation has recently been studied using the concepts of *change structures* and *derivatives* of programs, where the derivative of a function allows updating the output of the function based on a change to its input. We generalise change structures to *change actions*, and study their algebraic properties. We develop change actions for common structures in computer science, including directed-complete partial orders and Boolean algebras. We then show how to compute derivatives of fixpoints. This allows us to perform incremental evaluation and maintenance of recursively defined functions with particular application generalised Datalog programs. Moreover, unlike previous results, our techniques are *modular* in that they are easy to apply both to variants of Datalog and to other programming languages.

Keywords: Incremental computation · Datalog · Semantics · Fixpoints

1 Introduction

Consider the following classic Datalog program[1], which computes the transitive closure of an edge relation e:

$$tc(x, y) \leftarrow e(x, y)$$
$$tc(x, y) \leftarrow e(x, z) \wedge tc(z, y)$$

The semantics of Datalog tells us that the denotation of this program is the least fixpoint of the rule tc. Kleene's fixpoint Theorem tells us that we can compute this fixpoint by repeatedly applying the rule until the output stops changing, starting from the empty relation. For example, supposing that $e = \{(1, 2), (2, 3), (3, 4)\}$, we get the following evaluation trace:

[1] See [1, part D] for an introduction to Datalog.

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 525–552, 2019.
https://doi.org/10.1007/978-3-030-17184-1_19

Iteration	Newly deduced facts	Accumulated data in tc
0	$\{\}$	$\{\}$
1	$\{(1,2),(2,3),(3,4)\}$	$\{(1,2),(2,3),(3,4)\}$
2	$\{(1,2),(2,3),(3,4),$ $(1,3),(2,4)\}$	$\{(1,2),(2,3),(3,4),$ $(1,3),(2,4)\}$
3	$\{(1,2),(2,3),(3,4),$ $(1,3),(2,4),(1,4),(1,4)\}$	$\{(1,2),(2,3),(3,4),$ $(1,3),(2,4),(1,4)\}$
4	(as above)	(as above)

At this point we have reached a fixpoint, and so we are done.

However, this process is quite wasteful. We deduced the fact $(1,2)$ at every iteration, even though we had already deduced it in the first iteration. Indeed, for a chain of n such edges we will deduce $O(n^2)$ facts along the way.

The standard improvement to this evaluation strategy is known as "semi-naive" evaluation (see [1, section 13.1]), where we transform the program into a *delta* program with two parts:

- A *delta* rule that computes the *new* facts at each iteration.
- An *accumulator* rule that accumulates the delta at each iteration to compute the final result.

In this case our delta rule is simple: we only get new transitive edges at iteration $n+1$ if we can deduce them from transitive edges we deduced at iteration n.

$$\begin{aligned} \Delta tc_0(x,y) &\leftarrow e(x,y) \\ \Delta tc_{i+1}(x,y) &\leftarrow e(x,z) \wedge \Delta tc_i(z,y) \\ tc_0(x,y) &\leftarrow \Delta tc_0(x,y) \\ tc_{i+1}(x,y) &\leftarrow tc_i(x,y) \vee \Delta tc_{i+1}(x,y) \end{aligned}$$

Iteration	Δtc_i	tc_i
0	$\{(1,2),(2,3),(3,4)\}$	$\{(1,2),(2,3),(3,4)\}$
1	$\{(1,3),(2,4)\}$	$\{(1,2),(2,3),(3,4),$ $(1,3),(2,4)\}$
2	$\{(1,4)\}$	$\{(1,2),(2,3),(3,4),$ $(1,3),(2,4),(1,4)\}$
3	$\{\}$	(as above)

This is much better—we have turned a quadratic computation into a linear one. The delta transformation is a kind of *incremental computation*: at each stage we compute the changes in the rule given the previous changes to its inputs.

But the delta rule translation works only for traditional Datalog. It is common to liberalise the formula syntax with additional features, such as disjunction, existential quantification, negation, and aggregation.[2] This allows us to

[2] See, for example, LogiQL [26,32], Datomic [18], Souffle [38,42], and DES [36], which between them have all of these features and more. We do not here explore supporting extensions to the syntax of rule *heads*, although as long as this can be given a denotational semantics in a similar style our techniques should be applicable.

write programs like the following, where we compute whether all the nodes in a subtree given by *child* have some property p:

$$treeP(x) \leftarrow p(x) \wedge \neg\exists y.(child(x, y) \wedge \neg treeP(y))$$

The body of this predicate amounts to recursion through an *universal* quantifier (encoded as $\neg\exists\neg$). We would like to be able to use semi-naive evaluation for this rule too, but the standard definition of semi-naive transformation is not well defined for the extended program syntax, and it is unclear how to extend it (and the correctness proof) to handle such cases.

It is possible, however, to write a delta program for $treeP$ by hand; indeed, here is a definition for the delta predicate (the accumulator is as before):[3]

$$\begin{aligned}\Delta_{i+1} treeP(x) \leftarrow & p(x) \\ & \wedge \exists y.(child(x, y) \wedge \Delta_i treeP(y)) \\ & \wedge \neg\exists y.(child(x, y) \wedge \neg treeP_i(y))\end{aligned}$$

This is a *correct* delta program (in that using it to iteratively compute $treeP$ gives the right answer), but it is not *precise* because it derives some facts repeatedly. We will show how to construct correct delta programs generally using a program transformation, and show how we have some freedom to optimize within a range of possible alternatives to improve precision or ease evaluation.

Handling extended Datalog is of more than theoretical interest—the research in this paper was carried out at Semmle, which makes heavy use of a commercial Datalog implementation to implement large-scale static program analysis [7,37, 39,40]. Semmle's implementation includes parity-stratified negation[4], recursive aggregates [34], and other non-standard features, so we are faced with a dilemma: either abandon the new language features, or abandon incremental computation.

We can tell a similar story about *maintenance* of Datalog programs. Maintenance means updating the results of the program when its inputs change, for example, updating the value of tc given a change to e. Again, this is a kind of incremental computation, and there are known solutions for traditional Datalog [25], but these break down when the language is extended.

There is a piece of folkloric knowledge in the Datalog community that hints at a solution: the semi-naive translation of a rule corresponds to the *derivative* of that rule [8,9, section 3.2.2]. The idea of performing incremental computation using derivatives has been studied recently by Cai et al. [14], who give an account using *change structures*. They use this to provide a framework for incrementally evaluating lambda calculus programs.

[3] This rule should be read as: we can newly deduce that x is in $treeP$ if x satisfies the predicate, and we have newly deduced that one of its children is in $treeP$, and we currently believe that all of its children are in $treeP$.

[4] Parity-stratified negation means that recursive calls must appear under an even number of negations. This ensures that the rule remains monotone, so the least fixpoint still exists.

However, Cai et al.'s work isn't directly applicable to Datalog: the tricky part of Datalog's semantics are recursive definitions and the need for the *fixpoints*, so we need some additional theory to tell us how to handle incremental evaluation and maintenance of fixpoint computations.

This paper aims to bridge that gap by providing a solid semantic foundation for the incremental computation of Datalog, and other recursive programs, in terms of changes and differentiable functions.

Contributions. We start by generalizing change structures to *change actions* (Sect. 2). Change actions are simpler and weaker than change structures, while still providing enough structure to handle incremental computation, and have fruitful interactions with a variety of structures (Sects. 3 and 6.1).

We then show how change actions can be used to perform incremental evaluation and maintenance of non-recursive program semantics, using the formula semantics of generalized Datalog as our primary example (Sect. 4). Moreover, the structure of the approach is modular, and can accommodate arbitrary additional formula constructs (Sect. 4.3).

We also provide a method of incrementally computing and maintaining fixpoints (Sect. 6.2). We use this to perform incremental evaluation and maintenance of *recursive* program semantics, including generalized recursive Datalog (Sect. 7). This provides, to the best of our knowledge, the world's first incremental evaluation and maintenance mechanism for Datalog that can handle negation, disjunction, and existential quantification.

We have omitted the proofs from this paper. Most of the results have routine proofs, but the proofs of the more substantial results (especially those in Sect. 6.2) are included in an extended report [3], along with some extended worked examples, and additional material on the precision of derivatives.

2 Change Actions and Derivatives

Incremental computation requires understanding how values *change*. For example, we can change an integer by adding a natural to it. Abstractly, we have a set of values (the integers), and a set of changes (the naturals) which we can "apply" to a value (by addition) to get a new value.

This kind of structure is well-known—it is a set action. It is also very natural to want to combine changes sequentially, and if we do this then we find ourselves with a monoid action.

Using monoid actions for changes gives us a reason to think that change actions are an adequate representation of changes: any subset of $A \to A$ which is closed under composition can be represented as a monoid action on A, so we are able to capture all of these as change actions.

2.1 Change Actions

Definition 1. *A* change action *is a tuple:*

$$\hat{A} \coloneqq (A, \Delta A, \oplus_A)$$

where A is a set, ΔA is a monoid, and $\oplus_A : A \times \Delta A \to A$ is a monoid action on A.[5]

We will call A the base set, and ΔA the change set *of the change action. We will use $\cdot$ for the monoid operation of ΔA, and $\mathbf{0}$ for its identity element. When there is no risk of confusion, we will simply write $\oplus$ for $\oplus_A$.*

Examples. A typical example of a change action is $(A^*, A^*, +\!\!+)$ where A^* is the set of finite words (or lists) of A. Here we represent changes to a word made by concatenating another word onto it. The changes themselves can be combined using $+\!\!+$ as the monoid operation with the empty word as the identity, and this is a monoid action: $(a +\!\!+ b) +\!\!+ c = a +\!\!+ (b +\!\!+ c)$.

This is a very common case: any monoid $(A, \cdot, \mathbf{0})$ can be seen as a change action $(A, (A, \cdot, \mathbf{0}), \cdot)$. Many practical change actions can be constructed in this way. In particular, for any change action $(A, \Delta A, \oplus)$, $(\Delta A, \Delta A, \cdot)$ is also a change action. This means that we do not have to do any extra work to talk about changes to changes—we can always take $\Delta\Delta A = \Delta A$ (although there may be other change actions available).

Three examples of change actions are of particular interest to us. First, whenever L is a Boolean algebra, we can give it the change actions $(L, L, \vee)$ and $(L, L, \wedge)$, as well as a combination of these (see Sect. 3.2). Second, the natural numbers with addition have a change action $\hat{\mathbb{N}} \coloneqq (\mathbb{N}, \mathbb{N}, +)$, which will prove useful during inductive proofs.

Another interesting example of change actions is *semiautomata*. A semiautomaton is a triple (Q, Σ, T), where Q is a set of states, Σ is a (non-empty) finite input alphabet and $T : Q \times \Sigma \to Q$ is a transition function. Every semiautomaton corresponds to a change action (Q, Σ^*, T^*) on the free monoid over Σ^*, with T^* being the free extension of T. Conversely, every change action $\hat{A}$ whose change set ΔA is freely generated by a finite set corresponds to a semiautomaton.

Other recurring examples of change actions are:

- $\hat{A}_\bot \coloneqq (A, M, \lambda(a, \delta a).a)$, where M is any monoid, which we call the *empty* change action on any base set, since it induces no changes at all.
- $\hat{A}_\top \coloneqq (A, A \to A, \mathrm{ev})$, where A is an arbitrary set, $A \to A$ denotes the set of all functions from A into itself, considered as a monoid under composition and ev is the usual evaluation map. We will call this the "full" change action on A since it contains every possible non-redundant change.

These are particularly relevant because they are, in a sense, the "smallest" and "largest" change actions that can be imposed on an arbitrary set A.

Many other notions in computer science can be understood naturally in terms of change actions, *e.g.* databases and database updates, files and diffs, Git repositories and commits, even video compression algorithms that encode a frame as a series of changes to the previous frame.

[5] Why not just work with monoid actions? The reason is that while the category of monoid actions and the category of change actions have the same objects, they have different morphisms. See Sect. 8.1 for further discussion.

2.2 Derivatives

When we do incremental computation we are usually trying to save ourselves some work. We have an expensive function $f : A \to B$, which we've evaluated at some point a. Now we are interested in evaluating f after some change δa to a, but ideally we want to avoid actually computing $f(a \oplus \delta a)$ directly.

A solution to this problem is a function $f' : A \times \Delta A \to \Delta B$, which given a and δa tells us how to change $f(a)$ to $f(a \oplus \delta a)$. We call this a *derivative* of a function.

Definition 2. *Let $\hat{A}$ and $\hat{B}$ be change actions. A* derivative *of a function $f : A \to B$ is a function $f' : A \times \Delta A \to \Delta B$ such that*

$$f(a \oplus_A \delta a) = f(a) \oplus_B f'(a, \delta a)$$

A function which has a derivative is differentiable*, and we will write $\hat{A} \to \hat{B}$ for the set of differentiable functions between A and B.*[6]

Derivatives need not be unique in general, so we will speak of "a" derivative. Functions into "thin" change actions—where $a \oplus \delta a = a \oplus \delta b$ implies $\delta a = \delta b$—have unique derivatives, but many change actions are not thin. For example, $(\mathcal{P}(\mathbb{N}), \mathcal{P}(\mathbb{N}), \cap)$ is not thin because $\{0\} \cap \{1\} = \{0\} \cap \{2\}$.

Derivatives capture the structure of incremental computation, but there are important operational considerations that affect whether using them for computation actually saves us any work. As we will see in a moment (Proposition 1), for many change actions we will have the option of picking the "worst" derivative, which merely computes $f(a \oplus \delta a)$ directly and then works out the change that maps $f(a)$ to this new value. While this is formally a derivative, using it certainly does not save us any work! We will be concerned with both the possibility of constructing correct derivatives (Sects. 3.2 and 6.2 in particular), and also in giving ourselves a range of derivatives to choose from so that we can soundly optimize for operational value.

For our Datalog case study, we aim to cash out the folkloric idea that incremental computation functions via a derivative. We will construct a derivative of the semantics of Datalog in stages: first the non-recursive formula semantics (Sect. 4); and later the full, recursive, semantics (Sect. 7).

2.3 Useful Facts About Change Actions and Derivatives

The Chain Rule. The derivative of a function can be computed compositionally, because derivatives satisfy the standard chain rule.

[6] Note that we do not require that $f'(a, \delta a \cdot \delta b) = f'(a, \delta a) \cdot f'(a \oplus \delta a, \delta b)$ nor that $f'(a, \mathbf{0}) = \mathbf{0}$. These are natural conditions, and all the derivatives we have studied also satisfy them, but none of the results on this paper require them to hold.

Theorem 1 (The Chain Rule). *Let $f : \hat{A} \to \hat{B}$, $g : \hat{B} \to \hat{C}$ be differentiable functions. Then $g \circ f$ is also differentiable, with a derivative given by*

$$(g \circ f)'(x, \delta x) = g'\,(f(x), f'(x, \delta x))$$

or, in curried form

$$(g \circ f)'(x) = g'(f(x)) \circ f'(x)$$

Complete change actions and minus operators. Complete change actions are an important class of change actions, because they have changes between *any* two values in the base set.

Definition 3. *A change action is* complete *if for any $a, b \in A$, there is a change $\delta a \in \Delta A$ such that $a \oplus \delta a = b$.*

Complete change actions have convenient "minus operators" that allow us to compute the difference between two values.

Definition 4. *A* minus operator *is a function $\ominus : A \times A \to \Delta A$ such that $a \oplus (b \ominus a) = b$ for all $a, b \in A$.*

Proposition 1. *Given a minus operator $\ominus$, and a function f, let*

$$f'_{\ominus}(a, \delta a) \coloneqq f(a \oplus \delta a) \ominus f(a)$$

Then $f'_{\ominus}$ is a derivative for f.

Proposition 2. *Let $\hat{A}$ be a change action. Then the following are equivalent:*

- *$\hat{A}$ is complete.*
- *There is a minus operator on $\hat{A}$.*
- *For any change action $\hat{B}$ all functions $f : B \to A$ are differentiable.*

This last property is of the utmost importance, since we are often concerned with the differentiability of functions.

Products and sums. Given change actions on sets A and B, the question immediately arises of whether there are change actions on their Cartesian product $A \times B$ or disjoint union $A + B$. While there are many candidates, there is a clear "natural" choice for both.

Proposition 3 (Products). *Let $\hat{A} = (A, \Delta A, \oplus_A)$ and $\hat{B} = (B, \Delta B, \oplus_B)$ be change actions.*

Then $\hat{A} \times \hat{B} \coloneqq (A \times B, \Delta A \times \Delta B, \oplus_\times)$ is a change action, where $\oplus_\times$ is defined by:

$$(a, b) \oplus_{A \times B} (\delta a, \delta b) \coloneqq (a \oplus_A \delta a, b \oplus_B \delta b)$$

The projection maps π_1,π_2 are differentiable with respect to it. Furthermore, a function $f : A \times B \to C$ is differentiable from $\hat{A} \times \hat{B}$ into $\hat{C}$ if and only if, for every fixed $a \in A$ and $b \in B$, the partially applied functions

$$f(a, \cdot) : B \to C$$
$$f(\cdot, b) : A \to C$$

are differentiable.

Whenever $f : A \times B \to C$ is differentiable, we will sometimes use $\partial_1 f$ and $\partial_2 f$ to refer to derivatives of the partially applied versions, i.e. if $f'_a : B \times \Delta B \to \Delta C$ and $f'_b : A \times \Delta A \to \Delta C$ refer to derivatives for $f(a, \cdot), f(\cdot, b)$ respectively, then

$$\partial_1 f : A \times \Delta A \times B \to \Delta C$$
$$\partial_1 f(a, \delta a, b) \coloneqq f'_b(a, \delta a)$$
$$\partial_2 f : A \times B \times \Delta B \to \Delta C$$
$$\partial_2 f(a, b, \delta b) \coloneqq f'_a(b, \delta b)$$

Proposition 4 (Disjoint unions). *Let $\hat{A} = (A, \Delta A, \oplus_A)$ and $\hat{B} = (B, \Delta B, \oplus_B)$ be change actions.*

Then $\hat{A} + \hat{B} \coloneqq (A + B, \Delta A \times \Delta B, \oplus_+)$ is a change action, where $\oplus_+$ is defined as:

$$\iota_1 a \oplus_+ (\delta a, \delta b) \coloneqq \iota_1(a \oplus_A \delta a)$$
$$\iota_2 b \oplus_+ (\delta a, \delta b) \coloneqq \iota_2(b \oplus_B \delta b)$$

The injection maps ι_1, ι_2 are differentiable with respect to $\hat{A}+\hat{B}$. Furthermore, whenever $\hat{C}$ is a change action and $f : A \to C, g : B \to C$ are differentiable, then so is $[f, g]$.

2.4 Comparing Change Actions

Much like topological spaces, we can compare change actions on the same base set according to coarseness. This is useful since differentiability of functions between change actions is characterized entirely by the coarseness of the actions.

Definition 5. *Let $\hat{A}_1$ and $\hat{A}_2$ be change actions on A. We say that $\hat{A}_1$ is* coarser *than $\hat{A}_2$ (or that $\hat{A}_2$ is* finer *than $\hat{A}_1$) whenever for every $x \in A$ and change $\delta a_1 \in \Delta A_1$, there is a change $\delta a_2 \in \Delta A_2$ such that $x \oplus_{A_1} \delta a_1 = x \oplus_{A_2} \delta a_2$.*

We will write $\hat{A}_1 \leq \hat{A}_2$ whenever $\hat{A}_1$ is coarser than $\hat{A}_2$. If $\hat{A}_1$ is both finer and coarser than $\hat{A}_2$, we will say that $\hat{A}_1$ and $\hat{A}_2$ are equivalent.

The relation $\leq$ defines a preorder (but not a partial order) on the set of all change actions over a fixed set A. Least and greatest elements do exist up to equivalence, and correspond respectively to the empty change action $\hat{A}_\perp$ and any complete change action, such as the full change action $\hat{A}_\top$, defined in Sect. 2.1.

Proposition 5. *Let $\hat{A}_2 \leq \hat{A}_1$, $\hat{B}_1 \leq \hat{B}_2$ be change actions, and suppose the function $f : A \to B$ is differentiable as a function from $\hat{A}_1$ into $\hat{B}_1$. Then f is differentiable as a function from $\hat{A}_2$ into $\hat{B}_2$.*

A consequence of this fact is that whenever two change actions are equivalent they can be used interchangeably without affecting which functions are differentiable. One last parallel with topology is the following result, which establishes a simple criterion for when a change action is coarser than another:

Proposition 6. *Let $\hat{A}_1, \hat{A}_2$ be change actions on A. Then $\hat{A}_1$ is coarser than $\hat{A}_2$ if and only if the identity function* $\mathrm{id} : A \to A$ *is differentiable from $\hat{A}_1$ to $\hat{A}_2$.*

3 Posets and Boolean Algebras

The semantic domain of Datalog is a complete Boolean algebra, and so our next step is to construct a good change action for Boolean algebras. Along the way, we will consider change actions over posets, which give us the ability to *approximate* derivatives, which will turn out to be very important in practice.

3.1 Posets

Ordered sets give us a constrained class of functions: monotone functions. We can define *ordered* change actions, which are those that are well-behaved with respect to the order on the underlying set.[7]

Definition 6. *A change action $\hat{A}$ is* ordered *if*

- *A and ΔA are posets.*
- *$\oplus$ is monotone as a map from $A \times \Delta A \to A$*
- *$\cdot$ is monotone as a map from $\Delta A \times \Delta A \to \Delta A$*

In fact, any change action whose base set is a poset induces a partial order on the corresponding change set:

Definition 7. *$\delta a \leq_\Delta \delta b$ iff for all $a \in A$ it is the case that $a \oplus \delta a \leq a \oplus \delta b$.*

Proposition 7. *Let $\hat{A}$ be a change action on a set A equipped with a partial order $\leq$ such that $\oplus$ is monotone in its first argument. Then $\hat{A}$ is an ordered change action when ΔA is equipped with the partial order $\leq_\Delta$.*

In what follows, we will extend the partial order $\leq_\Delta$ on some change set ΔB pointwise to functions from some A into ΔB. This pointwise order interacts nicely with derivatives, in that it gives us the following lemma:

[7] If we were giving a presentation that was generic in the base category, then this would simply be the definition of being a change action in the category of posets and monotone maps.

Theorem 2 (Sandwich lemma). *Let $\hat{A}$ be a change action, and $\hat{B}$ be an ordered change action, and let $f : A \to B$ and $g : A \times \Delta A \to \Delta B$ be function. If $f_\uparrow$ and $f_\downarrow$ are derivatives for f such that*

$$f_\downarrow \leq_\Delta g \leq_\Delta f_\uparrow$$

then g is a derivative for f.

If unique minimal and maximal derivatives exist, then this gives us a characterisation of all the derivatives for a function.

Theorem 3. *Let $\hat{A}$ and $\hat{B}$ be change actions, with $\hat{B}$ ordered, and let $f : A \to B$ be a function. If there exist $f_{\downarrow\downarrow}$ and $f_{\uparrow\uparrow}$ which are unique minimal and maximal derivatives of f, respectively, then the derivatives of f are precisely the functions f' such that*

$$f_{\downarrow\downarrow} \leq_\Delta f' \leq_\Delta f_{\uparrow\uparrow}$$

This theorem gives us the leeway that we need when trying to pick a derivative: we can pick out the bounds, and that tells us how much "wiggle room" we have above and below.

3.2 Boolean Algebras

Complete Boolean algebras are a particularly nice domain for change actions because they have a negation operator. This is very helpful for computing differences, and indeed Boolean algebras have a complete change action.

Proposition 8 (Boolean algebra change actions). *Let L be a complete Boolean algebra. Define*

$$\hat{L}_{\bowtie} \coloneqq (L, L \bowtie L, \oplus_{\bowtie})$$

where

$$L \bowtie L \coloneqq \{(a, b) \in L \times L \mid a \wedge b = \bot\}$$
$$a \oplus_{\bowtie} (p, q) \coloneqq (a \vee p) \wedge \neg q$$

$$(p, q) \cdot (r, s) \coloneqq ((p \wedge \neg s) \vee r, (q \wedge \neg r) \vee s)$$

with identity element $(\bot, \bot)$.

Then $\hat{L}_{\bowtie}$ is a complete change action on L.

We can think of $\hat{L}_{\bowtie}$ as tracking changes as pairs of "upwards" and "downwards" changes, where the monoid action simply applies one after the other, with an adjustment to make sure that the components remain disjoint.[8] For example,

[8] The intuition that $\hat{L}_{\bowtie}$ is made up of an "upwards" and a "downwards" change action glued together can in fact be made precise, but the specifics are outside the scope of this paper.

in the powerset Boolean algebra $\mathcal{P}(\mathbb{N})$, a change to $\{1,2\}$ might consist of *adding* $\{3\}$ and *removing* $\{1\}$, producing $\{2,3\}$. In $\mathcal{P}(\mathbb{N})_{\bowtie}$ this would be represented as $(\{1,2\}) \oplus (\{3\},\{1\}) = \{2,3\}$.

Boolean algebras also have unique maximal and minimal derivatives, under the usual partial order based on implication. The change set is, as usual, given the change partial order, which in this case corresponds to the natural order on $L \times L^{\mathrm{op}}$.

Proposition 9. *Let L be a complete Boolean algebra with the $\hat{L}_{\bowtie}$ change action, and $f : A \to L$ be a function. Then, the following are minus operators:*

$$a \ominus_{\bot} b = (a \wedge \neg b, \neg a)$$
$$a \ominus_{\top} b = (a, b \wedge \neg a)$$

Additionally, $f'_{\ominus_{\bot}}$ and $f'_{\ominus_{\top}}$ define unique least and greatest derivatives for f.

Theorem 3 then gives us bounds for all the derivatives on Boolean algebras:

Corollary 1. *Let L be a complete Boolean algebra with the corresponding change action $\hat{L}_{\bowtie}$, $\hat{A}$ be an arbitrary change action, and $f : A \to L$ be a function. Then the derivatives of f are precisely those functions $f' : A \times \Delta A \to \Delta A$ such that*

$$f'_{\ominus_{\bot}} \leq_{\Delta} f' \leq_{\Delta} f'_{\ominus_{\top}}$$

This makes Theorem 3 actually usable in practice, since we have concrete definitions for our bounds (which we will make use of in Sect. 4.2).

4 Derivatives for Non-recursive Datalog

We now want to apply the theory we have developed to the specific case of the semantics of Datalog. Giving a differentiable semantics for Datalog will lead us to a strategy for performing incremental evaluation and maintenance of Datalog programs. To begin with, we will restrict ourselves to the non-recursive fragment of the language—the formulae that make up the right hand sides of Datalog rules. We will tackle the full program semantics in a later section, once we know how to handle fixpoints.

Although the techniques we are using should work for any language, Datalog provides a non-trivial case study where the need for incremental computation is real and pressing, as we saw in Sect. 1.

4.1 Semantics of Datalog Formulae

Datalog is usually given a logical semantics where formulae are interpreted as first-order logic predicates and the semantics of a program is the set of models of its constituent predicates. We will instead give a simple denotational semantics (as is typical when working with fixpoints, see e.g. [17]) that treats a Datalog formula as directly denoting a relation, i.e. a set of named tuples, with variables ranging over a finite schema.

Definition 8. *A* schema Γ *is a finite set of names. A* named tuple *over* Γ *is an assignment of a value* v_i *for each name* x_i *in* Γ*. Given disjoint schemata* $\Gamma = \{x_1, \ldots, x_n\}$ *and* $\Sigma = \{y_1, \ldots, y_m\}$*, the* selection function σ_Γ *is defined as*

$$\sigma_\Gamma(\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, y_1 \mapsto w_1, \ldots, y_m \mapsto w_m\}) \coloneqq \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$$

i.e. σ_Γ *restricts a named tuple over* $\Gamma \cup \Sigma$ *into a tuple over* Γ *with the same values for the names in* Γ*. We denote the elementwise extension of* σ_Γ *to sets of tuples also as* σ_Γ*.*

We will adopt the usual closed-world assumption to give a denotation to negation.

Definition 9. *For any schema* Γ*, there exists a universal relation* $\mathcal{U}_\Gamma$*. Negation on relations can then be defined as*

$$\neg R \coloneqq \mathcal{U}_\Gamma \setminus R$$

This makes $\mathbf{Rel}_\Gamma$, the set of all subsets of $\mathcal{U}_\Gamma$, a complete Boolean algebra.

Definition 10. *A Datalog formula* T *whose free term variables are contained in* Γ *denotes a function from* $\mathbf{Rel}_\Gamma^n$ *to* $\mathbf{Rel}_\Gamma$*.*

$$[\![\_]\!]_\Gamma : \text{Formula} \to \mathbf{Rel}_\Gamma^n \to \mathbf{Rel}_\Gamma$$

If $\mathcal{R} = (\mathcal{R}_1, \ldots, \mathcal{R}_n)$ *is a choice of a relation* $\mathcal{R}_i$ *for each of the variables* R_i*,* $[\![T]\!](\mathcal{R})$ *is inductively defined according to the rules in Fig. 1.*

$$\begin{array}{rr}
[\![\top]\!]_\Gamma(\mathcal{R}) \coloneqq \mathcal{U}_\Gamma & [\![T \wedge U]\!]_\Gamma(\mathcal{R}) \coloneqq [\![T]\!]_\Gamma(\mathcal{R}) \cap [\![U]\!]_\Gamma(\mathcal{R}) \\
[\![\bot]\!]_\Gamma(\mathcal{R}) \coloneqq \emptyset & [\![T \vee U]\!]_\Gamma(\mathcal{R}) \coloneqq [\![T]\!]_\Gamma(\mathcal{R}) \cup [\![U]\!]_\Gamma(\mathcal{R}) \\
[\![R_j]\!]_\Gamma(\mathcal{R}) \coloneqq \mathcal{R}_j & [\![\neg T]\!]_\Gamma(\mathcal{R}) \coloneqq \neg [\![T]\!]_\Gamma(\mathcal{R})
\end{array}$$

$$[\![\exists x.T]\!]_\Gamma(\mathcal{R}) \coloneqq \sigma_\Gamma([\![T]\!]_{\Gamma \cup \{x\}}(\mathcal{R}))$$

Fig. 1. Formula semantics for Datalog

Since $\mathbf{Rel}_\Gamma$ is a complete Boolean algebra, and so is $\mathbf{Rel}_\Gamma^n$, $[\![T]\!]_\Gamma$ is a function between complete Boolean algebras. For brevity, we will often leave the schema implicit, as it is clear from the context.

4.2 Differentiability of Datalog Formula Semantics

In order to actually perform our incremental computation, we first need to provide a concrete derivative for the semantics of Datalog formulae. Of course, since $[\![T]\!]_\Gamma$ is a function between the complete Boolean algebras $\mathbf{Rel}_\Gamma^n$ and $\mathbf{Rel}_\Gamma$, and

$$
\begin{aligned}
\Delta(\bot) &\coloneqq \bot & \nabla(\bot) &\coloneqq \bot \\
\Delta(\top) &\coloneqq \bot & \nabla(\top) &\coloneqq \bot \\
\Delta(R_j) &\coloneqq \Delta R_j & \nabla(R_j) &\coloneqq \nabla R_j \\
\Delta(T \vee U) &\coloneqq \Delta(T) \vee \Delta(U) & \nabla(T \vee U) &\coloneqq (\nabla(T) \wedge \neg \mathsf{X}(U)) \\
\Delta(T \wedge U) &\coloneqq (\Delta(T) \wedge \mathsf{X}(U)) & &\quad \vee (\nabla(U) \wedge \neg \mathsf{X}(T)) \\
&\quad \vee (\Delta(U) \wedge \mathsf{X}(T)) & \nabla(T \wedge U) &\coloneqq (\nabla(T) \wedge U) \vee (T \wedge \nabla(U)) \\
\Delta(\neg T) &\coloneqq \nabla(T) & \nabla(\neg T) &\coloneqq \Delta(T) \\
\Delta(\exists x.T) &\coloneqq \exists x.\Delta(T) & \nabla(\exists x.T) &\coloneqq \exists x.\nabla(T) \wedge \neg \exists x.\mathsf{X}(T)
\end{aligned}
$$

$$\mathsf{X}(R) \coloneqq (R \vee \Delta(R)) \wedge \neg \nabla(R)$$

Fig. 2. Upwards and downwards formula derivatives for Datalog

we know that the corresponding change actions $\widehat{\mathbf{Rel}^n_{\Gamma \bowtie}}$ and $\widehat{\mathbf{Rel}_{\Gamma \bowtie}}$ are complete, this guarantees the existence of a derivative for $[\![T]\!]$.

Unfortunately, this does not necessarily provide us with an *efficient* derivative for $[\![T]\!]$. The derivatives that we know how to compute (Corollary 1) rely on computing $f(a \oplus \delta a)$ itself, which is the very thing we were trying to avoid computing!

Of course, given a concrete definition of a derivative we can simplify this expression and hopefully make it easier to compute. But we also know from Corollary 1 that *any* function bounded by $f'_{\ominus \bot}$ and $f'_{\ominus \top}$ is a valid derivative, and we can therefore optimize anywhere within that range to make a trade-off between ease of computation and precision.[9]

There is also the question of how to compute the derivative. Since the change set for $\widehat{\mathbf{Rel}_{\bowtie}}$ is a subset of $\mathbf{Rel} \times \mathbf{Rel}$, it is possible and indeed very natural to compute the two components via a pair of Datalog formulae, which allows us to reuse an existing Datalog formula evaluator. Indeed, if this process is occurring in an optimizing compiler, the derivative formulae can themselves be optimized. This is very beneficial in practice, since the initial formulae may be quite complex.

This does give us additional constraints that the derivative formulae must satisfy: for example, we need to be able to evaluate them; and we may wish to pick formulae that will be easy or cheap for our evaluation engine to compute, even if they compute a less precise derivative.

The upshot of these considerations is that the optimal choice of derivatives is likely to be quite dependent on the precise variant of Datalog being evaluated, and the specifics of the evaluation engine. Here is one possibility, which is the one used at Semmle.

[9] The idea of using an approximation to the precise derivative, and a soundness condition, appears in Bancilhon [9].

A concrete Datalog formula derivative. In Fig. 2, we define a "symbolic" derivative operator as a pair of mutually recursive functions, Δ and ∇, which turn a Datalog formula T into new formulae that compute the upwards and downwards parts of the derivative, respectively. Our definition uses an auxiliary function, X, which computes the "neXt" value of a term by applying the upwards and downwards derivatives. As is typical for a derivative, the new formulae will have additional free relation variables for the upwards and downwards derivatives of the free relation variables of T, denoted as ΔR and ∇R respectively. Evaluating the formula as a derivative means evaluating it as a normal Datalog formula with the new relation variables set to the input relation changes.

While the definitions mostly exhibit the dualities we would expect between corresponding operators, there are a few asymmetries to explain.

The asymmetry between the cases for $\Delta(T \vee U)$ and $\nabla(T \wedge U)$ is for operational reasons. The symmetrical version of $\Delta(T \vee U)$ is $(\Delta(T) \wedge \neg U) \vee (\Delta(U) \wedge \neg T)$ (which is also precise). The reason we omit the negated conjuncts is simply that they are costly to compute and not especially helpful to our evaluation engine.

The asymmetry between the cases for $\exists$ is because our dialect of Datalog does not have a primitive universal quantifier. If we did have one, the cases for $\exists$ would be dual to the corresponding cases for $\forall$.

Theorem 4 (Concrete Datalog formula derivatives). *Let* Δ, ∇, X : Formula $\to$ Formula *be mutually recursive functions defined by structural induction as in Fig. 2.*

Then $\Delta(T)$ *and* $\nabla(T)$ *are disjoint, and for any schema* Γ *and any Datalog formula* T *whose free term variables are contained in* Γ, $[\![T]\!]'_\Gamma := ([\![\Delta(T)]\!]_\Gamma, [\![\nabla(T)]\!]_\Gamma)$ *is a derivative for* $[\![T]\!]_\Gamma$.

We can give a derivative for our *treeP* predicate by mechanically applying the recursive functions defined in Fig. 2.

$$\begin{aligned}&\Delta(treeP(x))\\ &= p(x) \wedge \exists y.(child(x,y) \wedge \Delta(treeP(y))) \wedge \neg\exists y.(child(x,y) \wedge \neg\mathsf{X}(treeP(y)))\end{aligned}$$

$$\begin{aligned}&\nabla(treeP(x))\\ &= p(x) \wedge \exists y.(child(x,y) \wedge \nabla(treeP(y)))\end{aligned}$$

The upwards difference in particular is not especially easy to compute. If we naively compute it, the third conjunct requires us to recompute the whole of the recursive part. However, the second conjunct gives us a guard: if it is empty we then the whole formula will be, so we only need to evaluate the third conjunct if the second conjunct is non-empty, i.e if there is *some* change in the body of the existential.

This shows that our derivatives aren't a panacea: it is simply *hard* to compute downwards differences for $\exists$ (and, equivalently, upwards differences for $\forall$) because we must check that there is no other way of deriving the same facts.[10] However,

[10] The "support" data structures introduced by [25] are an attempt to avoid this issue by tracking the number of derivations of each tuple.

we can still avoid the re-evaluation in many cases, and the inefficiency is local to this subformula.

4.3 Extensions to Datalog

Our formulation of Datalog formula semantics and derivatives is generic and modular, so it is easy to extend the language with new formula constructs: all we need to do is add cases for Δ and ∇.

In fact, because we are using a complete change action, we can *always* do this by using the maximal or minimal derivative. This justifies our claim that we can support *arbitrary* additional formula constructs: although the maximal and minimal derivatives are likely to be impractical, having them available as options means that we will never be completely stymied.

This is important in practice: here is a real example from Semmle's variant of Datalog. This includes a kind of aggregates which have well-defined recursive semantics. Aggregates have the form

$$r = \mathrm{agg}(p)(vs \mid T \mid U)$$

where agg refers to an aggregation function (such as "sum" or "min"), vs is a sequence of variables, p and r are variables, T is a formula possibly mentioning vs, and U is a formula possibly mentioning vs and p. The full details can been found in Moor and Baars [34], but for example this allows us to write

$$\begin{aligned} \mathit{height}(n, h) \leftarrow & \neg\exists c.(\mathit{child}(n, c)) \wedge h = 0 \\ & \vee \exists h'.(h' = \max(p)(c \mid \mathit{child}(n, c) \mid \mathit{height}(c, p)) \wedge h = h' + 1) \end{aligned}$$

which recursively computes the height of a node in a tree.

Here is an upwards derivative for an aggregate formula:

$$\Delta(r = \mathrm{agg}(p)(vs \mid T \mid U)) \coloneqq \exists vs.(T \wedge \Delta U) \wedge r = \mathrm{agg}(p)(vs \mid T \mid U)$$

While this isn't a precise derivative, it is still substantially cheaper than re-evaluating the whole subformula, as the first conjunct acts as a guard, allowing us to skip the second conjunct when U has not changed.

5 Changes on Functions

So far we have defined change actions for the kinds of things that typically make up *data*, but we would also like to have change actions on *functions*. This would allow us to define derivatives for higher-order languages (where functions are first-class); and for semantic operators like fixpoint operators $\mathbf{fix} : (A \to A) \to A$, which also operate on functions.

Function spaces, however, differ from products and disjoint unions in that there is no obvious "best" change action on $A \to B$. Therefore instead of trying to define a single choice of change action, we will instead pick out subsets of function spaces which have "well-behaved" change actions.

Definition 11 (Functional Change Action). *Given change actions $\hat{A}$ and $\hat{B}$ and a set $U \subseteq A \to B$, a change action $\hat{U} = (U, \Delta U, \oplus_U)$ is* functional *whenever the evaluation map* $\mathrm{ev} : U \times A \to B$ *is differentiable, that is to say, whenever there exists a function* $\mathrm{ev}' : (U \times A) \times (\Delta U \times \Delta A) \to \Delta B$ *such that:*

$$(f \oplus_U \delta f)(a \oplus_A \delta a) = f(a) \oplus_B \mathrm{ev}'((f, a), (\delta f, \delta a))$$

We will write $\hat{U} \subseteq \hat{A} \Rightarrow \hat{B}$ whenever $U \subseteq A \to B$ and $\hat{U}$ is functional.

There are two reasons why functional change actions are usually associated with a *subset* of $U \subseteq A \to B$. Firstly, it allows us to restrict ourselves to spaces of monotone or continuous functions. But more importantly, functional change actions are necessarily made up of differentiable functions, and thus a functional change action may not exist for the entire function space $A \to B$.

Proposition 10. *Let $\hat{U} \subseteq \hat{A} \Rightarrow \hat{B}$ be a functional change action. Then every $f \in U$ is differentiable, with a derivative f' given by:*

$$f'(x, \delta x) = \mathrm{ev}'((f, x), (\mathbf{0}, \delta x))$$

5.1 Pointwise Functional Change Actions

Even if we restrict ourselves to the differentiable functions between $\hat{A}$ and $\hat{B}$ it is hard to find a concrete functional change action for this set. Fortunately, in many important cases there is a simple change action on the set of differentiable functions.

Definition 12 (Pointwise functional change action). *Let $\hat{A}$ and $\hat{B}$ be change actions. The* pointwise functional change action *$\hat{A} \Rightarrow_{pt} \hat{B}$, when it is defined, is given by $(\hat{A} \to \hat{B}, A \to \Delta B, \oplus_\to)$, with the monoid structure $(A \to \Delta B, \cdot_\to, \mathbf{0}_\to)$ and the action $\oplus_\to$ defined by:*

$$\begin{aligned}(f \oplus_\to \delta f)(x) &\coloneqq f(x) \oplus_B \delta f(x)\\ (\delta f \cdot_\to \delta g)(x) &\coloneqq \delta f(x) \cdot_B \delta g(x)\\ \mathbf{0}_\to(x) &\coloneqq \mathbf{0}_B\end{aligned}$$

That is, a change is given pointwise, mapping each point in the domain to a change in the codomain.

The above definition is not always well-typed, since given $f : \hat{A} \to \hat{B}$ and $\delta f : A \to \Delta B$ there is no guarantee that $f \oplus_\to \delta f$ is differentiable. We present two sufficient criteria that guarantee this.

Theorem 5. *Let $\hat{A}$ and $\hat{B}$ be change actions, and suppose that $\hat{B}$ satisfies one of the following conditions:*

- *$\hat{B}$ is a complete change action.*
- *The change action $\widehat{\Delta B} \coloneqq (\Delta B, \Delta B, \cdot_B)$ is complete and $\oplus_B : B \times \Delta B \to B$ is differentiable.*

Then the pointwise functional change action $(\hat{A} \to \hat{B}, A \to \Delta B, \oplus_{\to})$ *is well defined.*[11]

As a direct consequence of this, it follows that whenever L is a Boolean algebra (and hence has a complete change action), the pointwise functional change action $\hat{A} \Rightarrow_{pt} \hat{L}_{\bowtie}$ is well-defined.

Pointwise functional change actions are functional in the sense of Definition 11. Moreover, the derivative of the evaluation map is quite easy to compute.

Proposition 11 (Derivatives of the evaluation map). *Let* $\hat{A}$ *and* $\hat{B}$ *be change actions such that the pointwise functional change action* $\hat{A} \Rightarrow_{pt} \hat{B}$ *is well defined, and let* $f : \hat{A} \to \hat{B}$, $a \in A$, $\delta a \in \Delta A$, $\delta f \in A \to \Delta B$.

Then the following are both derivatives of the evaluation map:

$$\mathrm{ev}'_1((f, a), (\delta f, \delta a)) \coloneqq f'(a, \delta a) \cdot \delta f(a \oplus \delta a)$$
$$\mathrm{ev}'_2((f, a), (\delta f, \delta a)) \coloneqq \delta f(a) \cdot (f \oplus \delta f)'(a, \delta a)$$

A functional change action merely tells us that a derivative of the evaluation map exists—a pointwise change action actually gives us a definition of it. In practice, this means that we will only be able to use the results in Sect. 6.2 (incremental computation and derivatives of fixpoints) when we have pointwise change actions, or where we have some other way of computing a derivative of the evaluation map.

6 Directed-Complete Partial Orders and Fixpoints

Directed-complete partial orders (dcpos) equipped with a least element, are an important class of posets. They allow us to take *fixpoints* of (Scott-)continuous maps, which is important for interpreting recursion in program semantics.

6.1 Dcpos

As before, we can define change actions on dcpos, rather than sets, as change actions whose base and change sets are endowed with a dcpo structure, and where the monoid operation and action are (Scott-)continuous.

Definition 13. *A change action* $\hat{A}$ *is* continuous *if*

- A *and* ΔA *are dcpos.*
- $\oplus$ *is Scott-continuous as a map from* $A \times \Delta A \to A$.
- $\cdot$ *is Scott-continuous as a map from* $\Delta A \times \Delta A \to \Delta A$.

[11] Either of these conditions is enough to guarantee that the pointwise functional change action is well defined, but it can be the case that $\hat{B}$ satisfies neither and yet pointwise change actions into $\hat{B}$ do exist. A precise account of when pointwise functional change actions exist is outside the scope of this paper.

Unlike posets, the change order $\leq_\Delta$ does *not*, in general, induce a dcpo on ΔA. As a counterexample, consider the change action $(\overline{\mathbb{N}}, \mathbb{N}, +)$, where $\overline{\mathbb{N}}$ denotes the dcpo of natural numbers extended with positive infinity.

A key example of a continuous change action is the $\hat{L}_{\bowtie}$ change action on Boolean algebras.

Proposition 12 (Boolean algebra continuity). *Let L be a Boolean algebra. Then $\hat{L}_{\bowtie}$ is a continuous change action.*

For a general overview of results in domain theory and dcpos, we refer the reader to an introductory work such as [2], but we state here some specific results that we shall be using, such as the following, whose proof can be found in [2, Lemma 3.2.6]:

Proposition 13. *A function $f : A \times B \to C$ is continuous iff it is continuous in each variable separately.*

It is a well-known result in standard calculus that the limit of an absolutely convergent sequence of differentiable functions $\{f_i\}$ is itself differentiable, and its derivative is equal to the limit of the derivatives of the f_i. A consequence of Proposition 13 is the following analogous result:

Corollary 2. *Let $\hat{A}$ and $\hat{B}$ be change actions, with $\hat{B}$ continuous and let $\{f_i\}$ and $\{f_i'\}$ be I-indexed directed sets of functions in $A \to B$ and $A \times \Delta A \to \Delta B$ respectively.*

Then, if for every $i \in I$ it is the case that f_i' is a derivative of f_i, then $\bigsqcup_{i \in I} f_i'$ is a derivative of $\bigsqcup_{i \in I} f_i$.

6.2 Fixpoints

Fixpoints appear frequently in the semantics of languages with recursion. If we can give a generic account of how to compute fixpoints using change actions, then this gives us a compositional way of extending a derivative for the non-recursive semantics of a language to a derivative that can also handle recursion. We will later apply this technique to create a derivative for the semantics of full recursive Datalog (Sect. 7.2).

Iteration functions. Over directed-complete partial orders we can define a least fixpoint operator **lfp** in terms of the iteration function **iter**:

$$
\begin{aligned}
&\mathbf{iter} : (A \to A) \times \mathbb{N} \to A \\
&\mathbf{iter}(f, 0) \coloneqq \bot \\
&\mathbf{iter}(f, n) \coloneqq f^n(\bot) \\
&\mathbf{lfp} : (A \to A) \to A \\
&\mathbf{lfp}(f) \coloneqq \bigsqcup_{n \in \mathbb{N}} \mathbf{iter}(f, n) \qquad \text{(where } f \text{ is continuous)}
\end{aligned}
$$

The iteration function is the basis for all the results in this section: we can take a partial derivative with respect to n, and this will give us a way to get to the next iteration incrementally; and we can take the partial derivative with respect to f, and this will give us a way to get from iterating f to iterating $f \oplus \delta f$.

Incremental computation of fixpoints. The following theorems provide a generalization of semi-naive evaluation to any differentiable function over a continuous change action. Throughout this section we will assume that we have a continuous change action $\hat{A}$, and any reference to the change action $\hat{\mathbb{N}}$ will refer to the monoidal change action on the naturals defined in Sect. 2.1.

Since we are trying to incrementalize the iterative step, we start by taking the partial derivative of **iter** with respect to n.

Proposition 14 (Derivative of the iteration map with respect to n). *Let $\hat{A}$ be a complete change action and let $f : A \to A$ be a differentiable function. Then* **iter** *is differentiable with respect to its second argument, and a partial derivative is given by:*

$$\begin{aligned}
&\partial_2\mathbf{iter} : (A \to A) \times \mathbb{N} \times \Delta\mathbb{N} \to \Delta A \\
&\partial_2\mathbf{iter}(f, \mathbf{0}, m) \coloneqq \mathbf{iter}(f, m) \ominus \mathbf{iter}(f, 0) \\
&\partial_2\mathbf{iter}(f, n+1, m) \coloneqq f'(\mathbf{iter}(f, n), \partial_2\mathbf{iter}(f, n, m))
\end{aligned}$$

By using the following recurrence relation, we can then compute $\partial_2\mathbf{iter}$ along with **iter** simultaneously:

$$\begin{aligned}
&\mathbf{recur}_f : A \times \Delta A \to A \times \Delta A \\
&\mathbf{recur}_f(\bot, \bot) \coloneqq (\bot, f(\bot) \ominus \bot) \\
&\mathbf{recur}_f(a, \delta a) \coloneqq (a \oplus \delta a, f'(a, \delta a))
\end{aligned}$$

Which has the property that

$$\mathbf{recur}_f^n(\bot, \bot) = (\mathbf{iter}(f, n), \partial_2\mathbf{iter}(f, n, 1))$$

This gives us a way to compute a fixpoint incrementally, by adding successive changes to an accumulator until we reach it. This is exactly how semi-naive evaluation works: you compute the delta relation and the accumulator simultaneously, adding the delta into the accumulator at each stage until it becomes the final output.

Theorem 6 (Incremental computation of least fixpoints). *Let $\hat{A}$ be a complete, continuous change action, $f : \hat{A} \to \hat{A}$ be continuous and differentiable. Then* $\mathbf{lfp}(f) = \bigsqcup_{n \in \mathbb{N}}(\pi_1(\mathbf{recur}_f^n(\bot, \bot)))$.[12]

[12] Note that we have *not* taken the fixpoint of $\mathbf{recur}_f$, since it is not continuous.

Derivatives of fixpoints. In the previous section we have shown how to use derivatives to compute fixpoints more efficiently, but we also want to take the derivative of the fixpoint operator itself. A typical use case for this is where we have calculated some fixpoint

$$F_E \coloneqq \mathbf{fix}(\lambda X.F(E, X))$$

then update the parameter E with some change δE and wish to compute the new value of the fixpoint, i.e.

$$F_{E \oplus \delta E} \coloneqq \mathbf{fix}(\lambda X.F(E \oplus \delta E, X))$$

This can be seen as applying a change to the *function* whose fixpoint we are taking. We go from computing the fixpoint of $F(E, \_)$ to computing the fixpoint of $F(E \oplus \delta E, \_)$. If we have a pointwise functional change action then we can express this change as a function giving the change at each point, that is:

$$\lambda X.F(E \oplus \delta E, X) \ominus F(E, X)$$

In Datalog this would allow us to update a recursively defined relation given an update to one of its non-recursive dependencies, or the extensional database. For example, we might want to take the transitive closure relation and update it by changing the edge relation e.

However, to compute these examples would requires us to provide a derivative for the fixpoint operator **fix**: we want to know how the resulting fixpoint changes given a change to its input function.

Definition 14 (Derivatives of fixpoints). *Let $\hat{A}$ be a change action, let $\hat{U} \subseteq \hat{A} \Rightarrow \hat{A}$ be a functional change action (not necessarily pointwise) and suppose $\mathbf{fix}_U$ and $\mathbf{fix}_{\Delta A}$ are fixpoint operators for endofunctions on U and ΔA respectively.*

Then we define

$$\begin{aligned}
&\mathbf{adjust} : U \times \Delta U \to (\Delta A \to \Delta A)\\
&\mathbf{adjust}(f, \delta f) \coloneqq \lambda\ \delta a.\, \mathrm{ev}'((f, \mathbf{fix}_U(f)), (\delta f, \delta a))\\
&\mathbf{fix}'_U : U \times \Delta U \to \Delta A\\
&\mathbf{fix}'_U(f, \delta f) \coloneqq \mathbf{fix}_{\Delta A}(\mathbf{adjust}(f, \delta f))
\end{aligned}$$

The suggestively named $\mathbf{fix}'_U$ will in fact turn out to be a derivative—for *least* fixpoints. The appearance of ev', a derivative of the evaluation map, in the definition of **adjust** is also no coincidence: as evaluating a fixpoint consists of many steps of applying the evaluation map, so computing the derivative of a fixpoint consists of many steps of applying the derivative of the evaluation map.[13]

[13] Perhaps surprisingly, the authors first discovered an expanded version of this formula, and it was only later that we realised the remarkable connection to ev'.

Since **lfp** is characterized as the limit of a chain of functions, Corollary 2 suggests a way to compute its derivative. It suffices to find a derivative $\mathbf{iter}'_n$ of each iteration map such that the resulting set $\{\mathbf{iter}'_n \mid n \in \mathbb{N}\}$ is directed, which will entail that $\bigsqcup_{n\in\mathbb{N}} \mathbf{iter}'_n$ is a derivative of **lfp**.

These correspond to the first partial derivative of **iter**—this time with respect to f. While we are differentiating with respect to f, we are still going to need to define our derivatives inductively in terms of n.

Proposition 15 (Derivative of the iteration map with respect to f). **iter** *is differentiable with respect to its first argument and a derivative is given by:*

$$\begin{aligned}
&\partial_1\mathbf{iter} : (A \to A) \times \Delta(A \to A) \times \mathbb{N} \to \Delta A\\
&\partial_1\mathbf{iter}(f, \delta f, \mathbf{0}) \coloneqq \bot_{\Delta A}\\
&\partial_1\mathbf{iter}(f, \delta f, n+1) \coloneqq \mathrm{ev}'((f, \mathbf{iter}(f, n)), (\delta f, \partial_1\mathbf{iter}(f, \delta f, n)))
\end{aligned}$$

As before, we can now compute $\partial_1\mathbf{iter}$ together with **iter** by mutual recursion.[14]

$$\begin{aligned}
&\mathbf{recur}_{f,\delta f} : A \times \Delta A \to A \times \Delta A\\
&\mathbf{recur}_{f,\delta f}(a, \delta a) \coloneqq (f(a), \mathrm{ev}'((f, a), (\delta f, \delta a)))
\end{aligned}$$

Which has the property that

$$\mathbf{recur}^n_{f,\delta f}(\bot, \bot) = (\mathbf{iter}(f, n), \partial_1\mathbf{iter}(f, \delta f, n)).$$

This indeed provides us with a function whose limit we can take. If we do so we will discover that it is exactly $\mathbf{lfp}'$ (defined as in Definition 14), showing that $\mathbf{lfp}'$ is a true derivative.

Theorem 7 (Derivatives of least fixpoint operators). *Let*

- $\hat{A}$ *be a continuous change action*
- U *be the set of continuous functions* $f : A \to A$*, with a functional change action* $\hat{U} \subseteq \hat{A} \Rightarrow \hat{A}$
- $f \in U$ *be a continuous, differentiable function*
- $\delta f \in \Delta U$ *be a function change*
- ev' *be a derivative of the evaluation map which is continuous with respect to* a *and* δa*.*

Then $\mathbf{lfp}'$ *is a derivative of* **lfp**.

Computing this derivative still requires computing a fixpoint—over the change lattice—but this may still be significantly less expensive than recomputing the full new fixpoint.

[14] In fact, the recursion here is not *mutual*: the first component does not depend on the second. However, writing it in this way makes it amenable to computation by fixpoint, and we will in fact be able to avoid the recomputation of $\mathbf{iter}_n$ when we show that it is equivalent to $\mathbf{lfp}'$.

7 Derivatives for Recursive Datalog

Given the non-recursive semantics for a language, we can extend it to handle recursive definitions using fixpoints. Section 6.2 lets us extend our derivative for the non-recursive semantics to a derivative for the recursive semantics, as well as letting us compute the fixpoints themselves incrementally.

Again, we will demonstrate the technique with Datalog, although the approach is generic.

7.1 Semantics of Datalog Programs

First of all, we define the usual "immediate consequence operator" which computes "one step" of our program semantics.

Definition 15. *Given a program $\mathbb{P} = (P_1, \ldots, P_n)$, where P_i is a predicate, with schema Γ_i, the* immediate consequence operator $\mathcal{I} : \mathbf{Rel}^n \to \mathbf{Rel}^n$ *is defined as follows:*

$$\mathcal{I}(\mathcal{R}_1, \ldots, \mathcal{R}_n) = (\llbracket P_1 \rrbracket_{\Gamma_1}(\mathcal{R}_1, \ldots, \mathcal{R}_n), \ldots, \llbracket P_n \rrbracket_{\Gamma_n}(\mathcal{R}_1, \ldots, \mathcal{R}_n))$$

That is, given a value for the program, we pass in all the relations to the denotation of each predicate, to get a new tuple of relations.

Definition 16. *The semantics of a program $\mathbb{P}$ is defined to be*

$$\llbracket \mathbb{P} \rrbracket \coloneqq \mathbf{lfp}_{\mathbf{Rel}^n}(\mathcal{I})$$

and may be calculated by iterative application of $\mathcal{I}$ to $\bot$ until fixpoint is reached.

Whether or not this program semantics exists will depend on whether the fixpoint exists. Typically this is ensured by constraining the program such that $\mathcal{I}$ is monotone (or, in the context of a dcpo, continuous). We do not require monotonicity to apply Theorem 6 (and hence we can incrementally compute fixpoints that happen to exist even though the generating function is not monotonic), but it is required to apply Theorem 7.

7.2 Incremental Evaluation of Datalog

We can easily extend a derivative for the formula semantics to a derivative for the immediate consequence operator $\mathcal{I}$. Putting this together with the results from Sect. 6.2, we have now created *modular* proofs for the two main results, which allows us to preserve them in the face of changes to the underlying language.

Corollary 3. *Datalog program semantics can be evaluated incrementally.*

Corollary 4. *Datalog program semantics can be incrementally maintained with changes to relations.*

Note that our approach makes no particular distinction between changes to the *extensional* relations (adding or removing facts), and changes to the *intensional* relations (changing the definition). The latter simply amounts to a change to the denotation of that relation, which can be incrementally propagated in exactly the same way as we would propagate a change to the extensional relations.

8 Related Work

8.1 Change Actions and Incremental Computation

Change structures. The seminal paper in this area is Cai et al. [14]. We deviate from that excellent paper in three regards: the inclusion of minus operators, the nature of function changes, and the use of dependent types.

We have omitted minus operators from our definition because there are many interesting change actions that are not complete and so cannot have a minus operator. Where we can find a change structure with a minus operator, often we are forced to use unwieldy representations for change sets, and Cai et al. cite this as their reason for using a dependent type of changes. For example, the monoidal change actions on sets and lists are clearly useful for incremental computation on streams, yet they do not admit minus operators—instead, one would be forced to work with e.g. multisets admitting negative arities, as Cai et al. do.

Our function changes (when well behaved) correspond to what Cai et al. call *pointwise differences* (see [14, section 2.2]). As they point out, you can reconstruct their function changes from pointwise changes and derivatives, so the two formulations are equivalent.

The equivalence of our presentations means that our work should be compatible with their Incremental Lambda Calculus (see [14, section 3]). The derivatives we give in Sect. 4.2 are more or less a "change semantics" for Datalog (see [14, section 3.5]).

S-acts. S-acts (i.e the category of monoid actions on sets) and their categorical structure have received a fair amount of attention over the years (Kilp, Knauer, and Mikhalev [30] is a good overview). However, there is a key difference between change actions considered as a category (**CAct**) and the category of S-acts (**SAct**): the objects of **SAct** all maintain the same monoid structure, whereas we are interested in changing both the base set *and* the structure of the action.

Derivatives of fixpoints. Arntzenius [5] gives a derivative operator for fixpoints based on the framework in Cai et al. [14]. However, since we have different notions of function changes, the result is inapplicable as stated. In addition, we require a somewhat different set of conditions; in particular, we do not require our changes to always be increasing.

8.2 Datalog

Incremental evaluation. The earliest interpretation of semi-naive evaluation as a derivative appears in Bancilhon [8]. The idea of using an approximate derivative and the requisite soundness condition appears as a throwaway comment in Bancilhon and Ramakrishnan [9, section 3.2.2], and it would appear that nobody has since developed that approach.

As far as we know, traditional semi-naive is the state of the art in incremental, bottom-up, Datalog evaluation, and there are no strategies that accommodate additional language features such as parity-stratified negation and aggregates.

Incremental maintenance. There is existing literature on incremental maintenance of relational algebra expressions.

Griffin, Libkin, and Trickey [24] following Qian and Wiederhold [35] compute differences with both an "upwards" and a "downwards" component, and produce a set of rules that look quite similar to those we derive in Theorem 4. However, our presentation is significantly more generic, handles recursive expressions, and works on set semantics rather than bag semantics.[15]

Several approaches [25,27]—most notably DReD—remove facts until one can start applying the rules again to reach the new fixpoint. Given a good way of deciding what facts to remove this can be quite efficient. However, such techniques tend to be tightly coupled to the domain. Although we know of no theoretical reason why either approach should give superior performance when both are applicable, an empirical investigation of this could prove interesting.

Other approaches [19,43] consider only restricted subsets of Datalog, or incur other substantial constraints.

Embedding Datalog. Datafun (Arntzenius and Krishnaswami [6]) is a functional programming language that embeds Datalog, allowing significant improvements in genericity, such as the use of higher-order functions. Since we have directly defined a change action and derivative operator for Datalog, our work could be used as a "plugin" in the sense of Cai et al., allowing Datafun to compute its internal fixpoints incrementally, but also allowing Datafun expressions to be fully incrementally maintained.

In a different direction, Cathcart Burn, Ong, and Ramsay [15] have proposed *higher-order constrained Horn clauses* (HoCHC), a new class of constraints for the automatic verification of higher-order programs. HoCHC may be viewed as a higher-order extension of Datalog. Change actions can be readily applied to organise an efficient semi-naive method for solving HoCHC systems.

8.3 Differential λ-calculus

Another setting where derivatives of arbitrary higher-order programs have been studied is the *differential λ-calculus* [20,21]. This is a higher-order, simply-typed

[15] The same approach of finding derivatives would work with bag semantics, although unfortunately the Boolean algebra structure is missing.

λ-calculus which allows for computing the derivative of a function, in a similar way to the notion of derivative in Cai's work and the present paper.

While there are clear similarities between the two systems, the most important difference is the properties of the derivatives themselves: in the differential λ-calculus, derivatives are guaranteed to be linear in their second argument, whereas in our approach derivatives do not have this restriction but are instead required to satisfy a strong relation to the function that is being differentiated (see Definition 2).

Families of denotational models for the differential λ-calculus have been studied in depth [12,13,16,29], and the relationship between these and change actions is the subject of ongoing work.

8.4 Higher-Order Automatic Differentiation

Automatic differentiation [23] is a technique that allows for efficiently computing the derivative of arbitrary programs, with applications in probabilistic modeling [31] and machine learning [10] among other areas. In recent times, this technique has been successfully applied to higher-order languages [11,41]. While some approaches have been suggested [28,33], a general theoretical framework for this technique is still a matter of open research.

To this purpose, some authors have proposed the incremental λ-calculus as a foundational framework on which models of automatic differentiation can be based [28]. We believe our change actions are better suited to this purpose than the incremental λ-calculus, since one can easily give them a synthetic differential geometric reading (by interpreting $\hat{A}$ as an Euclidean module and ΔA as its corresponding spectrum, for example).

9 Conclusions and Future Work

We have presented change actions and their properties, and used them to provide novel, compositional, strategies for incrementally evaluating and maintaining recursive functions, in particular the semantics of Datalog.

The main avenue for future theoretical work is the categorical structure of change actions. This has begun to be explored by the authors in [4], where change actions are generalized to arbitrary Cartesian base categories and a construction is provided to obtain "canonical" Cartesian closed categories of change actions and differentiable maps.

We hope that these generalizations would allow us to extend the theory of change actions towards other classes of models, such as synthetic differential geometry and domain theory. Some early results in [4] also indicate a connection between 2-categories and change actions which has yet to be fully mapped.

The compositional nature of these techniques suggest that an approach like that used in [22] could be used for an even more generic approach to automatic differentiation.

In addition, there is plenty of scope for practical application of the techniques given here to languages other than Datalog.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases: The Logical Level. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
2. Abramsky, S., Jung, A.: Domain theory. In: Handbook of Logic in Computer Science. Oxford University Press, New York (1994)
3. Alvarez-Picallo, M., Eyers-Taylor, A., Jones, M.P., Ong, C.L.: Fixing incremental computation: derivatives of fixpoints, and the recursive semantics of datalog. CoRR abs/1811.06069 (2018). http://arxiv.org/abs/1811.06069
4. Alvarez-Picallo, M., Ong, C.H.L.: Change actions: models of generalised differentiation. In: International Conference on Foundations of Software Science and Computation Structures. Springer (2019, in press)
5. Arntzenius, M.: Static differentiation of monotone fixpoints (2017). http://www.rntz.net/files/fixderiv.pdf
6. Arntzenius, M., Krishnaswami, N.R.: Datafun: a functional datalog. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, pp. 214–227. ACM (2016)
7. Avgustinov, P., de Moor, O., Jones, M.P., Schäfer, M.: QL: object-oriented queries on relational data. In: LIPIcs-Leibniz International Proceedings in Informatics, vol. 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
8. Bancilhon, F.: Naive evaluation of recursively defined relations. In: Brodie, M.L., Mylopoulos, J. (eds.) On Knowledge Base Management Systems. TINF, pp. 165–178. Springer, New York (1986). https://doi.org/10.1007/978-1-4612-4980-1_17
9. Bancilhon, F., Ramakrishnan, R.: An amateur's introduction to recursive query processing strategies, vol. 15. ACM (1986)
10. Baydin, A.G., Pearlmutter, B.A.: Automatic differentiation of algorithms for machine learning. arXiv preprint arXiv:1404.7456 (2014)
11. Baydin, A.G., Pearlmutter, B.A., Siskind, J.M.: DiffSharp: an AD library for .NET languages. arXiv preprint arXiv:1611.03423 (2016)
12. Blute, R., Ehrhard, T., Tasson, C.: A convenient differential category. arXiv preprint arXiv:1006.3140 (2010)
13. Bucciarelli, A., Ehrhard, T., Manzonetto, G.: Categorical models for simply typed resource calculi. Electron. Notes Theor. Comput. Sci. **265**, 213–230 (2010)
14. Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K.: A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In: ACM SIGPLAN Notices, vol. 49, pp. 145–155. ACM (2014)
15. Cathcart Burn, T., Ong, C.L., Ramsay, S.J.: Higher-order constrained horn clauses for verification. PACMPL **2**(POPL), 11:1–11:28 (2018). https://doi.org/10.1145/3158099
16. Cockett, J.R.B., Gallagher, J.: Categorical models of the differential λ-calculus revisited. Electron. Notes Theor. Comput. Sci. **325**, 63–83 (2016)
17. Compton, K.J.: Stratified least fixpoint logic. Theor. Comput. Sci. **131**(1), 95–120 (1994)
18. Datomic website (2018). https://www.datomic.com. Accessed 01 Jan 2018
19. Dong, G., Su, J.: Incremental maintenance of recursive views using relational calculus/SQL. ACM SIGMOD Rec. **29**(1), 44–51 (2000)
20. Ehrhard, T.: An introduction to differential linear logic: proof-nets, models and antiderivatives. Math. Struct. Comput. Sci. 1–66 (2017)
21. Ehrhard, T., Regnier, L.: The differential lambda-calculus. Theor. Comput. Sci. **309**(1–3), 1–41 (2003)

22. Elliott, C.: The simple essence of automatic differentiation. Proc. ACM Program. Lang. **2**(ICFP), 70 (2018)
23. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, vol. 105. SIAM, Philadelphia (2008)
24. Griffin, T., Libkin, L., Trickey, H.: An improved algorithm for the incremental recomputation of active relational expressions. IEEE Trans. Knowl. Data Eng. **3**, 508–511 (1997)
25. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. ACM SIGMOD Rec. **22**(2), 157–166 (1993)
26. Halpin, T., Rugaber, S.: LogiQL: A Query Language for Smart Databases. CRC Press, Boca Raton (2014)
27. Harrison, J.V., Dietrich, S.W.: Maintenance of materialized views in a deductive database: an update propagation approach. In: Workshop on Deductive Databases, JICSLP, pp. 56–65 (1992)
28. Kelly, R., Pearlmutter, B.A., Siskind, J.M.: Evolving the incremental λ calculus into a model of forward automatic differentiation (AD). arXiv preprint arXiv:1611.03429 (2016)
29. Kerjean, M., Tasson, C.: Mackey-complete spaces and power series-a topological model of differential linear logic. Math. Struct. Comput. Sci. 1–36 (2016)
30. Kilp, M., Knauer, U., Mikhalev, A.V.: Monoids, Acts and Categories: With Applications to Wreath Products and Graphs. A Handbook for Students and Researchers, vol. 29. Walter de Gruyter, Berlin (2000)
31. Kucukelbir, A., Tran, D., Ranganath, R., Gelman, A., Blei, D.M.: Automatic differentiation variational inference. J. Mach. Learn. Res. **18**(1), 430–474 (2017)
32. LogicBlox Inc. website (2018). http://www.logicblox.com. Accessed 01 Jan 2018
33. Manzyuk, O.: A simply typed λ-calculus of forward automatic differentiation. Electron. Notes Theor. Comput. Sci. **286**, 257–272 (2012)
34. de Moor, O., Baars, A.: Doing a doaitse: simple recursive aggregates in datalog. In: Liber Amicorum for Doaitse Swierstra, pp. 207–216 (2013). http://www.staff.science.uu.nl/~hage0101/liberdoaitseswierstra.pdf. Accessed 01 Jan 2018
35. Qian, X., Wiederhold, G.: Incremental recomputation of active relational expressions. IEEE Trans. Knowl. Data Eng. **3**(3), 337–341 (1991)
36. Sáenz-Pérez, F.: DES: a deductive database system. Electron. Notes Theor. Comput. Sci. **271**, 63–78 (2011)
37. Schäfer, M., de Moor, O.: Type inference for datalog with complex type hierarchies. In: ACM SIGPLAN Notices, vol. 45, pp. 145–156. ACM (2010)
38. Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On fast large-scale program analysis in datalog. In: Proceedings of the 25th International Conference on Compiler Construction, pp. 196–206. ACM (2016)
39. Semmle Ltd. website (2018). https://semmle.com. Accessed 01 Jan 2018
40. Sereni, D., Avgustinov, P., de Moor, O.: Adding magic to an optimising datalog compiler. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 553–566. ACM (2008)
41. Siskind, J.M., Pearlmutter, B.A.: Nesting forward-mode AD in a functional framework. High.-Order Symb. Comput. **21**(4), 361–376 (2008)
42. Souffle language website (2018). http://souffle-lang.org. Accessed 01 Jan 2018
43. Urpi, T., Olive, A.: A method for change computation in deductive databases. In: VLDB, vol. 92, pp. 225–237 (1992)

Incremental λ-Calculus in Cache-Transfer Style

Static Memoization by Program Transformation

Paolo G. Giarrusso[1(✉)], Yann Régis-Gianas[2], and Philipp Schuster[3]

[1] LAMP—EPFL, Lausanne, Switzerland
[2] IRIF, University of Paris Diderot, Inria, Paris, France
[3] University of Tübingen, Tübingen, Germany

Abstract. Incremental computation requires propagating changes and reusing intermediate results of base computations. Derivatives, as produced by static differentiation [7], propagate changes but do not reuse intermediate results, leading to wasteful recomputation. As a solution, we introduce conversion to *Cache-Transfer-Style*, an additional program transformations producing purely incremental functional programs that create and maintain nested tuples of intermediate results. To prove CTS conversion correct, we extend the correctness proof of static differentiation from STLC to untyped λ-calculus via *step-indexed logical relations*, and prove sound the additional transformation via simulation theorems.

To show ILC-based languages can improve performance relative to from-scratch recomputation, and that CTS conversion can extend its applicability, we perform an initial performance case study. We provide derivatives of primitives for operations on collections and incrementalize selected example programs using those primitives, confirming expected asymptotic speedups.

1 Introduction

After computing a base output from some base input, we often need to produce updated outputs corresponding to updated inputs. Instead of rerunning the same *base program* on the updated input, incremental computation transforms the input change to an output change, potentially reducing asymptotic time complexity and significantly improving efficiency, especially for computations running on large data sets.

Incremental λ-Calculus (ILC) [7] is a recent framework for *higher-order* incremental computation. ILC represents changes from a base value v_1 to an updated value v_2 as a first-class *change value* dv. Since functions are first-class values, change values include *function changes*.

ILC also statically transforms *base programs* to *incremental programs* or *derivatives*, that are functions mapping input changes to output changes. Incremental language designers can then provide their language with (higher-order) primitives (with their derivatives) that efficiently encapsulate incrementalizable

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 553–580, 2019.
https://doi.org/10.1007/978-3-030-17184-1_20

computation skeletons (such as tree-shaped folds), and ILC will incrementalize higher-order programs written in terms of these primitives.

Alas, ILC only incrementalizes efficiently *self-maintainable computations* [7, Sect. 4.3], that is, computations whose output changes can be computed using only input changes, but not the inputs themselves [11]. Few computations are self-maintainable: for instance, mapping self-maintainable functions on a sequence is self-maintainable, but dividing numbers is not! We elaborate on this problem in Sect. 2.1. In this paper, we extend ILC to non-self-maintainable computations. To this end, we must enable derivatives to reuse intermediate results created by the base computation.

Many incrementalization approaches remember intermediate results through dynamic memoization: they typically use hashtables to memoize function results, or dynamic dependence graphs [1] to remember a computation trace. However, looking up intermediate results in such dynamic data structure has a runtime cost that is hard to optimize; and reasoning on dynamic dependence graphs and computation traces is often complex. Instead, ILC produces purely functional programs, suitable for further optimizations and equational reasoning.

To that end, we replace dynamic memoization with *static memoization*: following Liu and Teitelbaum [20], we transform programs to *cache-transfer style (CTS)*. A CTS function outputs their primary result along with *caches* of intermediate results. These caches are just nested tuples whose structure is derived from code, and accessing them does not involve looking up keys depending on inputs. Instead, intermediate results can be fetched from these tuples using statically known locations. To integrate CTS with ILC, we extend differentiation to produce *CTS derivatives*: these can extract from caches any intermediate results they need, and produce updated caches for the next computation step.

The correctness proof of static differentiation in CTS is challenging. First, we must show a forward simulation relation between two triples of reduction traces (the first triple being made of the source base evaluation, the source updated evaluation and the source derivative evaluation; the second triple being made of the corresponding CTS-translated evaluations). Dealing with six distinct evaluation environments at the same time was error prone on paper and for this reason, we conducted the proof using Coq [26]. Second, the simulation relation must not only track values but also caches, which are only partially updated while in the middle of the evaluation of derivatives. Finally, we study the translation for an untyped λ-calculus, while previous ILC correctness proofs were restricted to simply-typed λ-calculus. Hence, we define which changes are valid via a *logical relation* and show its *fundamental property*. Being in an untyped setting, our logical relation is not indexed by types, but *step-indexed*. We study an untyped language, but our work also applies to the erasure of typed languages. Formalizing a type-preserving translation is left for future work because giving a type to CTS programs is challenging, as we shall explain.

In addition to the correctness proof, we present preliminary experimental results from three case studies. We obtain efficient incremental programs even on non self-maintainable functions.

We present our contributions as follows. First, we summarize ILC and illustrate the need to extend it to remember intermediate results via CTS (Sect. 2). Second, in our mechanized formalization (Sect. 3), we give a novel proof of correctness for ILC differentiation for untyped λ-calculus, based on step-indexed logical relations (Sect. 3.4). Third, building on top of ILC differentiation, we show how to transform untyped higher-order programs to CTS (Sect. 3.5) and we show that CTS functions and derivatives *simulate* correctly their non-CTS counterparts (Sect. 3.7). Finally, in our case studies (Sect. 4), we compare the performance of the generated code to the base programs. Section 4.4 discusses limitations and future work. Section 5 discusses related work and Sect. 6 concludes. Our mechanized proof in Coq, the case study material, and the extended version of this paper with appendixes are available online at https://github.com/yurug/cts.

2 ILC and CTS Primer

In this section we exemplify ILC by applying it on an average function, show why the resulting incremental program is asymptotically inefficient, and use CTS conversion and differentiation to incrementalize our example efficiently and speed it up asymptotically (as confirmed by benchmarks in Sect. 4.1). Further examples in Sect. 4 apply CTS to higher-order programs and suggest that CTS enables incrementalizing efficiently some core database primitives such as joins.

2.1 Incrementalizing *average* via ILC

Our example computes the average of a bag of numbers. After computing the *base output* y_1 of the average function on the *base input* bag xs_1, we want to update the output in response to a stream of updates to the input bag. Here and throughout the paper, we contrast *base* vs *updated* inputs, outputs, values, computations, and so on. For simplicity, we assume we have two *updated inputs* xs_2 and xs_3 and want to compute two *updated outputs* y_2 and y_3. We express this program in Haskell as follows:

$$
\begin{aligned}
&\mathit{average} &&:: \mathit{Bag}\ \mathbb{Z} \to \mathbb{Z}\\
&\mathit{average}\ \mathit{xs} &&= \mathbf{let}\ s = \mathit{sum}\ \mathit{xs}; n = \mathit{length}\ \mathit{xs}; r = \mathit{div}\ s\ n\ \mathbf{in}\ r\\
&\mathit{average}_3 &&= \mathbf{let}\ y_1 = \mathit{average}\ \mathit{xs}_1; y_2 = \mathit{average}\ \mathit{xs}_2; y_3 = \mathit{average}\ \mathit{xs}_3\\
&&&\quad \mathbf{in}\ (y_1, y_2, y_3)
\end{aligned}
$$

To compute the updated outputs y_2 and y_3 in $\mathit{average}_3$ faster, we try using ILC. For that, we assume that we receive not only updated inputs xs_2 and xs_3 but also *input change* dxs_1 from xs_1 to xs_2 and input change dxs_2 from xs_2 to xs_3. A change dx from x_1 to x_2 describes the changes from base value x_1 to updated value x_2, so that x_2 can be computed via the *update operator* $\oplus$ as $x_1 \oplus dx$. A nil change $\mathbf{0}_x$ is a change from base value x to updated value x itself.

ILC differentiation automatically transforms the *average* function to its derivative $daverage :: Bag\ \mathbb{Z} \to \Delta(Bag\ \mathbb{Z}) \to \Delta\mathbb{Z}$. A derivative maps input changes to output changes: here, $dy_1 = daverage\ xs_1\ dxs_1$ is a change from base output $y_1 = average\ xs_1$ to updated output $y_2 = average\ xs_2$, hence $y_2 = y_1 \oplus dy_1$.

Thanks to *daverage*'s correctness, we can rewrite $average_3$ to avoid expensive calls to *average* on updated inputs and use *daverage* instead:

$$
\begin{aligned}
&incrementalAverage_3 :: (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})\\
&incrementalAverage_3 =\\
&\quad \textbf{let}\ y_1 = average\ xs_1; dy_1 = daverage\ xs_1\ dxs_1\\
&\qquad\ \ y_2 = y_1 \oplus dy_1; dy_2 = daverage\ xs_2\ dxs_2\\
&\qquad\ \ y_3 = y_2 \oplus dy_2\\
&\quad \textbf{in}\ \ (y_1, y_2, y_3)
\end{aligned}
$$

In general, also the value of a function $f :: A \to B$ can change from a base value f_1 to an updated value f_2, mainly when f is a closure over changing data. In that case, the change from base output $f_1\ x_1$ to updated output $f_2\ x_2$ is given by $df\ x_1\ dx$, where $df :: A \to \Delta A \to \Delta B$ is now a *function change* from f_1 to f_2. Above, *average* exemplifies the special case where $f_1 = f_2 = f$: then the function change df is a nil change, and $df\ x_1\ dx$ is a change from $f_1\ x_1 = f\ x_1$ and $f_2\ x_2 = f\ x_2$. That is, a nil function change for f is a derivative of f.

2.2 Self-maintainability and Efficiency of Derivatives

Alas, derivatives are efficient only if they are *self-maintainable*, and *daverage* is not, so $incrementalAverage_3$ is no faster than $average_3$! Consider the result of differentiating *average*:

$$
\begin{aligned}
&daverage :: Bag\ \mathbb{Z} \to \Delta(Bag\ \mathbb{Z}) \to \Delta\mathbb{Z}\\
&daverage\ xs\ dxs = \textbf{let}\ s = sum\ xs; ds = dsum\ xs\ dxs;\\
&\qquad\qquad\qquad\qquad\ \ n = length\ xs; dn = dlength\ xs\ dxs;\\
&\qquad\qquad\qquad\qquad\ \ r = div\ s\ n; dr = ddiv\ s\ ds\ n\ dn\\
&\qquad\qquad\qquad\ \ \textbf{in}\ dr
\end{aligned}
$$

Just like *average* combines *sum*, *length*, and *div*, its derivative *daverage* combines those functions and their derivatives. *daverage* recomputes base intermediate results s, n and r exactly as done in *average*, because they might be needed as base inputs of derivatives. Since r is unused, its recomputation can be dropped during later optimizations, but expensive intermediate results s and n are used by *ddiv*:

$$
\begin{aligned}
&ddiv :: \mathbb{Z} \to \Delta\mathbb{Z} \to \mathbb{Z} \to \Delta\mathbb{Z} \to \Delta\mathbb{Z}\\
&ddiv\ a\ da\ b\ db = div\ (a \oplus da)\ (b \oplus db) - div\ a\ b
\end{aligned}
$$

Function *ddiv* computes the difference between the updated and the original result, so it needs its base inputs a and b. Hence, *daverage* must recompute s and n and will be slower than *average*!

Typically, ILC derivatives are only efficient if they are *self-maintainable*: a self-maintainable derivative does not inspect its base inputs, but only its change inputs, so recomputation of its base inputs can be elided. Cai et al. [7] leave efficient support for non-self-maintainable derivatives for future work.

But this problem is fixable: executing *daverage xs dxs* will compute exactly the same s and n as executing *average xs*, so to avoid recomputation we must simply save s and n and reuse them. Hence, we CTS-convert each function f to a *CTS function fC* and a *CTS derivative dfC*: CTS function *fC* produces, together with its final result, a *cache* containing intermediate results, that the caller must pass to CTS derivative *dfC*.

CTS-converting our example produces the following code, which requires no wasteful recomputation.

$$
\begin{aligned}
&\textbf{type } \mathit{AverageC} = (\mathbb{Z}, \mathit{SumC}, \mathbb{Z}, \mathit{LengthC}, \mathbb{Z}, \mathit{DivC})\\
&\mathit{averageC} :: \mathit{Bag}\ \mathbb{Z} \to (\mathbb{Z}, \mathit{AverageC})\\
&\mathit{averageC}\ \mathit{xs} =\\
&\quad \textbf{let } (s, cs_1) = \mathit{sumC}\ \mathit{xs}; (n, cn_1) = \mathit{lengthC}\ \mathit{xs}; (r, cr_1) = \mathit{divC}\ s\ n\\
&\quad \textbf{in } (r, (s, cs_1, n, cn_1, r, cr_1))\\
&\mathit{daverageC} :: \mathit{Bag}\ \mathbb{Z} \to \Delta(\mathit{Bag}\ \mathbb{Z}) \to \mathit{AverageC} \to (\Delta\mathbb{Z}, \mathit{AverageC})\\
&\mathit{daverageC}\ \mathit{xs}\ \mathit{dxs}\ (s, cs_1, n, cn_1, r, cr_1) =\\
&\quad \textbf{let } (ds, cs_2) = \mathit{dsumC}\ \mathit{xs}\ \mathit{dxs}\ cs_1\\
&\qquad\quad (dn, cn_2) = \mathit{dlengthC}\ \mathit{xs}\ \mathit{dxs}\ cn_1\\
&\qquad\quad (dr, cr_2) = \mathit{ddivC}\ s\ ds\ n\ dn\ cr_1\\
&\quad \textbf{in } (dr, ((s \oplus ds), cs_2, (n \oplus dn), cn_2, (r \oplus dr), cr_2))
\end{aligned}
$$

For each function f, we introduce a type FC for its cache, such that a CTS function *fC* has type $A \to (B, FC)$ and CTS derivative *dfC* has type $A \to \Delta A \to FC \to (\Delta B, FC)$. Crucially, CTS derivatives like *daverageC* must return an updated cache to ensure correct incrementalization, so that application of further changes works correctly. In general, if $(y_1, c_1) = fC\ x_1$ and $(dy, c_2) = dfC\ x_1\ dx\ c_1$, then $(y_1 \oplus dy, c_2)$ must equal the result of the base function *fC* applied to the updated input $x_1 \oplus dx$, that is $(y_1 \oplus dy, c_2) = fC\ (x_1 \oplus dx)$.

For CTS-converted functions, the cache type FC is a tuple of intermediate results and caches of subcalls. For primitive functions like *div*, the cache type *DivC* could contain information needed for efficient computation of output changes. In the case of *div*, no additional information is needed. The definition of *divC* uses *div* and produces an empty cache, and the definition of *ddivC* follows the earlier definition for *ddiv*, except that we now pass along an empty cache.

$$
\begin{aligned}
&\textbf{data } \mathit{DivC} = \mathit{DivC}\\
&\mathit{divC} :: \mathbb{Z} \to \mathbb{Z} \to (\mathbb{Z}, \mathit{DivC})\\
&\mathit{divC}\ a\ b = (\mathit{div}\ a\ b, \mathit{DivC})\\
&\mathit{ddivC} :: \mathbb{Z} \to \Delta\mathbb{Z} \to \mathbb{Z} \to \Delta\mathbb{Z} \to \mathit{DivC} \to (\Delta\mathbb{Z}, \mathit{DivC})\\
&\mathit{ddivC}\ a\ da\ b\ db\ \mathit{DivC} = (\mathit{div}\ (a \oplus da)\ (b \oplus db) - \mathit{div}\ a\ b, \mathit{DivC})
\end{aligned}
$$

Finally, we can rewrite $average_3$ to incrementally compute y_2 and y_3:

$$
\begin{aligned}
&ctsIncrementalAverage_3 :: (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}) \\
&ctsIncrementalAverage_3 = \\
&\quad \textbf{let } (y_1, c_1) = averageC\ xs_1; (dy_1, c_2) = daverageC\ xs_1\ dxs_1\ c_1 \\
&\qquad y_2 = y_1 \oplus dy_1; (dy_2, c_3) = daverageC\ xs_2\ dxs_2\ c_2 \\
&\qquad y_3 = y_2 \oplus dy_2 \\
&\quad \textbf{in } (y_1, y_2, y_3)
\end{aligned}
$$

Since functions of the same type translate to CTS functions of different types, in a higher-order language CTS translation is not always type-preserving; however, this is not a problem for our case studies (Sect. 4); Sect. 4.1 shows how to map such functions, and we return to this problem in Sect. 4.4.

3 Formalization

We now formalize CTS-differentiation for an untyped Turing-complete λ-calculus, and formally prove it sound with respect to differentiation. We also give a novel proof of correctness for differentiation itself, since we cannot simply adapt Cai et al. [7]'s proof to the new syntax: Our language is untyped and Turing-complete, while Cai et al. [7]'s proof assumed a strongly normalizing simply-typed λ-calculus and relied on its naive set-theoretic denotational semantics. Our entire formalization is mechanized using Coq [26]. For reasons of space, some details are deferred to the appendix.

	Terms	
$a_t ::= \textbf{let}\, a_p = a_t\, \textbf{in}\, a_t$	*Let*	
$\mid a_T$	*Tuple*	
$\mid f\ \overline{x}$	*Application*	
	Nested tuples	
$a_T ::= x$	*Variable*	
$\mid x \oplus dx$	*Update*	
$\mid (\overline{a_T})$	*Tuple*	
	Patterns	
$a_p ::= x$	*Variable*	
$\mid (\overline{a_p})$	*Tuple*	
	Closed values	
$a_v ::= a_E[\lambda \overline{a_p}.\, a_t]$	*Closure*	
$\mid (\overline{a_v})$	*Tuple*	
$\mid \ell$	*Literal*	
$\mid \mathbf{p}$	*Primitive*	
$\mid 0_{\mathbf{p}}$	*Nil change for primitive*	
$\mid\ !a_v$	*Replacement change*	
	Value environments	
$a_E ::= \bullet$	*Empty*	
$\mid a_E; x = a_v$	*Value binding*	
$j, k, n \in \mathbb{N}$	***Step indexes***	

Fig. 1. Our language λ_L of lambda-lifted programs. Tuples can be nullary.

Transformations. We introduce and prove sound three term transformations, namely differentiation, CTS translation and CTS differentiation, that take a function to its corresponding (non-CTS) derivative, CTS function and CTS derivative. Each CTS function produces a base output and a cache from a base input, while each CTS derivative produces an output change and an updated cache from an input, an input change and a base cache.

Proof technique. To show soundness, we prove that CTS functions and derivatives simulate respectively non-CTS functions and derivatives. In turn, we formalize (non-CTS) differentiation as well, and we prove differentiation sound with respect to non-incremental evaluation. Overall, this shows that CTS functions and derivatives are sound relatively to non-incremental evaluation. Our presentation proceeds in the converse order: first, we present differentiation, formulated as a variant of Cai et al. [7]'s definition; then, we study CTS differentiation.

By using logical relations, we simplify significantly the setup of Cai et al. [7]. To handle an untyped language, we employ *step-indexed* logical relations. Besides, we conduct our development with big-step operational semantics because that choice simplifies the correctness proof for CTS conversion. Using big-step semantics for a Turing complete language restricts us to terminating computations. But that is not a problem: to show incrementalization is correct, we need only consider computations that terminate on both old and new inputs, following Acar et al. [3] (compared with in Sect. 5).

Structure of the formalization. Section 3.1 introduces the syntax of the language λ_L we consider in this development, and introduces its four sublanguages λ_{AL}, λ_{IAL}, λ_{CAL} and λ_{ICAL}. Section 3.2 presents the syntax and the semantics of λ_{AL}, the source language for our transformations. Section 3.3 defines differentiation and its target language λ_{IAL}, and Sect. 3.4 proves differentiation correct. Section 3.5 defines CTS conversion, comprising CTS translation and CTS differentiation, and their target languages λ_{CAL} and λ_{ICAL}. Section 3.6 presents the semantics of λ_{CAL}. Finally, Sect. 3.7 proves CTS conversion correct.

Notations. We write $\overline{X}$ for a sequence of X of some unspecified length $X_1, \ldots, X_m$.

3.1 Syntax for λ_L

A superlanguage. To simplify our transformations, we require input programs to have been lambda-lifted [15] and converted to A'-normal form (A'NF). Lambda-lifted programs are convenient because they allow us to avoid a specific treatment for free variables in transformations. A'NF is a minor variant of ANF [24], where every result is bound to a variable before use; unlike ANF, we also bind the result of the tail call. Thus, every result can thus be stored in a cache by CTS conversion and reused later (as described in Sect. 2). This requirement is not onerous: A'NF is a minimal variant of ANF, and lambda-lifting and ANF conversion are routine in compilers for functional languages. Most examples we show are in this form.

In contrast, our transformation's outputs are lambda-lifted but not in A'NF. For instance, we restrict base functions to take exactly one argument—a base input. As shown in Sect. 2.1, CTS functions take instead two arguments—a base input and a cache—and CTS derivatives take three arguments—an input, an input change, and a cache. We could normalize transformation outputs to inhabit the source language and follow the same invariants, but this would complicate our proofs for little benefit. Hence, we do not *prescribe* transformation outputs

to satisfy the same invariants, and we rather *describe* transformation outputs through separate grammars.

As a result of this design choice, we consider languages for base programs, derivatives, CTS programs and CTS derivatives. In our Coq mechanization, we formalize those as four separate languages, saving us many proof steps to check the validity of required structural invariants. For simplicity, in this paper we define a single language called λ_L (for λ-Lifted). This language satisfies invariants common to all these languages (including some of the A'NF invariants). Then, we define *sublanguages* of λ_L. We describe the semantics of λ_L informally, and we only formalize the semantics of its sublanguages.

Syntax for terms. The λ_L language is a relatively conventional lambda-lifted λ-calculus with a limited form of pattern matching on tuples. The syntax for terms and values is presented in Fig. 1. We separate terms and values in two distinct syntactic classes because we use big-step operational semantics. Our **let**-bindings are non-recursive as usual, and support shadowing. Terms cannot contain λ-expressions directly, but only refer to closures through the environment, and similarly for literals and primitives; we elaborate on this in Sect. 3.2. We do not introduce case expressions, but only bindings that destructure tuples, both in **let**-bindings and λ-expressions of closures. Our semantics does not assign meaning to match failures, but pattern-matchings are only used in generated programs and our correctness proofs ensure that the matches always succeed. We allow tuples to contain terms of form $x \oplus dx$, which update base values x with changes in dx, because A'NF-converting these updates is not necessary to the transformations. We often inspect the result of a function call "$f\ x$", which is not a valid term in our syntax. Hence, we write "$@(f, x)$" as a syntactic sugar for "$\mathbf{let}\ y = f\ x\ \mathbf{in}\ y$" with y chosen fresh.

Syntax for closed values. A closed value is either a closure, a tuple of values, a literal, a primitive, a nil change for a primitive or a replacement change. A closure is a pair of an evaluation environment E and a λ-abstraction closed with respect to E. The set of available literals ℓ is left abstract. It may contain usual first-order literals like integers. We also leave abstract the primitives $\mathbf{p}$ like `if-then-else` or projections of tuple components. Each primitive $\mathbf{p}$ comes with a nil change, which is its derivative as explained in Sect. 2. A change value can also represent a replacement by some closed value a_v. Replacement changes are not produced by static differentiation but are useful for clients of derivatives: we include them in the formalization to make sure that they are not incompatible with our system. As usual, environments E map variables to closed values.

Sublanguages of λ_L. The source language for all our transformations is a sublanguage of λ_L named λ_{AL}, where A stands for A'NF. To each transformation we associate a target language, which matches the transformation image. The target language for CTS conversion is named λ_{CAL}, where "C" stands for CTS. The target languages of differentiation and CTS differentiation are called, respectively, λ_{IAL} and λ_{ICAL}, where the "I" stands for incremental.

3.2 The Source Language λ_{AL}

We show the syntax of λ_{AL} in Fig. 2. As said above, λ_{AL} is a sublanguage of λ_L denoting lambda-lifted base terms in A'NF. With no loss of generality, we assume that all bound variables in λ_{AL} programs and closures are distinct. The step-indexed big-step semantics (Fig. 3) for base terms is defined by the judgment written $E \vdash t \Downarrow_n v$ (where n can be omitted) and pronounced "Under environment E, base term t evaluates to closed value v in n steps." Intuitively, our step-indexes count the number of "nodes" of a big-step derivation.[1] As they are relatively standard, we defer the explanations of these rules to Appendix B.

Term differentiation $\boxed{dt = \mathcal{D}^\iota(t)}$

$$\begin{aligned}
\mathcal{D}^\iota(x) &= dx \\
\mathcal{D}^\iota(\mathbf{let}\ y = f\ x\ \mathbf{in}\ t) &= \\
&\quad \mathbf{let}\ y = f\ x, dy = df\ x\ dx\ \mathbf{in}\ \mathcal{D}^\iota(t) \\
\mathcal{D}^\iota(\mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ t) &= \\
&\quad \mathbf{let}\ y = (\overline{x}), dy = (\overline{dx})\ \mathbf{in}\ \mathcal{D}^\iota(t)
\end{aligned}$$

Value differentiation $\boxed{dv = \mathcal{D}^\iota(v)}$

$$\begin{aligned}
\mathcal{D}^\iota((\overline{v})) &= (\overline{\mathcal{D}^\iota(v)}) \\
\mathcal{D}^\iota(E_f[\lambda x.\, t]) &= \mathcal{D}^\iota(E_f)[\lambda x\ dx.\, \mathcal{D}^\iota(t)] \\
\mathcal{D}^\iota(\ell) &= \mathbf{nil}\,\ell \\
\mathcal{D}^\iota(\mathbf{p}) &= 0_{\mathbf{p}}
\end{aligned}$$

Environment differentiation $\boxed{dE = \mathcal{D}^\iota(E)}$

$$\begin{aligned}
\mathcal{D}^\iota(\bullet) &= \bullet \\
\mathcal{D}^\iota(E; x = v) &= \mathcal{D}^\iota(E); x = v; dx = \mathcal{D}^\iota(v)
\end{aligned}$$

Base/updated environment $\boxed{E = \lfloor dE \rfloor_i}$

$$\begin{aligned}
\lfloor \bullet \rfloor_i &= \bullet \qquad \mathrm{i} = 1, 2 \\
\lfloor dE; x = v; dx = dv \rfloor_i &= \lfloor dE \rfloor_i; x = v' \\
&\quad v' = v \text{ if } i = 1 \text{ or} \\
&\quad v' = v \oplus dv \text{ if } i = 2
\end{aligned}$$

λ_{IAL} **change terms**

$$\begin{aligned}
dt ::= {} & dx \\
\mid {} & \mathbf{let}\ y = f\ x, dy = df\ x\ dx \\
& \mathbf{in}\ dt \\
\mid {} & \mathbf{let}\ y = (\overline{x}), dy = (\overline{dx}) \\
& \mathbf{in}\ dt
\end{aligned}$$

λ_{IAL} **change values**

$$dv ::= (\overline{dv}) \mid dE[\lambda x\ dx.\, dt] \mid d\ell \mid 0_{\mathbf{p}} \mid\ !v$$

λ_{IAL} **change environments**

$$dE ::= \bullet \mid dE; x = v; dx = dv$$

λ_{AL} **base terms**

$$\begin{aligned}
t ::= {} & x \mid \mathbf{let}\ y = f\ x\ \mathbf{in}\ t \mid \\
& \mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ t
\end{aligned}$$

λ_{AL} **closed values**

$$v ::= (\overline{v}) \mid E[\lambda x.\, t] \mid \ell \mid \mathbf{p}$$

λ_{AL} **value environments**

$$E ::= \bullet \mid E; x = v$$

Fig. 2. Static differentiation $\mathcal{D}^\iota(-)$; syntax of its target language λ_{IAL}, tailored to the output of differentiation; syntax of its source language λ_{AL}. We assume that in λ_{IAL} the same **let** binds both y and dy and that α-renaming preserves this invariant. We also define the *base environment* $\lfloor dE \rfloor_1$ and the *updated environment* $\lfloor dE \rfloor_2$ of a change environment dE.

Expressiveness. A closure in the base environment can be used to represent a top-level definition. Since environment entries can point to primitives, we need no syntax to directly represent calls of primitives in the syntax of base terms. To encode in our syntax a program with top-level definitions and a term to be evaluated representing the entry point, one can produce a term t representing the

[1] It is more common to count instead small-step evaluation steps [3,4], but our choice simplifies some proofs and makes a minor difference in others.

$$\frac{}{E \vdash x \Downarrow_1 E(x)}\ [\textsc{SVar}] \qquad \frac{E; y = (E(\overline{x})) \vdash t \Downarrow_n v}{E \vdash \mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ t \Downarrow_{n+1} v}\ [\textsc{STuple}] \qquad \frac{E(f) = \mathbf{p} \quad E; y = \delta_{\mathbf{p}}(E(x)) \vdash t \Downarrow_n v}{E \vdash \mathbf{let}\ y = f\ x\ \mathbf{in}\ t \Downarrow_{n+1} v}\ [\textsc{SPrimitiveCall}]$$

$$\frac{E(f) = E_f[\lambda x.\, t_f] \qquad E_f; x = E(x) \vdash t_f \Downarrow_m v_y \qquad E; y = v_y \vdash t \Downarrow_n v}{E \vdash \mathbf{let}\ y = f\ x\ \mathbf{in}\ t \Downarrow_{m+n+1} v}\ [\textsc{SClosureCall}]$$

Fig. 3. Step-indexed big-step semantics for base terms of source language λ_{AL}.

entry point together with an environment E containing as values any top-level definitions, primitives and literals used in the program. Semi-formally, given an environment E_0 mentioning needed primitives and literals, and a list of top-level function definitions $D = \overline{f = \lambda x.\, t}$ defined in terms of E_0, we can produce a base environment $E = \mathcal{L}(D)$, with $\mathcal{L}$ defined by:

$$\mathcal{L}(\bullet) = E_0 \text{ and } \mathcal{L}(D, f = \lambda x.\, t) = E, f = E[\lambda x.\, t] \text{ where } \mathcal{L}(D) = E$$

Correspondingly, we extend all our term transformations to values and environments to transform such encoded top-level definitions.

Our mechanization can encode n-ary functions "$\lambda(x_1, x_2, \ldots, x_n).\, t$" through unary functions that accept tuples; we encode partial application using a **curry** primitive such that, essentially, $\mathbf{curry}\ f\ x\ y = f\ (x, y)$; suspended partial applications are represented as closures. This encoding does not support currying efficiently, we further discuss this limitation in Sect. 4.4.

Control operators, like recursion combinators or branching, can be introduced as primitive operations as well. If the branching condition changes, expressing the output change in general requires replacement changes. Similarly to branching we can add tagged unions.

To check the assertions of the last two paragraphs, the Coq development contains the definition of a **curry** primitive as well as a primitive for a fixpoint combinator, allowing general recursion and recursive data structures as well.

3.3 Static Differentiation from λ_{AL} to λ_{IAL}

Previous work [7] defines static differentiation for simply-typed λ-calculus terms. Figure 2 transposes differentiation as a transformation from λ_{AL} to λ_{IAL} and defines λ_{IAL}'s syntax.

Differentiating a base term t produces a change term $\mathcal{D}^\iota(t)$, its *derivative*. Differentiating final result variable x produces its change variable dx. Differentiation copies each binding of an intermediate result y to the output and adds a new binding for its change dy. If y is bound to tuple $(\overline{x})$, then dy will be bound to the change tuple $(\overline{dx})$. If y is bound to function application "$f\ x$", then dy will be bound to the application of function change df to input x and its change dx. We explain differentiation of environments $\mathcal{D}^\iota(E)$ later in this section.

$$\frac{}{dE \vdash dx \Downarrow_1 dE(dx)}\ [\text{SDVar}] \qquad \frac{dE(\overline{x}, \overline{dx}) = \overline{v}_x, \overline{dv}_x \quad dE; y = (\overline{v}_x); dy = (\overline{dv}_x) \vdash dt \Downarrow_n dv}{dE \vdash \textbf{let}\ y = (\overline{x}), dy = (\overline{dx})\ \textbf{in}\ dt \Downarrow_{n+1} dv}\ [\text{SDTuple}]$$

$$\frac{\lfloor dE \rfloor_1 \vdash @(f,x) \Downarrow_m v_y \qquad \lfloor dE \rfloor_2 \vdash @(f,x) \Downarrow_n {v_y}' \quad dE(df) = !v_f \qquad dE; y = v_y; dy = !{v_y}' \vdash dt \Downarrow_p dv}{dE \vdash \textbf{let}\ y = f\ x, dy = df\ x\ dx\ \textbf{in}\ dt \Downarrow_{m+n+p+1} dv}\ [\text{SDReplaceCall}]$$

$$\frac{dE(f, df) = \mathbf{p}, 0_{\mathbf{p}} \qquad dE(x, dx) = v_x, dv_x \quad dE; y = \delta_{\mathbf{p}}(v_x); dy = \Delta_{\mathbf{p}}(v_x, dv_x) \vdash dt \Downarrow_n dv}{dE \vdash \textbf{let}\ y = f\ x, dy = df\ x\ dx\ \textbf{in}\ dt \Downarrow_{n+1} dv}\ [\text{SDPrimitiveNil}]$$

$$\frac{dE(f, df) = E_f[\lambda x.\, t_f], dE_f[\lambda x\ dx.\, dt_f] \quad dE(x, dx) = v_x, dv_x \qquad E_f; x = v_x \vdash t_f \Downarrow_m v_y \quad dE_f; x = v_x; dx = dv_x \vdash dt_f \Downarrow_n dv_y \qquad dE; y = v_y; dy = dv_y \vdash dt \Downarrow_p dv}{dE \vdash \textbf{let}\ y = f\ x, dy = df\ x\ dx\ \textbf{in}\ dt \Downarrow_{m+n+p+1} dv}\ [\text{SDClosureChange}]$$

Fig. 4. Step-indexed big-step semantics for the change terms of λ_{IAL}.

Evaluating $\mathcal{D}^\iota(t)$ recomputes all intermediate results computed by t. This recomputation will be avoided through cache-transfer style in Sect. 3.5. A comparison with the original static differentiation [7] can be found in Appendix A.

Semantics for λ_{IAL}. We move on to define how λ_{IAL} change terms evaluate to change values. We start by defining necessary definitions and operations on changes, such as define *change values* dv, *change environments* dE, and the *update operator* $\oplus$.

Closed change values dv are particular λ_L values a_v. They are either a closure change, a tuple change, a literal change, a replacement change or a primitive nil change. A closure change is a closure containing a change environment dE and a λ-abstraction expecting a value and a change value as arguments to evaluate a change term into an output change value. An evaluation environment dE follows the same structure as **let**-bindings of change terms: it binds variables to closed values and each variable x is immediately followed by a binding for its associated change variable dx. As with **let**-bindings of change terms, α-renamings in an environment dE must rename dx into dy if x is renamed into y. We define the *update operator* $\oplus$ to update a value with a change. This operator is a partial function written "$v \oplus dv$", defined as follows:

$$
\begin{aligned}
v_1 \oplus !v_2 &= v_2 \\
\ell \oplus d\ell &= \delta_\oplus(\ell, d\ell) \\
E[\lambda x.\, t] \oplus dE[\lambda x\, dx.\, dt] &= (E \oplus dE)[\lambda x.\, t] \\
(v_1, \ldots, v_n) \oplus (dv_1, \ldots, dv_n) &= (v_1 \oplus dv_1, \ldots, v_n \oplus dv_n) \\
\mathbf{p} \oplus 0_{\mathbf{p}} &= \mathbf{p}
\end{aligned}
$$

where $(E; x = v) \oplus (dE; x = v; dx = dv) = ((E \oplus dE); x = (v \oplus dv))$.

Replacement changes can be used to update all values (literals, tuples, primitives and closures), while tuple changes can only update tuples, literal changes can only update literals, primitive nil can only update primitives and closure changes can only update closures. A replacement change overrides the current value v with a new one v'. On literals, $\oplus$ is defined via some interpretation function $\delta_\oplus$, which takes a literal and a literal change to produce an updated literal. Change update for a closure ignores dt instead of computing something like $dE[t \oplus dt]$. This may seem surprising, but we only need $\oplus$ to behave well for valid changes (as shown by Theorem 3.1): for valid closure changes, dt must behave anyway similarly to $\mathcal{D}^\iota(t)$, which Cai et al. [7] show to be a nil change. Hence, $t \oplus \mathcal{D}^\iota(t)$ and $t \oplus dt$ both behave like t, so $\oplus$ can ignore dt and only consider environment updates. This definition also avoids having to modify terms at runtime, which would be difficult to implement safely. We could also implement $f \oplus df$ as a function that invokes both f and df on its argument, as done by Cai et al. [7], but we believe that would be less efficient when $\oplus$ is used at runtime. As we discuss in Sect. 3.4, we restrict validity to avoid this runtime overhead.

Having given these definitions, we show in Fig. 4 a step-indexed big-step semantics for change terms, defined through judgment $dE \vdash dt \Downarrow_n dv$ (where n can be omitted). This judgment is pronounced "Under the environment dE, the change term dt evaluates into the closed change value dv in n steps." Rules [SDVar] and [SDTuple] are unsurprising. To evaluate function calls in **let**-bindings "**let** $y = f\, x, dy = df\, x\, dx$ **in** dt" we have three rules, depending on the shape of $dE(df)$. These rules all recompute the value v_y of y in the original environment, but compute differently the change dy to y. If $dE(df)$ replaces the value of f, [SDReplaceCall] recomputes $v'_y = f\, x$ from scratch in the new environment, and bind dy to $!v'_y$ when evaluating the **let** body. If $dE(df)$ is the nil change for primitive $\mathbf{p}$, [SDPrimitiveNil] computes dy by running $\mathbf{p}$'s derivative through function $\Delta_{\mathbf{p}}(-)$. If $dE(df)$ is a closure change, [SDClosureChange] invokes it normally to compute its change dv_y. As we show, if the closure change is valid, its body behaves like f's derivative, hence incrementalizes f correctly.

Closure changes with non-nil environment changes represent partial application of derivatives to non-nil changes; for instance, if f takes a pair and dx is a non-nil change, $0_{\mathbf{curry}}\, f\, df\, x\, dx$ constructs a closure change containing dx, using the derivative of **curry** mentioned in Sect. 3.2. In general, such closure changes do not arise from the rules we show, only from derivatives of primitives.

3.4 A New Soundness Proof for Static Differentiation

In this section, we show that static differentiation is sound (Theorem 3.3) and that Eq. (1) holds:

$$f\ a_2 = f\ a_1 \oplus \mathcal{D}^\iota(f)\ a_1\ da \tag{1}$$

whenever da is a valid change from a_1 to a_2 (as defined later). One might want to prove this equation assuming only that $a_1 \oplus da = a_2$, but this is false in general. A direct proof by induction on terms fails in the case for application (ultimately because $f_1 \oplus df = f_2$ and $a_1 \oplus da = a_2$ do not imply that $f_1\ a_1 \oplus df\ a_1\ da = f_2\ a_2$). As usual, this can be fixed by introducing a logical relation. We call ours *validity*: a function change is valid if it turns valid input changes into valid output changes.

- $d\ell \rhd_n \ell \hookrightarrow \delta_\oplus(\ell, d\ell)$ $\quad$ • $!v_2 \rhd_n v_1 \hookrightarrow v_2$ $\quad$ • $0_{\mathbf{p}} \rhd_n \mathbf{p} \hookrightarrow \mathbf{p}$
- $(dv_1, \ldots, dv_m) \rhd_n (v_1, \ldots, v_m) \hookrightarrow (v'_1, \ldots, v'_m)$
  if and only if $(v_1, \ldots, v_m) \oplus (dv_1, \ldots, dv_m) = (v'_1, \ldots, v'_m)$
  and $\forall k < n,\ \forall i \in [1 \ldots m],\ dv_i \rhd_k v_i \hookrightarrow v'_i$
- $dE[\lambda x\ dx.\ dt] \rhd_n E_1[\lambda x.\ t] \hookrightarrow E_2[\lambda x.\ t]$
  if and only if $E_2 = E_1 \oplus dE$ and
  $\forall k < n, v_1, dv, v_2,$
  if $dv \rhd_k v_1 \hookrightarrow v_2$ then
  $(dE; x = v_1; dx = dv \vdash dt) \blacktriangleright_k (E_1; x = v_1 \vdash t) \hookrightarrow (E_2; x = v_2 \vdash t)$
- $(dE \vdash dt) \blacktriangleright_n (E_1 \vdash t_1) \hookrightarrow (E_2 \vdash t_2)$
  if and only if $\forall k < n, v_1, v_2,$
  $E_1 \vdash t_1 \Downarrow_k v_1$ and $E_2 \vdash t_2 \Downarrow v_2$ implies that
  $\exists dv, dE \vdash dt \Downarrow dv \wedge dv \rhd_{n-k} v_1 \hookrightarrow v_2$

Fig. 5. Step-indexed validity, through judgments for values and for terms.

Static differentiation is only sound on input changes that are *valid*. Cai et al. [7] show soundness for a strongly normalizing simply-typed λ-calculus using denotational semantics. Using an operational semantics, we generalize this result to an untyped and Turing-complete language, so we must turn to a *step-indexed* logical relation [3,4].

Validity as a step-indexed logical relation. We say that "dv is a valid change from v_1 to v_2, up to k steps" and write

$$dv \rhd_k v_1 \hookrightarrow v_2$$

to mean that dv is a change from v_1 to v_2 and that dv is a *valid* description of the differences between v_1 and v_2, with validity tested with up to k steps. This relation *approximates* validity; if a change dv is valid at all approximations, it is simply valid (between v_1 and v_2); we write then $dv \rhd v_1 \hookrightarrow v_2$ (omitting the step-index k) to mean that validity holds at all step-indexes. We similarly omit step-indexes k from other step-indexed relations when they hold for all k.

To justify this intuition of validity, we show that a valid change from v_1 to v_2 goes indeed from v_1 to v_2 (Theorem 3.1), and that if a change is valid up to k steps, it is also valid up to fewer steps (Lemma 3.2).

Theorem 3.1 ($\oplus$ agrees with validity)
If $dv \triangleright_k v_1 \hookrightarrow v_2$ holds for all $k > 0$, then $v_1 \oplus dv = v_2$.

Lemma 3.2 (Downward-closure)
If $N \geq n$, then $dv \triangleright_N v_1 \hookrightarrow v_2$ implies $dv \triangleright_n v_1 \hookrightarrow v_2$.

Crucially, Theorem 3.1 enables (a) computing v_2 from a valid change and its source, and (b) showing Eq. (1) through validity. As discussed, $\oplus$ ignores changes to closure bodies to be faster, which is only sound if those changes are nil; to ensure Theorem 3.1 still holds, validity on closure changes must be adapted accordingly and forbid non-nil changes to closure bodies. This choice, while unusual, does not affect our results: if input changes do not modify closure bodies, intermediate changes will not modify closure bodies either. Logical relation experts might regard this as a domain-specific invariant we add to our relation. Alternatives are discussed by Giarrusso [10, Appendix C].

As usual with step-indexing, validity is defined by well-founded induction over naturals ordered by $<$; to show well-foundedness we observe that evaluation always takes at least one step.

Validity for values, terms and environments is formally defined by cases in Fig. 5. First, a literal change $d\ell$ is a valid change from ℓ to $\ell \oplus d\ell = \delta_\oplus(\ell, d\ell)$. Since the function $\delta_\oplus$ is partial, the relation only holds for the literal changes $d\ell$ which are valid changes for ℓ. Second, a replacement change $!v_2$ is always a valid change from any value v_1 to v_2. Third, a primitive nil change is a valid change between any primitive and itself. Fourth, a tuple change is valid up to step n, if each of its components is valid up to any step strictly less than n. Fifth, we define validity for closure changes. Roughly speaking, this statement means that a closure change is valid if (i) its environment change dE is valid for the original closure environment E_1 and for the new closure environment E_2; and (ii) when applied to related values, the closure *bodies* t are related by dt, as defined by the auxiliary judgment $(dE \vdash dt) \blacktriangleright_n (E_1 \vdash t_1) \hookrightarrow (E_2 \vdash t_2)$ for validity between terms under related environments (defined in Appendix C). As usual with step-indexed logical relations, in the definition for this judgment about terms, the number k of steps required to evaluate the term t_1 is subtracted from the number of steps n that can be used to relate the outcomes of the term evaluations.

Soundness of differentiation. We can state a soundness theorem for differentiation without mentioning step-indexes; thanks to this theorem, we can compute the updated result v_2 not by rerunning a computation, but by updating the base result v_1 with the result change dv that we compute through a derivative on the input change. A corollary shows Eq. (1).

Theorem 3.3 (Soundness of differentiation in λ_{AL}). *If dE is a valid change environment from base environment E_1 to updated environment E_2, that is $dE \rhd E_1 \hookrightarrow E_2$, and if t converges both in the base and updated environment, that is $E_1 \vdash t \Downarrow v_1$ and $E_2 \vdash t \Downarrow v_2$, then $\mathcal{D}^\iota(t)$ evaluates under the change environment dE to a valid change dv between base result v_1 and updated result v_2, that is $dE \vdash \mathcal{D}^\iota(t) \Downarrow dv$, $dv \rhd v_1 \hookrightarrow v_2$ and $v_1 \oplus dv = v_2$.*

We must first show that derivatives map input changes valid up to k steps to output changes valid up to k steps, that is, the *fundamental property* of our step-indexed logical relation:

Lemma 3.4 (Fundamental Property)
For each n, if $dE \rhd_n E_1 \hookrightarrow E_2$ then $(dE \vdash \mathcal{D}^\iota(t)) \blacktriangleright_n (E_1 \vdash t) \hookrightarrow (E_2 \vdash t)$.

Translation of terms $\boxed{M = \mathcal{T}_t(t')}$

$$\mathcal{T}_t(\mathbf{let}\ y = f\ x\ \mathbf{in}\ t') = \mathbf{let}\ y, c^y_{fx} = f\ x\ \mathbf{in}\ \mathcal{T}_t(t')$$
$$\mathcal{T}_t(\mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ t') = \mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ \mathcal{T}_t(t')$$
$$\mathcal{T}_t(x) = (x, \mathcal{C}(t))$$

Cache of a term $\boxed{C = \mathcal{C}(t)}$

$$\mathcal{C}(\mathbf{let}\ y = f\ x\ \mathbf{in}\ t) = ((\mathcal{C}(t), y), c^y_{fx})$$
$$\mathcal{C}(\mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ t) = (\mathcal{C}(t), y)$$
$$\mathcal{C}(x) = ()$$

Translation of values $\boxed{V = \mathcal{T}(v)}$

$$\mathcal{T}((\overline{v})) = (\overline{\mathcal{T}(v)})$$
$$\mathcal{T}(E[\lambda x.\, t]) = \mathcal{T}(E)[\lambda x.\, \mathcal{T}_t(t)]$$
$$\mathcal{T}(\ell) = \ell$$
$$\mathcal{T}(\mathbf{p}) = \mathbf{p}$$

Base terms

$$M ::= \mathbf{let}\ y, c^y_{fx} = f\ x\ \mathbf{in}\ M \mid \mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ M \mid (x, C)$$

Cache terms/patterns

$$C ::= (C, c^y_{fx}) \mid (C, x) \mid ()$$

Closed values

$$V ::= (\overline{V}) \mid F[\lambda x.\, M] \mid \ell \mid \mathbf{p}$$

Cache values

$$V_c ::= () \mid (V_c, V_c) \mid (V_c, V)$$

Evaluation environments

$$F ::= \bullet \mid F; D_v$$

Base environment entries

$$D_v ::= x = V \mid c^y_{fx} = V_c$$

Fig. 6. Cache-Transfer Style translation and syntax of its target language λ_{CAL}.

3.5 CTS Conversion

Figures 6 and 7 define both the syntax of λ_{CAL} and λ_{ICAL} and CTS conversion. The latter comprises CTS differentiation $\mathcal{D}(-)$, from λ_{AL} to λ_{ICAL}, and CTS translation $\mathcal{T}(-)$, from λ_{AL} to λ_{CAL}.

Syntax definitions for the target languages λ_{CAL} and λ_{ICAL}. Terms of λ_{CAL} follow again λ-lifted A'NF, like λ_{AL}, except that a **let**-binding for a function application "$f\ x$" now binds an extra *cache identifier* c^y_{fx} besides output y. Cache identifiers have non-standard syntax: it can be seen as a triple that refers to the value identifiers f, x and y. Hence, an α-renaming of one of these three identifiers must refresh the cache identifier accordingly. Result terms explicitly

return cache C through syntax (x, C). Caches are encoded through nested tuples, but they are in fact a tree-like data structure that is isomorphic to an execution trace. This trace contains both immediate values and the execution traces of nested function calls.

The syntax for λ_{ICAL} matches the image of the CTS derivative and witnesses the CTS discipline followed by the derivatives: to determine dy, the derivative of f evaluated at point x with change dx expects the cache produced by evaluating y in the base term. The derivative returns the updated cache which contains the intermediate results that would be gathered by the evaluation of $f\,(x \oplus dx)$. The result term of every change term returns the computed change and a cache update dC, where each value identifier x of the input cache is updated with its corresponding change dx.

Differentiation of terms $\boxed{dM = \mathcal{D}_t(t')}$

$$\begin{aligned}
\mathcal{D}_t(\mathbf{let}\ y = f\ x\ \mathbf{in}\ t') &= \mathbf{let}\ dy, c^y_{fx} = df\ x\ dx\ c^y_{fx}\ \mathbf{in} \\
&\quad \mathcal{D}_t(t') \\
\mathcal{D}_t(\mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ t') &= \mathbf{let}\ dy = (\overline{dx})\ \mathbf{in}\ \mathcal{D}_t(M') \\
\mathcal{D}_t(x) &= (dx, \mathcal{U}(t))
\end{aligned}$$

Cache update of a term $\boxed{dC = \mathcal{U}(t)}$

$$\begin{aligned}
\mathcal{U}(\mathbf{let}\ y = f\ x\ \mathbf{in}\ t) &= ((\mathcal{U}(t), y \oplus dy), c^y_{fx}) \\
\mathcal{U}(\mathbf{let}\ y = (\overline{x})\ \mathbf{in}\ t) &= (\mathcal{U}(t), y \oplus dy) \\
\mathcal{U}(x) &= ()
\end{aligned}$$

Differentiation of change values $\boxed{dV = \mathcal{T}(dv)}$

$$\begin{aligned}
\mathcal{T}((\overline{dv})) &= (\overline{\mathcal{T}(dv)}) \\
\mathcal{T}(dE[\lambda x\ dx.\, \mathcal{D}^\iota(t)]) &= \mathcal{T}(dE)[\lambda x\ dx\ (\mathcal{C}(t)).\, \mathcal{D}_t(t)] \\
\mathcal{T}(!v) &= !\mathcal{T}(v) \\
\mathcal{T}(d\ell) &= d\ell \\
\mathcal{T}(0_{\mathbf{p}}) &= 0_{\mathbf{p}}
\end{aligned}$$

Change terms

$$\begin{aligned}
dM ::= {} & \mathbf{let}\ dy, c^y_{fx} = df\ x\ dx\ c^y_{fx} \\
& \mathbf{in}\ dM \\
\mid {} & \mathbf{let}\ dy = (\overline{dx})\ \mathbf{in}\ dM \\
\mid {} & (dx, dC)
\end{aligned}$$

Cache updates

$$\begin{aligned}
dC ::= {} & (dC, c^y_{fx}) \mid (dC, x \oplus dx) \\
\mid {} & ()
\end{aligned}$$

Change values

$$\begin{aligned}
dV ::= {} & (\overline{dV}) \mid dF[\lambda x\ dx\, C.\, dM] \\
\mid {} & d\ell \mid 0_{\mathbf{p}} \mid !V
\end{aligned}$$

Change environments

$$dF ::= \bullet \mid dF; dD_v$$

Change environment entries

$$dD_v ::= D_v \mid dx = dV$$

Fig. 7. CTS differentiation and syntax of its target language λ_{ICAL}. Beware $\mathcal{T}(dE[\lambda x\ dx.\, \mathcal{D}^\iota(t)])$ applies a left-inverse of $\mathcal{D}^\iota(t)$ during pattern matching.

CTS conversion and differentiation. These translations use two auxiliary functions: $\mathcal{C}(t)$ which computes the cache term of a λ_{AL} term t, and $\mathcal{U}(t)$, which computes the cache update of t's derivative.

CTS translation on terms, $\mathcal{T}_t(t')$, accepts as inputs a *global* term t and a subterm t' of t. In tail position ($t' = x$), the translation generates code to return both the result x and the cache $\mathcal{C}(t)$ of the global term t. When the transformation visits **let**-bindings, it outputs extra bindings for caches c^y_{fx} on function calls and visits the **let**-body.

Similarly to $\mathcal{T}_t(t')$, CTS derivation $\mathcal{D}_t(t')$ accepts a global term t and a subterm t' of t. In tail position, the translation returns both the result change dx and the cache update $\mathcal{U}(t)$. On **let**-bindings, it *does not* output bindings for y but for dy, it outputs extra bindings for c^y_{fx} as in the previous case and visits the **let**-body.

To handle function definitions, we transform the base environment E through $\mathcal{T}(E)$ and $\mathcal{T}(\mathcal{D}^\iota(E))$ (translations of environments are done pointwise, see Appendix D). Since $\mathcal{D}^\iota(E)$ includes E, we describe $\mathcal{T}(\mathcal{D}^\iota(E))$ to also cover $\mathcal{T}(E)$. Overall, $\mathcal{T}(\mathcal{D}^\iota(E))$ CTS-converts each source closure $f = E[\lambda x.\, t]$ to a CTS-translated function, with body $\mathcal{T}_t(t)$, and to the CTS derivative df of f. This CTS derivative pattern matches on its input cache using cache pattern $\mathcal{C}(t)$. That way, we make sure that the shape of the cache expected by df is consistent with the shape of the cache produced by f. The body of derivative df is computed by CTS-deriving f's body via $\mathcal{D}_t(t)$.

3.6 Semantics of λ_{CAL} and λ_{ICAL}

An evaluation environment F of λ_{CAL} contains both values and cache values. Values V resemble λ_{AL} values v, cache values V_c match cache terms C and change values dV match λ_{IAL} change values dv. Evaluation environments dF for change terms must also bind change values, so functions in change closures take not just a base input x and an input change dx, like in λ_{IAL}, but also an input cache C. By abuse of notation, we reuse the same syntax C to both deconstruct and construct caches.

Base terms of the language are evaluated using a conventional big-step semantics, consisting of two judgments. Judgment "$F \vdash M \Downarrow (V, V_c)$" is read "Under evaluation environment F, base term M evaluates to value V and cache V_c". The semantics follows the one of λ_{AL}; since terms include extra code to produce and carry caches along the computation, the semantics evaluates that code as well. For space reasons, we defer semantic rules to Appendix E. Auxiliary judgment "$F \vdash C \Downarrow V_c$" evaluates cache terms into cache values: It traverses a cache term and looks up the environment for the values to be cached.

Change terms of λ_{ICAL} are also evaluated using a big-step semantics, which resembles the semantics of λ_{IAL} and λ_{CAL}. Unlike those semantics, evaluating cache updates $(dC, x \oplus dx)$ is evaluated using the $\oplus$ operator (overloaded on λ_{CAL} values and λ_{ICAL} changes). By lack of space, its rules are deferred to Appendix E. This semantics relies on three judgments. Judgment "$dF \vdash dM \Downarrow (dV, V_c)$" is read "Under evaluation environment F, change term dM evaluates to change value dV and updated cache V_c". The first auxiliary judgment "$dF \vdash dC \Downarrow V_c$" defines evaluation of cache update terms. The final auxiliary judgment "$V_c \sim C \to dF$" describes a limited form of pattern matching used by CTS derivatives: namely, how a cache pattern C matches a cache value V_c to produce a change environment dF.

3.7 Soundness of CTS Conversion

The proof is based on a simulation in lock-step, but two subtle points emerge. First, we must relate λ_{AL} environments that do not contain caches, with λ_{CAL} environments that do. Second, while evaluating CTS derivatives, the evaluation environment mixes caches from the base computation and updated caches computed by the derivatives.

Theorem 3.7 follows because differentiation is sound (Theorem 3.3) and evaluation commutes with CTS conversion; this last point requires two lemmas. First, CTS translation of base terms commutes with our semantics:

Lemma 3.5 (Commutation for base evaluations)
For all E, t and v, if $E \vdash t \Downarrow v$, there exists V_c, $\mathcal{T}(E) \vdash \mathcal{T}_t(t) \Downarrow (\mathcal{T}(v), V_c)$.

Second, we need a corresponding lemma for CTS translation of differentiation results: intuitively, evaluating a derivative and CTS translating the resulting change value must give the same result as evaluating the CTS derivative. But to formalize this, we must specify which environments are used for evaluation, and this requires two technicalities.

Assume derivative $\mathcal{D}^\iota(t)$ evaluates correctly in some environment dE. Evaluating CTS derivative $\mathcal{D}_t(t)$ requires cache values from the base computation, but they are not in $\mathcal{T}(dE)$! Therefore, we must introduce a judgment to complete a CTS-translated environment with the appropriate caches (see Appendix F).

Next, consider evaluating a change term of the form $dM = \mathbb{C}[dM']$, where $\mathbb{C}$ is a standard single-hole change-term context—that is, for λ_{ICAL}, a sequence of **let**-bindings. When evaluating dM, we eventually evaluate dM' in a change environment dF updated by $\mathbb{C}$: the change environment dF contains both the updated caches coming from the evaluation of $\mathbb{C}$ and the caches coming from the base computation (which will be updated by the evaluation of dM). Again, a new judgment, given in Appendix F, is required to model this process.

With these two judgments, the second key Lemma stating the commutation between evaluation of derivatives and evaluation of CTS derivatives can be stated. We give here an informal version of this Lemma, the actual formal version can be found in Appendix F.

Lemma 3.6 (Commutation for derivatives evaluation)
If the evaluation of $\mathcal{D}^\iota(t)$ leads to an environment dE_0 when it reaches the differentiated context $\mathcal{D}^\iota(\mathbb{C})$ where $t = \mathbb{C}[t']$, and if the CTS conversion of t under this environment completed with base (resp. changed) caches evaluates into a base value $\mathcal{T}(v)$ (resp. a changed value $\mathcal{T}(v')$) and a base cache value V_c (resp. an updated cache value V_c'), then under an environment containing the caches already updated by the evaluation of $\mathcal{D}^\iota(\mathbb{C})$ and the base caches to be updated, the CTS derivative of t' evaluates to $\mathcal{T}(dv)$ such that $v \oplus dv = v'$ and to the updated cache V_c'.

Finally, we can state soundness of CTS differentiation. This theorem says that CTS derivatives not only produce valid changes for incrementalization but that they also correctly consume and update caches.

Theorem 3.7 (Soundness of CTS differentiation)
If the following hypotheses hold:

1. $dE \rhd E \hookrightarrow E'$
2. $E \vdash t \Downarrow v$
3. $E' \vdash t \Downarrow v'$

then there exists dv, V_c, V_c' *and* F_0 *such that:*

1. $\mathcal{T}(E) \vdash \mathcal{T}(t) \Downarrow (\mathcal{T}(v), V_c)$
2. $\mathcal{T}(E') \vdash \mathcal{T}(t) \Downarrow (\mathcal{T}(v'), V_c')$
3. $\mathcal{C}(t) \sim V_c \to F_0$
4. $\mathcal{T}(dE); F_0 \vdash \mathcal{D}_t(t) \Downarrow (\mathcal{T}(dv), V_c')$
5. $v \oplus dv = v'$

4 Incrementalization Case Studies

In this section, we investigate two questions: whether our transformations can target a typed language like Haskell and whether automatically transformed programs can perform well. We implement by hand primitives on sequences, bags and maps in Haskell. The input terms in all case studies are written in a deep embedding of λ_{AL} into Haskell. The transformations generate Haskell code that uses our primitives and their derivatives.

We run the transformations on three case studies: a computation of the average value of a bag of integers, a nested loop over two sequences and a more involved example inspired by Koch et al. [17]'s work on incrementalizing database queries. For each case study, we make sure that results are consistent between from scratch recomputation and incremental evaluation; we measure the execution time for from scratch recomputation and incremental computation as well as the space consumption of caches. We obtain efficient incremental programs, that is ones for which incremental computation is faster than from scratch recomputation. The measurements indicate that we do get the expected asymptotic improvement in time of incremental computation over from scratch recomputation by a linear factor while the caches grows in a similar linear factor.

Our benchmarks were compiled by GHC 8.2.2 and run on a 2.20 GHz hexa core Intel(R) Xeon(R) CPU E5-2420 v2 with 32 GB of RAM running Ubuntu 14.04. We use the *criterion* [21] benchmarking library.

4.1 Averaging Bags of Integers

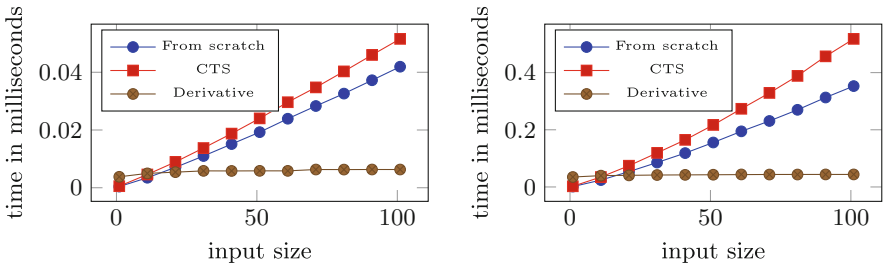
Section 2.1 motivates our transformation with a running example of computing the average over a bag of integers. We represent bags as maps from elements to (possibly negative) multiplicities. Earlier work [7,17] represents bag changes as bags of removed and added elements. We use a different representation of bag changes that takes advantage of the changes to elements and provide primitives on bags and their derivatives. The CTS variant of *map*, that we call *mapC*, takes a function *fC* in CTS and a bag *as* and produces a bag and a cache. The cache stores for each invocation of *fC*, and therefore for each distinct element in *as*, the result of *fC* of type b and the cache of type c.

Inspired by Rossberg et al. [23], all higher-order functions (and typically, also their caches) are parametric over cache types of their function arguments. Here, functions *mapC* and *dmapC* and cache type *MapC* are parametric over the cache type c of *fC* and *dfC*.

$$map :: (a \to b) \to Bag\ a \to Bag\ b$$
$$\textbf{data}\ MapC\ a\ b\ c = MapC\ (Map\ a\ (b, c))$$
$$mapC :: (a \to (b, c)) \to Bag\ a \to (Bag\ b, MapC\ a\ b\ c)$$
$$dmapC :: (a \to (b, c)) \to (a \to \Delta a \to c \to (\Delta b, c)) \to Bag\ a \to \Delta(Bag\ a) \to MapC\ a\ b\ c \to (\Delta(Bag\ b), MapC\ a\ b\ c)$$

We wrote the *length* and *sum* functions used in our benchmarks in terms of primitives *map* and *foldGroup* and had their CTS function and CTS derivative generated automatically.

We evaluate whether we can produce an updated result with *daverageC* shown in Sect. 2.1 faster than by from scratch recomputation with *average*. We expect the speedup of *daverageC* to depend on the size of the input bag n. We fix an input bag of size n as the bag containing the numbers from 1 to n. We define a change that inserts the integer 1 into the bag. To measure execution time of from scratch recomputation, we apply *average* to the input bag updated with the change. To measure execution time of the CTS function *averageC*, we apply *averageC* to the input bag updated with the change. To measure execution time of the CTS derivative *daverageC*, we apply *daverageC* to the input bag, the change and the cache produced by *averageC* when applied to the input bag. In all three cases we ensure that all results and caches are fully forced so as to not hide any computational cost behind laziness.



(a) Benchmark results for *average* (b) Benchmark results for *totalPrice*

Fig. 8. Benchmark results for *average* and *totalPrice*

The plot in Fig. 8a shows execution time versus the size n of the base input. To produce the base result and cache, the CTS transformed function *averageC* takes longer than the original *average* function takes to produce just the result. Producing the updated result incrementally is slower than from scratch recomputation for small input sizes, but because of the difference in time complexity becomes faster as the input size grows. The size of the cache grows linearly with the size of the input, which is not optimal for this example. We leave optimizing the space usage of examples like this to future work.

4.2 Nested Loops over Two Sequences

Next, we consider CTS differentiation on a higher-order example. To incrementalize this example efficiently, we have to enable detecting nil function changes at runtime by representing function changes as closures that can be inspected by incremental programs. Our example here is the Cartesian product of two sequences computed in terms of functions *map* and *concat*.

$$
\begin{aligned}
&\mathit{cartesianProduct} :: \mathit{Sequence}\ a \to \mathit{Sequence}\ b \to \mathit{Sequence}\ (a, b)\\
&\mathit{cartesianProduct}\ \mathit{xs}\ \mathit{ys} = \mathit{concatMap}\ (\lambda x \to \mathit{map}\ (\lambda y \to (x, y))\ \mathit{ys})\ \mathit{xs}\\
&\mathit{concatMap} :: (a \to \mathit{Sequence}\ b) \to \mathit{Sequence}\ a \to \mathit{Sequence}\ b\\
&\mathit{concatMap}\ f\ \mathit{xs} = \mathit{concat}\ (\mathit{map}\ f\ \mathit{xs})
\end{aligned}
$$

We implemented incremental sequences and related primitives following Firsov and Jeltsch [9]: our change operations and first-order operations (such as *concat*) reuse their implementation. On the other hand, we must extend higher-order operations such as *map* to handle non-nil function changes and caching. A correct and efficient CTS derivative *dmapC* has to work differently depending on whether the given function change is nil or not: For a non-nil function change it has to go over the input sequence; for a nil function change it has to avoid that.

Cai et al. [7] use static analysis to conservatively approximate nil function changes as changes to terms that are closed in the original program. But in this example the function argument $(\lambda y \to (x, y))$ to *map* in *cartesianProduct* is not a closed term. It is, however, crucial for the asymptotic improvement that we avoid looping over the inner sequence when the change to the free variable x in the change environment is $\mathbf{0}_x$.

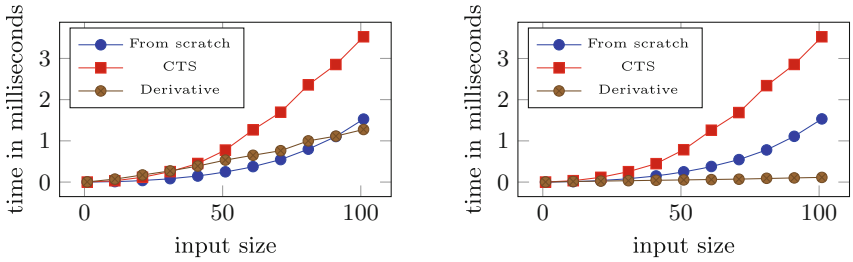
To enable runtime nil change detection, we apply closure conversion to the original program and explicitly construct closures and changes to closures. While the only valid change for closed functions is their nil change, for closures we can have non-nil function changes. A function change df, represented as a closure change, is nil exactly when all changes it closes over are nil.

We represent closed functions and closures as variants of the same type. Correspondingly we represent changes to a closed function and changes to a closure as variants of the same type of function changes. We inspect this representation at runtime to find out if a function change is a nil change.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Fun}\ a\ b\ c\ \textbf{where}\\
&\quad \mathit{Closed} :: (a \to (b, c)) \to \mathit{Fun}\ a\ b\ c\\
&\quad \mathit{Closure} :: (e \to a \to (b, c)) \to e \to \mathit{Fun}\ a\ b\ c\\
&\textbf{data}\ \Delta(\mathit{Fun}\ a\ b\ c)\ \textbf{where}\\
&\quad \mathit{DClosed} :: (a \to \Delta a \to c \to (\Delta b, c)) \to \Delta(\mathit{Fun}\ a\ b\ c)\\
&\quad \mathit{DClosure} :: (e \to \Delta e \to a \to \Delta a \to c \to (\Delta b, c)) \to e \to \Delta e \to \Delta(\mathit{Fun}\ a\ b\ c)
\end{aligned}
$$

We use the same benchmark setup as in the benchmark for the average computation on bags. The input of size n is a pair of sequences (xs, ys). Each sequence

initially contains the integers from 1 to n. Updating the result in reaction to a change *dxs* to the outer sequence *xs* takes less time than updating the result in reaction to a change *dys* to the inner sequence *ys*. While a change to the outer sequence *xs* results in an easily located change in the output sequence, a change for the inner sequence *ys* results in a change that needs a lot more calculation to find the elements it affects. We benchmark changes to the outer sequence *xs* and the inner sequence *ys* separately where the change to one sequence is the insertion of a single integer 1 at position 1 and the change for the other one is the nil change.



(a) Benchmark results for Cartesian product changing *inner* sequence.

(b) Benchmark results for Cartesian product changing *outer* sequence.

Fig. 9. Benchmark results for *cartesianProduct*

Figure 9 shows execution time versus input size. In this example again preparing the cache takes longer than from scratch recomputation alone. The speedup of incremental computation over from scratch recomputation increases with the size of the base input sequences because of the difference in time complexity. Eventually we do get speedups for both kinds of changes (to the inner and to the outer sequence), but for changes to the outer sequence we get a speedup earlier, at a smaller input size. The size of the cache grows super linearly in this example.

4.3 Indexed Joins of Two Bags

Our goal is to show that we can compose primitive functions into larger and more complex programs and apply CTS differentiation to get a fast incremental program. We use an example inspired from the DBToaster literature [17]. In this example we have a bag of orders and a bag of line items. An order is a pair of an order key and an exchange rate. A line item is a pair of an order key and a price. We build an index mapping each order key to the sum of all exchange rates of the orders with this key and an index from order key to the sum of the prices of all line items with this key. We then merge the two maps by key, multiplying corresponding sums of exchange rates and sums of prices. We compute the total price of the orders and line items as the sum of those products.

```
type Order = (ℤ, ℤ)
type LineItem = (ℤ, ℤ)

totalPrice :: Bag Order → Bag LineItem → ℤ
totalPrice orders lineItems = let
  orderIndex = groupBy fst orders
  orderSumIndex = Map.map (Bag.foldMapGroup snd) orderIndex
  lineItemIndex = groupBy fst lineItems
  lineItemSumIndex = Map.map (Bag.foldMapGroup snd) lineItemIndex
  merged = Map.merge orderSumIndex lineItemSumIndex
  total = Map.foldMapGroup multiply merged
  in total

groupBy :: (a → k) → Bag a → Map k (Bag a)
groupBy keyOf bag =
  Bag.foldMapGroup (λa → Map.singleton (keyOf a) (Bag.singleton a)) bag
```

Unlike DBToaster, we assume our program is already transformed to explicitly use indexes, as above. Because our indexes are maps, we implemented a change structure, CTS primitives and their CTS derivatives for maps.

To build the indexes, we use a *groupBy* function built from primitive functions *foldMapGroup* on bags and *singleton* for bags and maps respectively. The CTS function *groupByC* and the CTS derivative *dgroupByC* are automatically generated. While computing the indexes with *groupBy* is self-maintainable, merging them is not. We need to cache and incrementally update the intermediately created indexes to avoid recomputing them.

We evaluate the performance in the same way we did in the other case studies. The input of size n is a pair of bags where both contain the pairs (i, i) for i between 1 and n. The change is an insertion of the order $(1, 1)$ into the orders bag. For sufficiently large inputs, our CTS derivative of the original program produces updated results much faster than from scratch recomputation, again because of a difference in time complexity as indicated by Fig. 8b. The size of the cache grows linearly with the size of the input in this example. This is unavoidable, because we need to keep the indexes.

4.4 Limitations and Future Work

Typing of CTS programs. Functions of the same type $f_1, f_2 :: A \to B$ can be transformed to CTS functions $f_1 :: A \to (B, C_1), f_2 :: A \to (B, C_2)$ with different cache types C_1, C_2, since cache types depend on the implementation. This heterogeneous typing of translated functions poses difficult typing issues, e.g. what is the translated type of a *list* $(A \to B)$? We cannot hide cache types behind existential quantifiers because they would be too abstract for derivatives, which only work on very specific cache types. We can fix this problem with some runtime overhead by using a single type *Cache*, defined as a tagged union of all cache types or, maybe with more sophisticated type systems—like first-class translucent sums, open existentials or Typed Adapton's refinement types [12]—that could be able to correctly track down cache types properly.

In any case, we believe that these machineries would add a lot of complexity without helping much with the proof of correctness. Indeed, the simulation relation is more handy here because it maintains a global invariant about the whole evaluations (typically the consistency of cache types between base computations and derivatives), not many local invariants about values as types would.

One might wonder why caches could not be totally hidden from the programmer by embedding them in the derivatives themselves; or in other words, why we did not simply translate functions of type $A \to B$ into functions of type $A \to B \times (\Delta A \to \Delta B)$. We tried this as well; but unlike automatic differentiation, we must remember and update caches according to input changes (especially when receiving a sequence of such changes as in Sect. 2.1). Returning the updated cache to the caller works; we tried closing over the caches in the derivative, but this ultimately fails (because we could receive function changes to the original function, but those would need access to such caches).

Comprehensive performance evaluation. This paper focuses on theory and we leave benchmarking in comparison to other implementations of incremental computation to future work. The examples in our case study were rather simple (except perhaps for the indexed join). Nevertheless, the results were encouraging and we expect them to carry over to more complex examples, but not to all programs. A comparison to other work would also include a comparison of space usage for auxiliary data structure, in our case the caches.

Cache pruning via absence analysis. To reduce memory usage and runtime overhead, it should be possible to automatically remove from transformed programs any caches or cache fragments that are not used (directly or indirectly) to compute outputs. Liu [19] performs this transformation on CTS programs by using *absence analysis*, which was later extended to higher-order languages by Sergey et al. [25]. In lazy languages, absence analysis removes thunks that are not needed to compute the output. We conjecture that the analysis could remove unused caches or inputs, if it is extended to *not* treat caches as part of the output.

Unary vs n-ary abstraction. We only show our transformation correct for unary functions and tuples. But many languages provide efficient support for applying curried functions such as $div :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$. Naively transforming such a curried function to CTS would produce a function $divC$ of type $\mathbb{Z} \to (\mathbb{Z} \to (\mathbb{Z}, DivC_2)), DivC_1$ with $DivC_1 = ()$, which adds excessive overhead. In Sect. 2 and our evaluation we use curried functions and never need to use this naive encoding, but only because we always invoke functions of known arity.

5 Related Work

Cache-transfer-style. Liu [19]'s work has been the fundamental inspiration to this work, but her approach has no correctness proof and is restricted to a first-order untyped language. Moreover, while the idea of cache-transfer-style is similar,

it's unclear if her approach to incrementalization would extend to higher-order programs. Firsov and Jeltsch [9] also approach incrementalization by code transformation, but their approach does not deal with changes to functions. Instead of transforming functions written in terms of primitives, they provide combinators to write CTS functions and derivatives together. On the other hand, they extend their approach to support mutable caches, while restricting to immutable ones as we do might lead to a logarithmic slowdown.

Finite differencing. Incremental computation on collections or databases by finite differencing has a long tradition [6,22]. The most recent and impressive line of work is the one on DBToaster [16,17], which is a highly efficient approach to incrementalize queries over bags by combining iterated finite differencing with other program transformations. They show asymptotic speedups both in theory and through experimental evaluations. Changes are only allowed for datatypes that form groups (such as bags or certain maps), but not for instance for lists or sets. Similar ideas were recently extended to higher-order and nested computation [18], though only for datatypes that can be turned into groups. Koch et al. [18] emphasize that iterated differentiation is necessary to obtain efficient derivatives; however, ANF conversion and remembering intermediate results appear to address the same problem, similarly to the field of automatic differentiation [27].

Logical relations. To study correctness of incremental programs we use a logical relation among base values v_1, updated values v_2 and changes dv. To define a logical relation for an untyped λ-calculus we use a *step-indexed* logical relation, following Ahmed [4], Appel and McAllester [5]; in particular, our definitions are closest to the ones by Acar et al. [3], who also work with an untyped language, big-step semantics and (a different form of) incremental computation. However, they do not consider first-class changes. Technically, we use environments rather than substitution, and index our big-step semantics differently.

Dynamic incrementalization. The approaches to incremental computation with the widest applicability are in the family of self-adjusting computation [1,2], including its descendant Adapton [14]. These approaches incrementalize programs by combining memoization and change propagation: after creating a trace of base computations, updated inputs are compared with old ones in $O(1)$ to find corresponding outputs, which are updated to account for input modifications. Compared to self-adjusting computation, Adapton only updates results that are demanded. As usual, incrementalization is not efficient on arbitrary programs, but only on programs designed so that input changes produce small changes to the computation trace; refinement type systems have been designed to assist in this task [8,12]. To identify matching inputs, Nominal Adapton [13] replaces input comparisons by pointer equality with first-class labels, enabling more reuse.

6 Conclusion

We have presented a program transformation which turns a functional program into its derivative and efficiently shares redundant computations between them thanks to a statically computed cache.

Although our first practical case studies show promising results, this paper focused on putting CTS differentiation on solid theoretical ground. For the moment, we only have scratched the surface of the incrementalization opportunities opened by CTS primitives and their CTS derivatives: in our opinion, exploring the design space for cache data structures will lead to interesting new results in purely functional incremental programming.

Acknowledgments. We are grateful to anonymous reviewers: they made important suggestions to help us improve our technical presentation. We also thank Cai Yufei, Tillmann Rendel, Lourdes del Carmen González Huesca, Klaus Ostermann, Sebastian Erdweg for helpful discussions on this project. This work was partially supported by DFG project 282458149 and by SNF grant No. 200021_166154.

References

1. Acar, U.A.: Self-adjusting computation. Ph.D. thesis, Carnegie Mellon University (2005)
2. Acar, U.A.: Self-adjusting computation: (an overview). In: PEPM, pp. 1–6. ACM (2009)
3. Acar, U.A., Ahmed, A., Blume, M.: Imperative self-adjusting computation. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 309–322. ACM, New York (2008). https://doi.acm.org/10.1145/1328438.1328476
4. Ahmed, A.: Step-indexed syntactic logical relations for recursive and quantified types. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 69–83. Springer, Heidelberg (2006). https://doi.org/10.1007/11693024_6
5. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. **23**(5), 657–683 (2001). https://doi.acm.org/10.1145/504709.504712
6. Blakeley, J.A., Larson, P.A., Tompa, F.W.: Efficiently updating materialized views. In: SIGMOD, pp. 61–71. ACM (1986)
7. Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K.: A theory of changes for higher-order languages—incrementalizing λ-calculi by static differentiation. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 145–155. ACM, New York (2014). https://doi.acm.org/10.1145/2594291.2594304
8. Çiçek, E., Paraskevopoulou, Z., Garg, D.: A type theory for incremental computational complexity with control flow changes. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pp. 132–145. ACM, New York (2016)

9. Firsov, D., Jeltsch, W.: Purely functional incremental computing. In: Castor, F., Liu, Y.D. (eds.) SBLP 2016. LNCS, vol. 9889, pp. 62–77. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45279-1_5
10. Giarrusso, P.G.: Optimizing and incrementalizing higher-order collection queries by AST transformation. Ph.D. thesis, University of Tübingen (2018). Defended. http://inc-lc.github.io/
11. Gupta, A., Mumick, I.S.: Maintenance of materialized views: problems, techniques, and applications. In: Gupta, A., Mumick, I.S. (eds.) Materialized Views, pp. 145–157. MIT Press (1999)
12. Hammer, M.A., Dunfield, J., Economou, D.J., Narasimhamurthy, M.: Typed adapton: refinement types for incremental computations with precise names. October 2016 arXiv:1610.00097 [cs]
13. Hammer, M.A., et al.: Incremental computation with names. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, pp. 748–766. ACM, New York (2015). https://doi.acm.org/10.1145/2814270.2814305
14. Hammer, M.A., Phang, K.Y., Hicks, M., Foster, J.S.: Adapton: composable, demand-driven incremental computation. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 156–166. ACM, New York (2014)
15. Johnsson, T.: Lambda lifting: transforming programs to recursive equations. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 190–203. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-15975-4_37
16. Koch, C.: Incremental query evaluation in a ring of databases. In: Symposium Principles of Database Systems (PODS), pp. 87–98. ACM (2010)
17. Koch, C., et al.: DBToaster: higher-order delta processing for dynamic, frequently fresh views. VLDB J. **23**(2), 253–278 (2014). https://doi.org/10.1007/s00778-013-0348-4
18. Koch, C., Lupei, D., Tannen, V.: Incremental view maintenance for collection programming. In: Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, pp. 75–90. ACM, New York (2016)
19. Liu, Y.A.: Efficiency by incrementalization: an introduction. HOSC **13**(4), 289–313 (2000)
20. Liu, Y.A., Teitelbaum, T.: Caching intermediate results for program improvement. In: Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 1995, pp. 190–201. ACM, New York (1995). https://doi.acm.org/10.1145/215465.215590
21. O'Sullivan, B.: criterion: a Haskell microbenchmarking library (2014). http://www.serpentine.com/criterion/
22. Paige, R., Koenig, S.: Finite differencing of computable expressions. TOPLAS **4**(3), 402–454 (1982)
23. Rossberg, A., Russo, C.V., Dreyer, D.: F-ing modules. In: Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI 2010, pp. 89–102. ACM, New York (2010)
24. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. LISP Symb. Comput. **6**(3–4), 289–360 (1993)
25. Sergey, I., Vytiniotis, D., Peyton Jones, S.: Modular, higher-order cardinality analysis in theory and practice. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, pp. 335–347. ACM, New York (2014)

26. The Coq Development Team: The Coq proof assistant reference manual, version 8.8 (2018). http://coq.inria.fr
27. Wang, F., Wu, X., Essertel, G., Decker, J., Rompf, T.: Demystifying differentiable programming: shift/reset the penultimate backpropagator. Technical report (2018). https://arxiv.org/abs/1803.10228

Concurrency and Distribution

Asynchronous Timed Session Types
From Duality to Time-Sensitive Processes

Laura Bocchi[1(✉)], Maurizio Murgia[1,4], Vasco Thudichum Vasconcelos[2], and Nobuko Yoshida[3]

[1] University of Kent, Canterbury, UK
l.bocchi@kent.ac.uk
[2] LASIGE, Faculty of Sciences, University of Lisbon, Lisbon, Portugal
[3] Imperial College London, London, UK
[4] University of Cagliari, Cagliari, Italy

Abstract. We present a behavioural typing system for a higher-order timed calculus using session types to model timed protocols. Behavioural typing ensures that processes in the calculus perform actions in the time-windows prescribed by their protocols. We introduce duality and subtyping for timed asynchronous session types. Our notion of duality allows typing a larger class of processes with respect to previous proposals. Subtyping is critical for the precision of our typing system, especially in the presence of session delegation. The composition of dual (timed asynchronous) types enjoys progress when using an urgent receive semantics, in which receive actions are executed as soon as the expected message is available. Our calculus increases the modelling power of extant calculi on timed sessions, adding a blocking receive primitive with timeout and a primitive that consumes an arbitrary amount of time in a given range.

Keywords: Session types · Timers · Duality · π-calculus

1 Introduction

Time is at the basis of many real-life protocols. These include common client-server interactions as for example, *"An SMTP server SHOULD have a timeout of at least 5 minutes while it is awaiting the next command from the sender"* [22]. By protocol, we intend application-level specifications of interaction patterns (via message passing) among distributed applications. An extensive literature offers theories and tools for formal analysis of timed protocols, modelled for instance as timed automata [3,26,34] or Message Sequence Charts [2]. These works allow to reason on the properties of *protocols*, defined as formal models. Recent work,

This work has been partially supported by EPSRC EP/N035372/1, EP/K011715/1, EP/N027833/1, EP/K034413/1, EP/L00058X/1, EP/N028201/1, Aut. Reg. of Sardinia projects *Sardcoin* and *Smart collaborative engineering*, FCT through project Confident PTDC/EEI-CTP/4503/2014 and the LASIGE Research Unit UID/CEC/00408/2019. We thank Julien Lange for his advise and comments.

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 583–610, 2019.
https://doi.org/10.1007/978-3-030-17184-1_21

based on session types, focus on the relationship between time-sensitive protocols, modelled as timed extensions of session types, and their implementations abstracted as *processes* in some timed calculus. The relationship between protocols and processes is given in terms of static behavioural typing [12,15] or run-time monitoring [6,7,30] of processes against types. Existing work on timed session types [7,12,15,30] is based on simple abstractions for processes which do not capture time sensitive primitives such as blocking (as well as non-blocking) receive primitives with timeout and time consuming actions with variable, yet bound, duration. This paper provides a theory of asynchronous timed session types for a calculus that features these two primitives. We focus on the asynchronous scenario, as modern distributed systems (e.g., web) are often based on asynchronous communications via FIFO channels [4,33]. The link between protocols and processes is given in terms of static behavioural typing, checking for punctuality of interactions with respect to protocols prescriptions. Unlike previous work on asynchronous timed session types [12], our type system can check processes against protocols that are *not wait-free*. In wait-free protocols, the time-windows for corresponding send and receive actions have an empty intersection. We illustrate wait-freedom using a protocol modelled as two timed session types, each owning a set of clocks (with no shared clocks between types).

$$S_{\mathtt{c}} = !\mathtt{Command}(x < 5, \{x\}).S'_{\mathtt{c}} \qquad S_{\mathtt{s}} = ?\mathtt{Command}(\mathtt{y} < 5, \{y\}).S'_{\mathtt{s}} \tag{1}$$

The protocol in (1) involves a client $S_{\mathtt{c}}$ with a clock x, and a server $S_{\mathtt{s}}$ with a clock y (with both x and y initially set to 0). Following the protocol, the client must send a message of type `Command` within 5 min, reset x, and continue as $S'_{\mathtt{c}}$. Dually, the server must be ready to receive a command with a timeout of 5 min, reset y, and continue as $S'_{\mathtt{s}}$. The model in (1) is *not wait-free*: the intersection of the time-windows for the send and receive actions is non-empty (the time-windows actually coincide). The protocol in (2), where the server must wait until after the client's deadline to read the message, is wait-free.

$$!\mathtt{Command}(x < 5, \{x\}).S''_{\mathtt{c}} \qquad ?\mathtt{Command}(\mathtt{y} = 5, \{y\}).S''_{\mathtt{s}} \tag{2}$$

Patterns like the one in (1) are common (e.g., the SMPT fragment mentioned at the beginning of this introduction) but, unfortunately, they are *not wait-free*, hence ruled out in previous work [12]. Arguably, (2) is an unpractical wait-free variant of (1): the client must always wait for at least 5 min to have the message read, no matter how early this message was sent. The definition of protocols for our typing system (which allows for *not wait-free* protocols) is based on a notion of *asynchronous timed duality*, and on a subtyping relation that provides accuracy of typing, especially in the case of channel passing.

Asynchronous timed duality. In the untimed scenario, each session type has one unique *dual* that is obtained by changing the polarities of the actions (send vs. receive, and selection vs. branching). For example, the dual of a session type S

that sends an integer and then receives a string is a session type $\overline{S}$ that receives an integer and then sends a string.

$$S = !\texttt{Int}.?\texttt{String} \qquad \overline{S} = ?\texttt{Int}.!\texttt{String}$$

Duality characterises well-behaved systems: the behaviour described by the composition of dual types has no communication mismatches (e.g., unexpected messages, or messages with values of unexpected types) nor deadlocks. In the timed scenario, this is no longer true. Consider a timed extension of session types (using the model of time in timed automata [3]), and of (untimed) duality so that dual send/receive actions have equivalent time constraints and resets. The example below shows a timed type S with its dual $\overline{S}$, where S owns clock x, and $\overline{S}$ owns clock y (with x and y initially set to 0):

$$S = !\texttt{Int}(x \leqslant 1, x).?\texttt{String}(x \leqslant 2) \qquad \overline{S} = ?\texttt{Int}(y \leqslant 1, y).!\texttt{String}(y \leqslant 2)$$

Here S sends an integer at any time satisfying $x \leqslant 1$, and then resets x. After that, S receives a string at any time satisfying $x \leqslant 2$. The timed dual of S is obtained by keeping the same time constraints (and renaming the clock—to make it clear that clocks are not shared). To illustrate our point, we use the semantics from timed session types [12], borrowed from Communicating Timed automata [23]. This semantics is *separated*, in the sense that only time actions may 'take time', while all other actions (e.g., communications) are instantaneous.[1] The aforementioned semantics allows for the following execution of $S \mid \overline{S}$:

$$\begin{aligned}
S \mid \overline{S} &\xrightarrow{0.4}\xrightarrow{\texttt{Int}} ?\texttt{String}(x \leqslant 2) \mid \overline{S} && \text{(clocks values: } x = 0,\ y = 0.4) \\
&\xrightarrow{0.6}\xrightarrow{\texttt{Int}} ?\texttt{String}(x \leqslant 2) \mid !\texttt{String}(x \leqslant 2) && \text{(clocks values: } x = 0.6,\ y = 0) \\
&\xrightarrow{2}\xrightarrow{!\texttt{String}} ?\texttt{String}(x \leqslant 2) && \text{(clocks values: } x = 2.6,\ y = 2)
\end{aligned}$$

where: (i) the system makes a time step of 0.4, then S sends the integer and resets x, yielding a state where $x = 0$ and $y = 0.4$; (ii) the system makes a time step of 0.6, then $\overline{S}$ receives the integer and resets y, yielding a state where $x = 0.6$ and $y = 0$; (iii) the system makes a time step of 2, then the continuation of $\overline{S}$ sends the string, when $y = 2$ and $x = 2.6$. In (iii), the string was sent too late: constraint $x \leqslant 2$ of the receiving endpoint is now unsatisfiable. The system cannot do any further legal step, and is stuck.

Urgent receive semantics. The example above shows that, in the timed asynchronous scenario, the straightforward extension of duality to the timed scenario does not necessarily characterise well-behaved communications. We argue, however, that the execution of $S \mid \overline{S}$, in particular the time reduction with label 0.6, does not reflect the semantics of most common receive primitives. In fact, most mainstream programming languages implement *urgent receive* semantics

[1] Separated semantics can describe situations where actions have an associated duration.

for receive actions. We call a semantics *urgent receive* when receive actions are executed as soon as the expected message is available, given that the guard of that action is satisfied. Conversely, *non-urgent receive* semantics allows receive actions to fire at any time satisfying the time constraint, as long as the message is in the queue. The aforementioned reduction with label 0.6 is permitted by non-urgent receive semantics such as the one in [23], since it defers the reception of the integer despite the integer being ready for reception and the guard ($y \leqslant 2$) being satisfied, but not by urgent receive semantics. Urgent receive semantics allows, instead, the following execution for $S \mid \overline{S}$:

$$
\begin{array}{lll}
S \mid \overline{S} \xrightarrow{0.4} \xrightarrow{!\mathtt{int}} & ?\mathtt{String}(x \leqslant 2) \mid \overline{S} & \text{(clocks values: } x = 0,\ y = 0.4) \\
\xrightarrow{?\mathtt{int}} & ?\mathtt{String}(x \leqslant 2) \mid !\mathtt{String}(x \leqslant 2) & \text{(clocks values: } x = 0,\ y = 0) \\
\xrightarrow{2} \xrightarrow{!\mathtt{String}} & ?\mathtt{String}(x \leqslant 2) & \text{(clocks values: } x = 2,\ y = 2)
\end{array}
$$

If S sends the integer when $x = 0.4$, then $\overline{S}$ must receive the integer immediately, when $y = 0.4$. At this point, both endpoints reset their respective clocks, and the communication will continue in sync. Urgent receive primitives are common; some examples are the non-blocking `WaitFreeReadQueue.read()` and blocking `WaitFreeReadQueue.waitForData()` of Real-Time Java [13], and the receive primitives in Erlang and Golang. *Urgent receive semantics make interactions "more synchronous" but still as asynchronous as real-life programs.*

A calculus for timed asynchronous processes. Our calculus features two time-sensitive primitives. The first is a parametric receive operation $a^n(b).\,P$ on a channel a, with a timeout n that can be ∞ or any number in $\mathbf{R}_{\geqslant 0}$. The parametric receive captures a range of receive primitives: non-blocking ($n = 0$), blocking without timeout ($n = \infty$), or blocking with timeout ($n \in \mathbf{R}_{>0}$). The second primitive is a time-consuming action, $\mathtt{delay}(\delta).\,P$, where δ is a constraint expressing the time-window for the time consumed by that action. Delay processes model primitives like `Thread.sleep`(n) in real-time Java [13] or, more generally, any time-consuming action, with δ being an estimation of the delay of computation.

Processes in our calculus abstract implementations of protocols given as pairs of dual types. Consider the processes below.

$$
P_C = \mathtt{delay}(x < 3).\,\overline{a}\,\mathtt{HELO}.P'_C \quad P_S = \mathtt{delay}(x = 5).\,a^0(b).P'_S \quad Q_S = a^5(b).Q'_S
$$

Processes abiding the protocols in (2) could be as follows: P_C for the client S_C, and P_S for the server S_S. The client process P_C performs a time consuming action for up to 3 min, then sends command `HELO` to the server, and continues as P'_C. The server process P_S sleeps for exactly 5 min, receives the message immediately (without blocking), and continues as P'_S. A process for the protocol in (1) could, instead be the parallel composition of P_C, again for the client, and Q_S for the server. Process Q_S uses a blocking primitive with timeout; the server now blocks on the receive action with a timeout of 5 min, and continues as Q'_S as soon as a message is received. The blocking receive primitive with timeout is crucial

to model processes typed against protocols one can express with asynchronous timed duality, in particular those that are not wait-free.

A type system for timed asynchronous processes. The relationship between types and processes in our calculus is given as a typing system. Well-typed processes are ensured to communicate at the times prescribed by their types. This result is given via Subject Reduction (Theorem 4), establishing that well-typedness is preserved by reduction. In our timed scenario, Subject Reduction holds under *receive liveness*, an assumption on the interaction structure of processes. This assumption is orthogonal to time. To characterise the interaction structures of a timed process we erase timing information from that processes (*time erasure*). Receive liveness requires that, whenever a time-erased processes is waiting for a message, the corresponding message is eventually provided by the rest of the system. While receive liveness is not needed for Subject Reduction in untimed systems [21], it is required for timed processes. This reflects the natural intuition that if an untimed-process violates progress, then its timed counterpart may miss deadlines. Notably, we can rely on existing behavioural checking techniques from the untimed setting to ensure receive liveness [17].

Receive liveness is not required for Subject Reduction in a related work on asynchronous timed session types [12]. The dissimilarity in the assumptions is only apparent; it derives from differences in the two semantics for processes. When our processes cannot proceed correctly (e.g., in case of missed deadlines) they reduce to a failed state, whereas the processes in [12] become stuck (indicating violation of progress).

Synopsis. In Sect. 2 we introduce the syntax and the formation rules for asynchronous timed session types. In Sect. 3, we give a modular Labelled Transition System (LTS) for types in isolation (Sect. 3.1) and for compositions of types (Sect. 3.3). The subtyping relation is given in Sect. 3.2 and motivated in Example 8, after introducing the typing rules. We introduce timed asynchronous duality and its properties in Sect. 4. Remarkably, the composition of dual timed asynchronous types enjoys progress when using an urgent receive semantics (Theorem 1). Section 5 presents a calculus for timed processes and Sect. 6 introduces its typing system. The properties of our typing system—Subject Reduction (Theorem 4) and Time Safety (Theorem 5)—are introduced in Sect. 7. Conclusions and related works are in Sect. 8. Proofs and additional material can be found in the online report [11].

2 Asynchronous Timed Session Types

Clocks and predicates. We use the model of time from timed automata [3]. Let $\mathbb{X}$ be a finite set of clocks, let $x_1, \ldots, x_n$ range over clocks, and let each clock take values in $\mathbf{R}_{\geqslant 0}$. Let $t_1, \ldots, t_n$ range over non-negative real numbers and $n_1, \ldots, n_n$ range over non-negative rationals. The set $\mathcal{G}(\mathbb{X})$ of predicates over $\mathbb{X}$ is defined by the following grammar.

$$\delta ::= \mathtt{true} \mid x > n \mid x = n \mid x - y > n \mid x - y = n \mid \neg\delta \mid \delta_1 \wedge \delta_2 \quad \text{where} \quad x, y \in \mathbb{X}$$

We derive `false`, $<$, $\geqslant$, $\leqslant$ in the standard way. Predicates in the form $x - y > n$ and $x - y = n$ are called *diagonal* predicates; in these cases we assume $x \neq y$. Notation $cn(\delta)$ stands for the set of clocks in δ.

Clock valuation and resets. A clock valuation $\nu : \mathbb{X} \mapsto \mathbf{R}_{\geqslant 0}$ returns the time of the clocks in $\mathbb{X}$. We write $\nu + t$ for the valuation mapping all $x \in \mathbb{X}$ to $\nu(x) + t$, ν_0 for the initial valuation (mapping all clocks to 0), and, more generally, ν_t for the valuation mapping all clocks to t. Let $\nu \models \delta$ denote that δ is satisfied by ν. A reset predicate λ over $\mathbb{X}$ is a subset of $\mathbb{X}$. When λ is $\varnothing$ then no reset occurs, otherwise the assignment for each $x \in \lambda$ is set to 0. We write $\nu\,[\lambda \mapsto 0]$ for the clock assignment that is like ν everywhere except that its assigns 0 to all clocks in λ.

Types. Timed session types, hereafter just types, have the following syntax:

$$T ::= (\delta, S) \mid \texttt{Nat} \mid \texttt{Bool} \mid \dots$$

$$S ::= !T(\delta,\lambda).S \mid ?T(\delta,\lambda).S \mid \oplus\{\mathrm{l_i}(\delta_i,\lambda_i) : S_i\}_{i\in I} \mid \&\{\mathrm{l_i}(\delta_i,\lambda_i) : S_i\}_{i\in I} \mid \mu\alpha.S \mid \alpha \mid \texttt{end}$$

Sorts T include base types (`Nat`, `Bool`, etc.), and sessions (δ, S). Messages of type (δ, S) allow a participant involved in a session to delegate the remaining behaviour S; upon delegation the sender will no longer participate in the delegated session and receiver will execute the protocol described by S under any clock assignment satisfying δ. We denote the set of types with $\mathbb{T}$.

Type $!T(\delta,\lambda).S$ models a *send action* of a payload with sort T. The sending action is allowed at any time that satisfies the guard δ. The clocks in λ are reset upon sending. Type $?T(\delta,\lambda).S$ models the dual *receive action* of a payload with sort T. The receiving types require the endpoint to be ready to receive the message in the precise time window specified by the guard.

Type $\oplus\{\mathrm{l_i}(\delta_i,\lambda_i) : S_i\}_{i\in I}$ is a *select action*: the party chooses a branch $i \in I$, where I is a finite set of indices, selects the label l_i, and continues as prescribed by S_i. Each branch is annotated with a guard δ and reset λ. A branch j can be selected at any time allowed by δ_j. The dual type is $\&\{\mathrm{l_i}(\delta_i,\lambda_i) : S_i\}_{i\in I}$ for *branching actions*. Each branch is annotated with a guard and a reset. The endpoint must be ready to receive the label for j at any time allowed by δ_j (or until another branch is selected).

Recursive type $\mu\alpha.S$ associates a *type variable* α to a recursion body S. We assume that type variables are guarded in the standard way (i.e., they only occur under actions or branches). We let $\mathcal{A}$ denote the set of type variables.

Type `end` models successful termination.

2.1 Type Formation

The grammar for types allow to generate types that are not implementable in practice, as the one shown in Example 1.

Example 1 (Junk-types). Consider S in (3) under initial clock valuation ν_0.

$$S = ?T(x < 5, \varnothing).!T(x < 2, \varnothing).\texttt{end} \tag{3}$$

The specified endpoint must be ready to receive a message in the time-window between 0 and 5 time units, as we evaluate $x < 5$ in ν_0. Assume that this receive action happens when $x = 3$, yielding a new state in which: (i) the clock valuation maps x to 3, and (ii) the endpoint must perform a send action while $x < 2$. Evidently, (ii) is no longer possible in the new clock valuation, as the $x < 2$ is now unsatisfiable. We could amend (3) in several ways: (a) by resetting x after the receive action; (b) by restricting the guard of the receive action (e.g., $x < 2$ instead of $x < 5$); or (c) by relaxing the guard of the send action. All these amendments would, however, yield a different type.

In the remainder of this section we introduce formation rules to rule out junk types as the one in Example 1 and characterise types that are well-formed. Intuitively, well-formed types allow, at any point, to perform some action in the present time or at some point in the future, unless the type is `end`.

Judgments. The formation rules for types are defined on judgments of the form

$$A;\ \delta \vdash S$$

where A is an environment assigning type variables to guards, and δ is a guard in $\mathcal{G}(\mathbb{X})$. A is used as an invariant to form recursive types. Guard δ collects the possible 'pasts' from which the next action in S could be executed (unless S is `end`). We use notation $\downarrow \delta$ (the past of δ) for a guard δ' such that $\nu \models \delta'$ if and only if $\exists t : \nu + t \models \delta$. For example, $\downarrow (1 \leqslant x \leqslant 2) = x \leqslant 2$ and $\downarrow (x \geqslant 3) = \texttt{true}$. Similarly, we use the notation $\delta[\lambda \mapsto 0]$ to denote a guard in which all clocks in λ are reset. For example, $(x \leqslant 3 \wedge y \leqslant 2)[x \mapsto 0] = (x = 0 \wedge y \leqslant 2)$. We use the notation $\delta_1 \subseteq \delta_2$ whenever, for all ν, $\nu \models \delta_1 \implies \nu \models \delta_2$. The past and reset of a guard can be inferred algorithmically, and $\subseteq$ is decidable [8].

$$\frac{}{A;\ \texttt{true} \vdash \texttt{end}}\ [\texttt{end}]$$

$$\frac{\square \in \{!, ?\} \qquad A;\ \gamma \vdash S \qquad \delta[\lambda \mapsto 0] \subseteq \gamma \qquad T \text{ base type}}{A;\ \downarrow \delta \vdash \square\, T(\delta, \lambda).S}\ [\texttt{interact}]$$

$$\frac{\square \in \{!, ?\} \qquad A;\ \gamma \vdash S \qquad \delta[\lambda \mapsto 0] \subseteq \gamma \quad T = (\delta', S') \qquad \varnothing;\ \gamma' \vdash S' \quad \delta' \subseteq \gamma'}{A;\ \downarrow \delta \vdash \square\, T(\delta, \lambda).S}\ [\texttt{delegate}]$$

$$\frac{\square \in \{\oplus, \&\} \qquad \forall i \in I \quad A;\ \gamma_i \vdash S_i \quad \delta_i[\lambda_i \mapsto 0] \subseteq \gamma_i}{A;\ \downarrow \bigvee_{i \in I} \delta_i \vdash \square\, \{\mathrm{l}_i(\delta_i, \lambda_i) : S_i\}_{i \in I}}\ [\texttt{choice}]$$

$$\frac{A, \alpha : \delta;\ \delta \vdash S}{A;\ \delta \vdash \mu\alpha.S}\ [\texttt{rec}] \qquad \frac{}{A, \alpha : \delta;\ \delta \vdash \alpha}\ [\texttt{var}]$$

Rule [end] states that the terminated type is well-formed against any A. The guard of the judgement is `true` since `end` is a final state (as `end` has no continuation, morally, the constraint of its continuation is always satisfiable). Rule [interact] ensures that the past of the current action δ entails the past of the subsequent action γ (considering resets if necessary): this rules out types in which the subsequent action can only be performed in the past. Rules [end] and [interact] are illustrated by the three examples below.

Example 2. The judgment below shows a type being *discarded* after an application of rule [interact] :

$$\varnothing;\ x \leqslant 3 \nvdash\ ?\mathtt{Nat}(1 \leqslant x \leqslant 3, \varnothing).!\mathtt{Nat}(1 \leqslant x \leqslant 2, \varnothing).\mathtt{end} \tag{4}$$

The premise of [interact] would be $\delta \nsubseteq \downarrow \gamma$, which does not hold for $\delta = 1 \leqslant x \leqslant 3$ and $\downarrow \gamma = x \leqslant 2$. This means that guard $(1 \leqslant x \leqslant 3, \varnothing)$ of the first action may lead to a state in which guard $1 \leqslant x \leqslant 2$ for the subsequent action is unsatisfiable. If we amend the type in (4) by adding a reset in the first action, we obtain a well-formed type. We show its formation below, where for simplicity we omit obvious preconditions like `Nat` base type, etc.

$$\dfrac{\dfrac{\dfrac{}{\varnothing;\ \mathtt{true} \vdash \mathtt{end}}\ [\mathtt{end}] \qquad 1 \leqslant x \leqslant 2 \subseteq \mathtt{true}}{\varnothing;\ x \leqslant 2 \vdash !\mathtt{Nat}(1 \leqslant x \leqslant 2, \varnothing).\mathtt{end} \qquad x = 0 \subseteq x \leqslant 2}\ [\mathtt{interact}]}{\varnothing;\ x \leqslant 3 \vdash ?\mathtt{Nat}(1 \leqslant x \leqslant 3, \{x\}).!\mathtt{Nat}(1 \leqslant x \leqslant 2, \varnothing).\mathtt{end}}\ [\mathtt{interact}]$$

Rule [delegate] behaves as [interact] , with two additional premises on the delegated session: (1) S' needs to be well-formed, and (2) the guard of the next action in S' needs to be satisfiable with respect to δ'. Guard δ' is used to ensure a correspondence between the state of the delegating endpoint and that of the receiving endpoint. Rule [choice] is similar to [interact] but requires that there is at least one viable branch (this is accomplished by considering the weaker past $\downarrow \bigvee_{i \in I} \delta_i$) and checking each branch for formation. Rules [rec] and [var] are for recursive types and variables, respectively. In [rec] the guard δ can be easily computed by taking the past of the next action of the in S (or the disjunction if S is a branching or selection). An algorithm for deciding type formation can be found in [11].

Definition 1 (Well-formed types). *We say that S is well-formed against clock valuation ν if $\varnothing;\ \delta \vdash S$ and $\nu \models \delta$, for some guard δ. We say that S is well-formed if it is well formed against ν_0.*

We will tacitly assume types are well-formed, unless otherwise specified. The intuition of well-formedness is that if $A;\ \delta \vdash S$ then S can be run (using the types semantics given in Sect. 3) under any clock valuation ν such that $\nu \models \delta$. In the sequel, we take (well-formed) types equi-recursively [31].

3 Asynchronous Session Types Semantics and Subtyping

We give a compositional semantics of types. First, we focus on types in isolation from their environment and from their queues, which we call *simple type configurations*. Next we define subtyping for simple type configurations. Finally, we consider systems (i.e., composition of types communicating via queues).

$$\frac{\nu \models \delta}{(\nu, !T(\delta,\lambda)).S \xrightarrow{!T} (\nu\,[\lambda \mapsto 0], S)}\ [\text{snd}] \qquad \frac{\nu \models \delta}{(\nu, ?T(\delta,\lambda).S) \xrightarrow{?T} (\nu\,[\lambda \mapsto 0], S)}\ [\text{rcv}]$$

$$\frac{\nu \models \delta_j \qquad j \in I}{(\nu, \oplus\{\mathrm{l_i}(\delta_i,\lambda_i) : S_i\}_{i\in I}) \xrightarrow{!\mathrm{l_j}} (\nu\,[\lambda_j \mapsto 0], S_j)}\ [\text{sel}]$$

$$\frac{\nu \models \delta_j \qquad j \in I}{(\nu, \&\{\mathrm{l_i}(\delta,\lambda_i) : S_i\}_{i\in I}) \xrightarrow{?\mathrm{l_j}} (\nu\,[\lambda_j \mapsto 0], S_j)}\ [\text{bra}]$$

$$\frac{(\nu, S[\mu \mathtt{t}.S/\mathtt{t}]) \xrightarrow{\ell} (\nu',\ S')}{(\nu, \mu\mathtt{t}.S) \xrightarrow{\ell} (\nu', S')}\ [\text{rec}] \qquad (\nu, S) \xrightarrow{t} (\nu + t, S)\ [\text{time}]$$

Fig. 1. LTS for simple type configurations

3.1 Types in Isolation

The behaviour of *simple type configurations* is described by the Labelled Transition System (LTS) on pairs (ν, S) over $(\mathbb{V} \times \mathcal{S})$, where clock valuation ν gives the values of clocks in a specific state. The LTS is defined over the following labels

$$\ell ::= !m \mid ?m \mid t \mid \tau \qquad\qquad m ::= d \mid \mathrm{l}$$

Label $!m$ denotes an output action of message m and $?m$ an input action of m. A message m can be a sort T (that can be either a higher order message (δ, S) or base type), or a branching label l. The LTS for single types is defined as the least relation satisfying the rules in Fig. 1. Rules [snd], [rcv], [sel], and [bra] can only happen if the constraint of the next action is satisfied in the current clock valuation. Rule [rec] unfolds recursive types, and [time] always lets time elapse.

Let $\mathbf{s}$, $\mathbf{s}'$, $\mathbf{s}_i$ $(i \in \mathbb{N})$ range over simple type configurations (ν, S). We write $\mathbf{s} \xrightarrow{\ell}$ when there exists $\mathbf{s}'$ such that $\mathbf{s} \xrightarrow{\ell} \mathbf{s}'$, and write $\mathbf{s} \xrightarrow{t\,\ell}$ for $\mathbf{s} \xrightarrow{t} \xrightarrow{\ell}$.

3.2 Asynchronous Timed Subtyping

We define subtyping as a partial relation on simple type configurations. As in other subtyping relations for session types we consider send and receive actions dually [14,16,19]. Our subtyping relation is covariant on output actions and contra-variant on input actions, similarly to that of [14]. In this way, our subtyping $S <: S'$ captures the intuition that a process well-typed against S can be safely substituted with a process well-typed against S'. Definition 2, introduces a notation that is useful in the rest of this section.

Definition 2 (Future enabled send/receive). *Action ℓ is future enabled in* $\mathbf{s}$ *if* $\exists t : \mathbf{s} \xrightarrow{t\,\ell}$. *We write* $\mathbf{s} \overset{!}{\Rightarrow}$ *(resp.* $\mathbf{s} \overset{?}{\Rightarrow}$*) if there exists a sending action* $!m$ *(resp. a receiving action* $?m$*) that is future enabled in* $\mathbf{s}$.

As common in session types, the communication structure does not allow for mixed choices: the grammar of types enforces choices to be either all input (branching actions), or output (selection actions). From this fact it follows that, given $\mathbf{s}$, reductions $\mathbf{s} \overset{!}{\Rightarrow}$ and $\mathbf{s} \overset{?}{\Rightarrow}$ cannot hold simultaneously.

Definition 3 (Timed Type Simulation). *Fix* $\mathbf{s}_1 = (\nu_1, S_1)$ *and* $\mathbf{s}_2 = (\nu_2, S_2)$. *A relation* $\mathcal{R} \in (\mathbb{V} \times \mathcal{S})^2$ *is a* timed type simulation *if* $(\mathbf{s}_1, \mathbf{s}_2) \in \mathcal{R}$ *implies the following conditions:*

1. $S_1 = \mathtt{end}$ *implies* $S_2 = \mathtt{end}$
2. $\mathbf{s}_1 \xrightarrow{t\,!m_1} \mathbf{s}_1'$ *implies* $\exists \mathbf{s}_2', m_2 : \mathbf{s}_2 \xrightarrow{t\,!m_2} \mathbf{s}_2'$, $(m_2, m_1) \in \mathcal{S}, (\mathbf{s}_1', \mathbf{s}_2') \in \mathcal{R}$
3. $\mathbf{s}_2 \xrightarrow{t\,?m_2} \mathbf{s}_2'$ *implies* $\exists \mathbf{s}_1', m_1 : \mathbf{s}_1 \xrightarrow{t\,?m_1} \mathbf{s}_1'$, $(m_1, m_2) \in \mathcal{S}$, $(\mathbf{s}_1', \mathbf{s}_2') \in \mathcal{R}$
4. $\mathbf{s}_1 \overset{?}{\Rightarrow}$ *implies* $\mathbf{s}_2 \overset{?}{\Rightarrow}$ *and* $\mathbf{s}_2 \overset{!}{\Rightarrow}$ *implies* $\mathbf{s}_1 \overset{!}{\Rightarrow}$

where $\mathcal{S}$ *is the following extension of* $\mathcal{R}$ *to messages: (1)* $(T, T') \in \mathcal{S}$ *if* T *and* T' *are base types, and* T' *is a subtype of* T *by sorts subtyping, e.g.,* $(\mathtt{int}, \mathtt{nat}) \in \mathcal{S}$; *(2)* $(1, 1) \in \mathcal{S}$; *(3)* $((\delta_1, S_1), (\delta_2, S_2)) \in \mathcal{S}$, *if* $\forall \nu_1 \models \delta_1 \exists \nu_2 \models \delta_2 : ((\nu_1, S_1), (\nu_2, S_2)) \in \mathcal{R}$ *and* $\forall \nu_2 \models \delta_2 \exists \nu_1 \models \delta_1 : ((\nu_1, S_1), (\nu_2, S_2)) \in \mathcal{R}$.

Intuitively, if $(\mathbf{s}_1, \mathbf{s}_2) \in \mathcal{R}$ then any environment that can safely interact with $\mathbf{s}_2$, can do so with $\mathbf{s}_1$. We write that $\mathbf{s}_2$ simulates $\mathbf{s}_1$ whenever $\mathbf{s}_1$ and $\mathbf{s}_2$ are in a timed type simulation. Below, $\mathbf{s}_2$ simulates $\mathbf{s}_1$:

$$\mathbf{s}_1 = (\nu_0, !\mathtt{nat}(x < 5, \varnothing).\mathtt{end}) \quad \mathbf{s}_2 = (\nu_0, !\mathtt{int}(x \leqslant 10, \varnothing).\mathtt{end})$$

Conversely, $\mathbf{s}_1$ does not simulate $\mathbf{s}_2$ because of condition (2). Precisely, $\mathbf{s}_2$ can make a transition $\mathbf{s}_2 \xrightarrow{10\,!\mathtt{int}}$ that cannot be matched by $\mathbf{s}_1$ for two reasons: guard $x < 5$ is no longer satisfiable when $x = 10$, and $(\mathtt{nat}, \mathtt{int}) \notin \mathcal{S}$ since `int` is not a subtype of `nat`. For receive actions, instead, we could substitute $\mathbf{s}$ with $\mathbf{s}'$ if $\mathbf{s}'$ had at least the receiving capabilities of $\mathbf{s}$. Condition (4) in Definition 3 rules out relations that include, e.g., $((\nu, ?T(\mathtt{true}, \varnothing).\mathtt{end}), (\nu, !T(\mathtt{true}, \varnothing).\mathtt{end}))$.

Live simple type configurations. In our subtyping definition we are interested in simple type configurations that are not stuck. Consider the example below:

$$(\nu, !\mathtt{Int}(x \leqslant 10, \varnothing).\mathtt{end}) \tag{5}$$

The simple type configuration in (5) would not be stuck if $\nu = \nu_0$, but would be stuck for any $\nu = \nu'[x \mapsto 10]$. Definition 4 gives a formal definition of simple type configurations that are not stuck, i.e., that are *live*.

Definition 4 (Live simple type configuration). *A simple configuration* (ν, S) *is said* live *if:*

$$S = \mathtt{end} \quad \text{or} \quad \exists t, \ell : (\nu, S) \xrightarrow{t\,\circ m} \qquad (\circ \in \{!, ?\})$$

Observe that for all well-formed S, (ν_0, S) is live.

Subtyping for simple type configurations. We can now define subtyping for simple type configurations and state its decidability.

Definition 5 (Subtyping). $\mathbf{s}_1$ *is a subtype of* $\mathbf{s}_2$*, written* $\mathbf{s}_1 <: \mathbf{s}_2$*, if there exists a timed type simulation* $\mathcal{R}$ on live simple type configurations *such that* $(\mathbf{s}_1, \mathbf{s}_2) \in \mathcal{R}$*. We write* $S_1 <: S_2$ *when* $(\nu_0, S_1) <: (\nu_0, S_2)$*. Abusing the notation, we write* $m <: m'$ *iff there exists* $\mathcal{S}$ *such that* $(m, m') \in \mathcal{S}$*.*

Subtyping has been shown to be decidable in the untimed setting [19] and in the timed first order setting [6]. In [6], decidability is shown through a reduction to model checking of timed automata networks. The result in [6] can be extended to higher-order messages using the techniques in [3], based on finite representations (called regions) of possibly infinite sets of clock valuations.

Proposition 1 (Decidability of subtyping). *Checking if* $(\delta_1, S_1) <: (\delta_2, S_2)$ *is decidable.*

3.3 Types with Queues, and Their Composition

As interactions are asynchronous, the behaviour of types must capture the states in which messages are in transit. To do this, we extend simple type configurations with queues. A *configuration* $\mathbf{S}$ is a triple $(\nu, S, \mathtt{M})$ where ν is clock valuation, S is a type and $\mathtt{M}$ a FIFO unbounded queue of the following form:

$$\mathtt{M} ::= \varnothing \mid m; \mathtt{M}$$

$\mathtt{M}$ contains the messages sent by the co-party of S and not yet received by S. We write $\mathtt{M}$ for $\mathtt{M}; \varnothing$, and call $(\nu, S, \mathtt{M})$ *initial* if $\nu = \nu_0$ and $\mathtt{M} = \varnothing$.

Composing types. Configurations are composed into *systems*. We denote $\mathbf{S} \mid \mathbf{S}'$ as the parallel composition of the two configurations $\mathbf{S}$ and $\mathbf{S}'$.

The labelled transition rules for systems are given in Fig. 2. Rule (snd) is for send actions. A send action can occur only if the time constraint of S is satisfied (by the premise, which uses either rule [snd] or [sel] in Fig. 1). Rule (que) models actions on queues. A queue is always ready to receive any message m. Rule (rcv) is for receive actions, where a message is read from the queue. A receiving action can only occur if the time constraint of S is satisfied (by the premise, which uses either rule [rcv] or [bra] in Fig. 1). The message is removed from the head of the queue of the receiving configuration. The third clause in the premise uses the notion of subtyping (Definition 3) for basic sorts, labels, and higher order messages. Rule (crcv) is the action of a configuration pulling a message of its queue. Rule (com) is for communication between a sending configuration and a buffer. Rule (ctime) lets time elapse in the same way for all configurations in a system. Rule (time) models time passing for single configurations. Time passing is subject to two constrains, expressed by the second and third conditions in the premise. Condition $(\nu, S) \stackrel{!}{\Rightarrow}$ requires the time action t to preserve the satisfiability of some send action. For example, in configuration

$$\frac{(\nu, S) \xrightarrow{!m} (\nu', S')}{(\nu, S, \mathtt{M}) \xrightarrow{!m} (\nu', S', \mathtt{M})}\ (\text{snd}) \qquad (\nu, S, \mathtt{M}) \xrightarrow{?m} (\nu, S, \mathtt{M}; m)\ (\text{que})$$

$$\frac{(\nu, S) \xrightarrow{?m'} (\nu', S') \quad m' <: m}{(\nu, S, m; \mathtt{M}) \xrightarrow{\tau} (\nu', S', \mathtt{M})}\ (\text{rcv}) \qquad \frac{\mathbf{S}_1 \xrightarrow{\tau} \mathbf{S}_1'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_1' \mid \mathbf{S}_2}\ (\text{crcv})$$

$$\frac{\mathbf{S}_1 \xrightarrow{!m} \mathbf{S}_1' \quad \mathbf{S}_2 \xrightarrow{?m} \mathbf{S}_2'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{\tau} \mathbf{S}_1' \mid \mathbf{S}_2'}\ (\text{com}) \qquad \frac{\mathbf{S}_1 \xrightarrow{t} \mathbf{S}_1' \quad \mathbf{S}_2 \xrightarrow{t} \mathbf{S}_2'}{\mathbf{S}_1 \mid \mathbf{S}_2 \xrightarrow{t} \mathbf{S}_1' \mid \mathbf{S}_2'}\ (\text{ctime})$$

$$\frac{(\nu, S) \xrightarrow{t} (\nu', S) \qquad (\nu, S) \overset{!}{\Rightarrow} \text{ implies } (\nu', S) \overset{!}{\Rightarrow} \qquad \forall t' < t : (\nu + t', S, \mathtt{M}) \overset{\tau}{\not\to}}{(\nu, S, \mathtt{M}) \xrightarrow{t} (\nu', S, \mathtt{M})}\ (\text{time})$$

Fig. 2. LTS for systems. We omit the symmetric rules of (crcv), and (csnd).

$(\nu_0, !T(x < 2, \varnothing).S, \varnothing)$, a transition with label 2 would *not* preserve any send action (hence would not be allowed), while a transition with label 1.8 would be allowed by condition $(\nu, S) \overset{!}{\Rightarrow}$. Condition $\forall t' < t : (\nu + t', S, \mathtt{M}) \overset{\tau}{\not\to}$ in the premise of rule (time) checks that there is no ready message to be received in the queue. This is to model urgency: when a configuration is in a receiving state and a message is in the queue then the receiving action must happen without delay. For example, $(\nu_0, ?T(x < 2, \varnothing).S, \varnothing)$ can make a transition with label 1, but $(\nu_0, ?T(x < 2, \varnothing).S, m)$ cannot make any time transition. Below we show two examples of system executions. Example 3 illustrates a good communication, thanks to urgency. We also illustrate in Example 4 that without an urgent semantics the system in Example 3 gets stuck.

Example 3 (A good communication). Consider the following types:

$$S_1 = !T(x \leqslant 1, x).?T(x \leqslant 2).\mathtt{end} \qquad S_2 = ?T(y \leqslant 1, y).!T(y \leqslant 2).\mathtt{end}$$

System $(\nu[x \mapsto 0], S_1, \varnothing) \mid (\nu[x \mapsto 0], S_2, \varnothing)$ can make a time step with label 0.5 by (ctime), yielding the system in (6)

$$(\nu[x \mapsto 0.5], S_1, \varnothing) \mid (\nu[x \mapsto 0.5], S_2, \varnothing) \tag{6}$$

The system in (6) can move by a τ step thanks to (com): the left-hand side configuration makes a step with label $!T$ by (snd) while the right-hand side configuration makes a step $?T$ by (que), yielding system (7) below.

$$(\nu[x \mapsto 0], ?T(x \leqslant 2).\mathtt{end}, \varnothing) \mid (\nu[y \mapsto 0.5], S_2, T) \tag{7}$$

The right-hand side configuration in the system in (7) must *urgently* receive message T due to the third clause in the premise of rule (time). Hence, the only possible step forward for (7) is by (crcv) yielding the system in (8).

$$(\nu[x \mapsto 0], ?T(x \leqslant 2).\mathtt{end}, \varnothing) \mid (\nu[y \mapsto 0], !T(y \leqslant 2).\mathtt{end}, \varnothing) \tag{8}$$

Example 4 (In absence of urgency). Without urgency, the system in (7) from Example 3 may get stuck. Assume the third clause of rule (time) was removed: this would allow (7) to make a time step with label 0.5, followed by a step by (rcv) yielding the system in (9), where clock y is reset after the receive action.

$$(\nu[x \mapsto 0.5], ?T(x \leqslant 2).\mathtt{end}, \varnothing) \mid (\nu[y \mapsto 0], !T(y \leqslant 2).\mathtt{end}, \varnothing) \tag{9}$$

followed by a τ step by (com) reaching the following state:

$$(\nu[x \mapsto 2.5], ?T(x \leqslant 2).\mathtt{end}, T) \mid (\nu[y \mapsto 0], \mathtt{end}, \varnothing) \tag{10}$$

The message in the queue in (10) will never be received as the guard $x \leqslant 2$ is not satisfiable now or at any point in the future. This system is stuck. Instead, thanks to urgency, the clocks of the configurations of system (8) have been 'synchronised' after the receive action, preventing the system from getting stuck.

4 Timed Asynchronous Duality

We introduce a timed extension of duality. As in untimed duality, we let each send/select action be complemented by a corresponding receive/branching action. Moreover, we require time constraints and resets to match.

Definition 6 (Timed duality). *The dual type $\overline{S}$ of S is defined as follows:*

$$\overline{!T(\delta,\lambda).S} = ?T(\delta,\lambda).\overline{S} \qquad \overline{?T(\delta,\lambda).S} = !T(\delta,\lambda).\overline{S} \qquad \overline{\mu\alpha.S} = \mu\alpha.\overline{S}$$
$$\overline{\oplus\{\mathrm{l}_i(\delta_i,\lambda_i) : S_i\}_{i\in I}} = \&\{\mathrm{l}_i(\delta_i,\lambda_i) : \overline{S_i}\}_{i\in I} \qquad \overline{\alpha} = \alpha$$
$$\overline{\&\{\mathrm{l}_i(\delta_i,\lambda_i) : S_i\}_{i\in I}} = \oplus\{\mathrm{l}_i(\delta_i,\lambda_i) : \overline{S_i}\}_{i\in I} \qquad \overline{\mathtt{end}} = \mathtt{end}$$

Duality with urgent receive semantics enjoys the following properties: systems with dual types fulfil progress (Theorem 1); behaviour (resp. progress) of a system is preserved by the substitution of a type with a subtype (Theorem 2) (resp. Theorem 3). A system enjoys progress if it reaches states that are either final or that allow further communications, possibly after a delay. Recall that we assume types to be well-formed (cf. Definition 1): Theorems 1, 2, and 3 rely on this assumption.

Definition 7 (Type progress). *We say that a system* $(\nu, S, \mathtt{M})$ *is a* success *if* $S = \mathtt{end}$ *and* $\mathtt{M} = \varnothing$. *We say that* $\mathbf{S}_1 \mid \mathbf{S}_2$ *satisfies* progress *if:*

$$\mathbf{S}_1 \mid \mathbf{S}_2 \longrightarrow^* \mathbf{S}_1' \mid \mathbf{S}_2' \implies \mathbf{S}_1' \text{ and } \mathbf{S}_2' \text{ are success or } \exists t : \mathbf{S}_1' \mid \mathbf{S}_2' \xrightarrow{t\,\tau}$$

Theorem 1 (Duality progress). *System* $(\nu_0, S, \varnothing) \mid (\nu_0, \overline{S}, \varnothing)$ *enjoys progress.*

We show that subtyping does not introduce new behaviour, via the usual notion of timed simulation [1]. Let $\mathbf{c}, \mathbf{c}_1, \mathbf{c}_2$ range over systems. Fix $\mathbf{c}_1 = (\nu_1^1, S_1^1, \mathtt{M}_1^1) \mid (\nu_2^1, S_2^1, \mathtt{M}_2^1)$, and $\mathbf{c}_2 = (\nu_1^2, S_1^2, \mathtt{M}_1^2) \mid (\nu_2^2, S_2^2, \mathtt{M}_2^2)$. We say that a binary relation over systems preserves `end` if: $S_1^i = \mathtt{end} \wedge \mathtt{M}_1^i = \varnothing$ iff $S_2^i = \mathtt{end} \wedge \mathtt{M}_2^i = \varnothing$ for all $i \in \{1, 2\}$. Write $\mathbf{c}_1 \lesssim \mathbf{c}_2$ if $(\mathbf{c}_1, \mathbf{c}_2)$ are in a timed simulation that preserves `end`.

Theorem 2 (Safe substitution). *If* $S' <: \overline{S}$*, then* $(\nu_0, S, \varnothing) \mid (\nu_0, S', \varnothing) \lesssim (\nu_0, S, \varnothing) \mid (\nu_0, \overline{S}, \varnothing)$.

Theorem 3 (Progressing substitution). *If* $S' <: \overline{S}$*, then* $(\nu_0, S, \varnothing) \mid (\nu_0, S', \varnothing)$ *satisfies progress.*

5 A Calculus for Asynchronous Timed Processes

We introduce our asynchronous calculus for timed processes. The calculus abstracts implementations that execute one or more sessions. We let $P, P', Q, \ldots$ range over processes, X range over process variables, and define $n \in \mathbb{R}_{\geqslant 0} \cup \{\infty\}$. We use the notation $\boldsymbol{a}$ for ordered sequences of channels or variables.

$$
\begin{array}{lll}
P ::= \overline{a}\, v.P & \mid & \texttt{delay}(\delta).\, P \qquad \text{(time-consuming)} \\
\mid \quad a \triangleleft \mathrm{l}.\, P & \mid & a^n(b).\, P \\
\mid \quad \texttt{if } v \texttt{ then } P \texttt{ else } P & \mid & a^n \rhd \{\mathrm{l}_i : P_i\}_{i \in I} \\
\mid \quad P \mid P & \mid & \texttt{failed} \qquad \text{(run-time)} \\
\mid \quad 0 & \mid & \texttt{delay}(t).\, P \\
\mid \quad \texttt{def } D \texttt{ in } P & & \\
\mid \quad X\langle \boldsymbol{a}\, ;\, \boldsymbol{a} \rangle & D ::= X(\boldsymbol{a}\, ;\, \boldsymbol{a}) = P & \\
\mid \quad (\nu ab)P & & \\
\mid \quad ab : h & h ::= \varnothing \mid h \cdot v \mid h \cdot a &
\end{array}
$$

$\overline{a}\, v.P$ sends a value v on channel a and continues as P. Similarly, $a \triangleleft \mathrm{l}.\, P$ sends a label l on channel a and continue as P. Process $\texttt{if } v \texttt{ then } P \texttt{ else } Q$ behaves as either P or Q depending on the boolean value v. Process $P \mid Q$ is for parallel composition of P and Q, and 0 is the idle process. $\texttt{def } D \texttt{ in } P$ is the standard recursive process: D is a declaration, and P is a process that may contain recursive calls. In recursive calls $X\langle \boldsymbol{a}\, ;\, \boldsymbol{a} \rangle$ the first list of parameters has to be instantiated with values of ground types, while the second with channels. Recursive calls are instantiated with equations $X(\boldsymbol{a}\, ;\, \boldsymbol{a})$ in D. Process $(\nu ab)P$ is for scope restriction of endpoints a and b. Process $ab : h$ is a queue with name ab (colloquially used to indicate that it contains messages in transit from a to b) and content h. (νab) binds endpoints a and b, and queues ab and ba in P.

There are two kind of time-consuming processes: those performing a time-consuming action (e.g., method invocation, sleep), and those waiting to receive a message. We model the first kind of processes with $\texttt{delay}(\delta).\, P$, and the second kind of processes with $a^n(b).\, P$ (receive) and $a^n \rhd \{\mathrm{l}_i : P_i\}_{i \in I}$ (branching). In $\texttt{delay}(\delta).\, P$, δ is a constraints as those defined for types, but on one single clock x. The name of the clock here is immaterial: clock x is used as a syntactic tool to define intervals for the time-consuming (delay) action. In this sense, assume x is bound in $\texttt{delay}(\delta).\, P$. Process $\texttt{delay}(\delta).\, P$ consumes any amount of time t such that t is a solution of δ. For example $\texttt{delay}(x \leqslant 3).\, P$ consumes any value between 0 to 3 time units, then behaves as P. Process $a^n(b).\, P$ receive a message on channel a, instantiates b and continue as P. Parameter n models different receive primitives: non-blocking ($n = 0$), blocking ($n = \infty$), and blocking with

timeout ($n \in \mathbb{R}^{\geqslant 0}$). If $n \in \mathbb{R}^{\geqslant 0}$ and no message is in the queue, the process waits n time units before moving into a failed state. If n is set to ∞ the process models a blocking primitive without timeout. Branching process $a^n \triangleright \{\mathrm{l}_i : P_i\}_{i \in I}$ is similar, but receives a label l_i and continues as P_i.

Run-time processes are not written by programmers and only appear upon execution. Process `failed` is the process that has violated a time constraint. We say that P is a *failed state* if it has `failed` as a syntactic sub-term. Process $\texttt{delay}(t).\, P$ delays for exactly t time units.

Well-formed processes. Sessions are modelled as processes of the following form

$$(\nu ab)(P \mid ab : h \mid ba : h')$$

where P is the process for endpoints a and b, ab is the queue for messages from a to b, and ba is the queues for messages from b to a. A process can have more than one ongoing session. For each, we expect that all necessary queues are present and well-placed. We ensure that queues are well-placed via a well-formedness property for processes (see [11] for an inductive definition). Well-formedness rules out processes of the following form:

$$(\nu ab)\ (a^n(c).\, (ba : h' \mid P) \mid Q \mid ab : h) \tag{11}$$

The process in (11) in not well-formed since queue ba for communications to endpoint a is not usable as it is in the continuation of the receive action. Well-formedness of processes is necessary to our safety results. We check well-formedness orthogonally to the typing system for the sake of simpler typing rules. While well-formedness ensures the absence of misplaced queues, the presence of an appropriate pair of queues for every session is ensured by the typing rules.

Session creation. Usually well-formedness is ensured by construction, as sessions are created by a specific (synchronous) reduction rule [10,21]. This kind of session creation is cumbersome in the timed setting as it allows delays that are not captured by protocols, hence well-typed processes may miss deadlines. Other work on timed session types [12] avoids this problem by requiring that all session creations occur before any delay action. Our calculus allows session to be created at any point, even after delays. In (12) a session with endpoints c and d is created after a send action (assume P includes the queues for this new session).

$$(\nu ab)\ (\overline{a}\, v.\texttt{delay}(x \leqslant 3).\, (\nu cd)(P) \mid Q \mid ab : h \mid ba : h') \tag{12}$$

A process like the one in (12) may be thought as a dynamic session creation that happens synchronously (as in [10,21]), but assuming that all participants are ready to engage without delays. Our approach yields a simplification to the calculus (syntax and reduction rules) and, yet, a more general treatment of session initiation than the work in [12].

$$\frac{P \;\rightharpoonup\; P'}{P \;\longrightarrow\; P'} \qquad \frac{P \;\rightsquigarrow\; P'}{P \;\longrightarrow\; P'} \qquad [\texttt{Red1/Red2}]$$

$$\overline{a}\, v.P \mid ab : h \;\rightharpoonup\; P \mid ab : h \cdot v \qquad [\texttt{Send}]$$

$$a^n(c).\, P \mid ba : v \cdot h \;\rightharpoonup\; P[v/c] \mid ba : h \qquad [\texttt{Rcv}]$$

$$a \lhd \mathtt{l}.\, P \mid ab : h \;\rightharpoonup\; P \mid ab : h \cdot \mathtt{l} \qquad [\texttt{Sel}]$$

$$a^n \rhd \{\mathtt{l}_i : P_i\}_{i \in I} \mid ba : \mathtt{l}_j \cdot h \;\rightharpoonup\; P_j \mid ba : h \qquad (j \in I) \qquad [\texttt{Bra}]$$

$$\frac{\models \delta[t/x]}{\mathtt{delay}(\delta).\, P \;\rightharpoonup\; \mathtt{delay}(t).\, P} \qquad [\texttt{Det}]$$

$$\mathtt{if\ true\ then}\ P\ \mathtt{else}\ Q \;\rightharpoonup\; P \qquad [\texttt{IfT}]$$

$$\frac{P \;\rightharpoonup\; P'}{P \mid Q \;\rightharpoonup\; P' \mid Q} \qquad \frac{P \;\rightharpoonup\; P'}{\mathtt{def}\ D\ \mathtt{in}\ P \;\rightharpoonup\; \mathtt{def}\ D\ \mathtt{in}\ P'} \qquad [\texttt{Par/Def}]$$

$$\mathtt{def}\ X(\boldsymbol{a}' ; \boldsymbol{b}') = P'\ \mathtt{in}\ X\langle \boldsymbol{v} ; \boldsymbol{b} \rangle \mid Q \;\rightharpoonup\; \mathtt{def}\ X(\boldsymbol{a}' ; \boldsymbol{b}') = P'\ \mathtt{in}\ P'[\boldsymbol{v}, \boldsymbol{b}/\boldsymbol{a}', \boldsymbol{b}'] \mid Q \qquad [\texttt{Rec}]$$

$$\frac{P \equiv P' \quad P' \;\rightharpoonup\; Q' \quad Q' \equiv Q}{P \;\rightharpoonup\; Q} \qquad \frac{P \;\rightharpoonup\; P'}{(\nu ab)P \;\rightharpoonup\; (\nu ab)P'} \qquad [\texttt{AStr/AScope}]$$

$$\frac{P \equiv P' \quad P' \;\rightsquigarrow\; Q' \quad Q' \equiv Q}{P \;\rightsquigarrow\; Q} \qquad P \;\rightsquigarrow\; \Phi_t(P) \qquad [\texttt{TStr/Delay}]$$

Fig. 3. Reduction for processes (rule [IfF], symmetric for [IfT] is omitted).

$$\Phi_t(0) = 0 \qquad \Phi_t(ab : h) = ab : h \qquad \Phi_t(\mathtt{failed}) = \mathtt{failed}$$

$$\Phi_t(P_1 \mid P_2) = \Phi_t(P_1) \mid \Phi_t(P_2), \text{ if } \mathtt{Wait}(P_i) \cap \mathtt{NEQueue}(P_j) = \varnothing,\ i \neq j \in \{1, 2\}$$

$$\Phi_t(\mathtt{delay}(t').\, P) = \mathtt{delay}(t' - t).\, P \quad \text{if } t' \geqslant t$$

$$\Phi_t(a^{t'}(a').\, P) = \begin{cases} a^{t'-t}(a').\, P & \text{if } t' \geqslant t \\ \mathtt{failed} & \text{otherwise} \end{cases}$$

$$\Phi_t(a^{\infty}(a').\, P) = a^{\infty}(a').\, P$$

$$\Phi_t((\nu ab)P) = (\nu ab)\Phi_t(P)$$

$$\Phi_t(\mathtt{def}\ D\ \mathtt{in}\ P) = \mathtt{def}\ D\ \mathtt{in}\ \Phi_t(P)$$

Fig. 4. Time-passing function $\Phi_t(P)$. Rule for $a^{t'} \rhd \{\mathtt{l}_i : P_i\}_{i \in I}$ is omitted for brevity. $\phi_t(P)$ is undefined in the remaining cases.

Reduction for processes. Processes are considered modulo structural equivalence, denoted by $\equiv$, and defined by adding the following rule for delays to the standard ones [28]: $\texttt{delay}(0).\, P \equiv P$. Reduction rules for processes are given in Fig. 3. A reduction step $\longrightarrow$ can happen because of either an instantaneous step $\rightharpoonup$ by [Red1] or time-consuming step $\rightsquigarrow$ by [Red2]. Rules [Send], [Rcv], [Sel], and [Bra] are the usual asynchronous communication rules. Rule [Det] models the random occurrence of a precise delay t, with t being a solution of δ. The other untimed rules, [IfT], [Par], [Def], [Rec], [AStr], and [AScope] are standard. Note that rule [Par] does not allow time passing, which is handled by rule [Delay]. Rule [TStr] is the timed version of [AStr]. Rule [Delay] applies a *time-passing* function Φ_t (defined in Fig. 4) which distributes the delay t across all the parts of a process. $\Phi_t(P)$ is a partial function: it is undefined if P can immediately make an urgent action, such as evaluation of expressions or output actions. If $\Phi_t(P)$ is defined, it returns the process resulting from letting t time units elapse in P. $\Phi_t(P)$ may return a failed state, if delay t makes a deadline in P expire. The definition of $\Phi_t(P_1 \mid P_2)$ relies on two auxiliary functions: $\texttt{Wait}(P)$ and $\texttt{NEQueue}(P)$ (see [11] for the full definition). $\texttt{Wait}(P)$ returns the set of channels on which P (or some syntactic sub-term of P) is waiting to receive a message/label. $\texttt{NEQueue}(P)$ returns the set of endpoints with a non-empty inbound queue. For example, $\texttt{Wait}(a^t(b).\, Q) = \texttt{Wait}(a^t \rhd \{l_i : P_i\}_{i \in I}) = \{a\}$ and $\texttt{NEQueue}(ba : h) = \{a\}$ given that $h \neq \varnothing$. $\Phi_t(P_1 \mid P_2)$ is defined only if no urgent action could immediately happen in $P_1 \mid P_2$. For example, $\Phi_t(P_1 \mid P_2)$ is undefined for $P_1 = a^t(b).\, Q$ and $P_2 = ba : v$.

In the rest of this section we show the reductions of two processes: one with urgent actions (Example 5), and one to a failed state (Example 6). We omit processes that are immaterial for the illustration (e.g., unused queues).

Example 5 (Urgency and undefined Φ_t). We show the reduction of process $P = (\nu ab)(\overline{a}\,\text{'Hi'}.Q \mid ab : \varnothing \mid b^{10}(c).\, P')$ that has an urgent action. Process P can make the following reduction by [Send]:

$$P \quad \rightharpoonup \quad (\nu ab)(Q \mid ab : \text{'Hi'} \mid b^{10}(c).\, P')$$

At this point, to apply rule [Delay], say with $t = 5$, we need to apply the time-passing function as shown below:

$$\Phi_5((\nu ab)(\overline{a}\,\text{'Hi'}.Q \mid ab : \text{'Hi'} \mid b^{10}(c).\, P')) = (\nu ab)(\overline{a}\,\text{'Hi'}.Q \mid \Phi_5(ab : \text{'Hi'} \mid b^{10}(c).\, P'))$$

which is undefined. $\Phi_5(ab : \varnothing \mid b^{10}(c).\, P')$ is undefined because $\texttt{Wait}(b^{10}(c).\, P) \cap \texttt{NEQueue}(ab : \text{'Hi'}) = \{b\} \neq \varnothing$. Since $\Phi_5(P')$ is undefined. Instead, the message in queue ab can be received by rule [Rcv]:

$$(\nu ab)(Q \mid ab : \text{'Hi'} \mid b^{10}(c).\, P') \quad \rightharpoonup \quad (\nu ab)(Q \mid ab : \varnothing \mid P[\text{'Hi'}/c])$$

Example 6 (An execution with failure). We show a reduction to a failing state of a process with a non-blocking receive action (expecting a message immediately) composed with another process that sends a message after a delay.

$$\begin{array}{ll} \texttt{delay}(x=3).\,\overline{a}\,\text{`Hi'}.Q \mid ab:\varnothing \mid b^0(c).\,P & \text{apply } [\texttt{Det}] \\ \quad \rightharpoonup \ \texttt{delay}(3).\,\overline{a}\,\text{`Hi'}.Q \mid ab:\varnothing \mid b^0(c).\,P = P' & \text{apply } [\texttt{Delay}] \text{ with } t=3 \\ \quad \rightharpoonup \ \Phi_3(P') & \end{array}$$

The application of the time-passing function to P' yields a failing state (a message is not received in time) as shown below, where the second equality holds since $\texttt{Wait}(b^0(c).\,P) \cap \ \texttt{NEQueue}(ab:\varnothing) = \varnothing$:

$$\begin{array}{l} \Phi_3(\texttt{delay}(3).\,\overline{a}\,\text{`Hi'}.Q \mid b^0(c).\,P \mid ab:\varnothing) = \\ \Phi_3(\texttt{delay}(3).\,\overline{a}\,\text{`Hi'}.Q) \mid \Phi_3(b^0(c).\,P \mid \Phi_3(ab:\varnothing)) = \\ \texttt{delay}(0).\,\overline{a}\,\text{`Hi'}.Q \mid \texttt{failed} \mid ab:\varnothing \end{array}$$

6 Typing for Asynchronous Timed Processes

We validate programs against specifications using judgements of the form $\Gamma \vdash P \triangleright \Delta$. Environments are defined as follows:

$$\begin{array}{ll} \Delta ::= \varnothing \mid \Delta, a:(\nu, S) \mid \Delta, ab:\texttt{M} & \Theta ::= \varnothing \mid \Theta \cup \{\Delta\} \\ \Gamma ::= \varnothing \mid \Gamma, a:T \mid \Gamma, X:(\boldsymbol{T};\Theta) & \end{array}$$

Environment Δ is a session environment, used to keep track of the ongoing sessions. When $\Delta(a) = (\nu, S)$ it means that the process being validated is acting as a role in session a specified by S, and ν is the clock valuation describing a (virtual) time in which the next action in S may be executed. We write $\mathrm{dom}(\Delta)$ for the set of variables and channels in Δ. Environment Γ maps variables a to sorts T and process variables X to pairs $(\boldsymbol{T};\Theta)$, where $\boldsymbol{T}$ is a vector of sorts and Θ is a set of session environments. The mapping of process variable is used to type recursive processes: $\boldsymbol{T}$ is used to ensure well-typed instantiation of the recursion parameters, and Θ is used to model the set of possible scenarios when a new iteration begins.

Notation, assumptions, and auxiliary definitions. We write $\Delta + t$ for the session environment obtained by incrementing all clock valuations in the codomain of Δ by t.

Definition 8. *We define the disjoint union $A \uplus B$ of sets of clocks A and B as:*

$$A \uplus B = \{in_l(x) \mid x \in A\} \cup \{in_r(x) \mid x \in B\}$$

where in_l and in_r are one to one endofunctions on clocks and, for all $x \in A$ and $y \in B$, $in_l(x) \neq in_r(y)$. With an abuse of notation, we define the disjoint union of clock valuations ν_1, ν_2, in symbols $\nu_1 \uplus \nu_2$, as a clock valuation satisfying:

$$\nu_1 \uplus \nu_2(in_l(x)) = \nu_1(x) \qquad \nu_1 \uplus \nu_2(in_r(x)) = \nu_2(x)$$

We use the symbol $\biguplus$ for the iterate disjoint union.

For a configuration (ν, S) we define $\mathtt{val}((\nu, \mathtt{S})) = \nu$, and $\mathtt{type}((\nu, \mathtt{S})) = S$. We overload function $\mathtt{val}$ to session environments Δ as follows:

$$\mathtt{val}(\Delta) = \biguplus_{a \in \mathrm{dom}(\Delta)} \mathtt{val}(\Delta(a))$$

We require Θ to satisfy the following three conditions:

1. If $\Delta \in \Theta$ and $\Delta(a) = (\nu, S)$, then S is well-formed (Definition 1) against ν;
2. For all $\Delta_1 \in \Theta$, $\Delta_2 \in \Theta$: $\mathtt{type}(\Delta_1(a)) = S$ iff $\mathtt{type}(\Delta_2(a)) = S$;
3. There is guard δ such that:

$$\{\nu \mid \nu \models \delta\} = \bigcup_{\Delta \in \Theta} \mathtt{val}(\Delta).$$

The last condition ensures that Θ is finitely representable, and is key for decidability of type checking.

Example 7. We show some examples of Θ that do or do not satisfy the last requirement above. Let $S_1 = !T(x \leqslant 2).\mathtt{end}$ and $S_2 = !T(y \leqslant 2).\mathtt{end}$, and let:

$$\Theta_1 = \{\Delta \mid \Delta(a) = (\nu_1, S_1) \wedge \Delta(b) = (\nu_2, S_2) \wedge \nu_1(x) \leqslant 2 \wedge \nu_1(x) = \nu_2(y)\};$$
$$\Theta_2 = \{\Delta \mid \Delta(a) = (\nu_1, S_1) \wedge \Delta(b) = (\nu_2, S_2) \wedge \nu_1(x) \leqslant \sqrt{2} \wedge \nu_1(x) = \nu_2(y)\};$$
$$\Theta_3 = \{\Delta \mid \Delta(a) = (\nu_1, S_1) \wedge \Delta(b) = (\nu_2, S_2) \wedge \nu_1(x) + \nu_2(y) = 2\}.$$

We have that Θ_1 satisfies condition (3): let $\delta_1 = x \leqslant 2 \wedge y - x = 0$. It is easy to see that $\{\nu \mid \nu \models \delta_1\} = \bigcup_{\Delta \in \Theta} \mathtt{val}(\Delta)$. For Θ_2, a candidate proposition would be $\delta_2 = x \leqslant \sqrt{2} \wedge y - x = 0$. However, δ_2 can not be derived with the syntax of propositions, as $\sqrt{2}$ is irrational. Indeed, Θ_2 does not satisfy the condition. For Θ_3, let $\delta_3 = x + y = 2$. Again, δ_3 is not a guard, as additive constraints in the form $x + y = n$ are not allowed. Indeed, also Θ_3 does not satisfy the condition.

In the following, we write $\boldsymbol{a} : \boldsymbol{T}$ for $a_1 : T_1, \ldots, a_n : T_n$ when $\boldsymbol{a} = a_1, \ldots, a_n$ and $\boldsymbol{T} = T_1, \ldots, T_n$ (assuming $\boldsymbol{a}$ and $\boldsymbol{T}$ have the same number of elements). Similarly for $\boldsymbol{b} : \boldsymbol{(\nu, S)}$. In the typing rules, we use a few auxiliary definitions: Definition 9 (t-reading Δ) checks if any ongoing sessions in a Δ can perform an input action within a given timespan, and Definition 10 (Compatibility of configurations) extends the notion of duality to systems that are not in an initial state.

Definition 9 (t-reading Δ). *Session environment Δ is t-reading if there exist some $a \in \mathrm{dom}(\Delta)$, $t' < t$ and m such that: $\Delta(a) = (\nu, S) \wedge (\nu + t', S) \xrightarrow{?m}$.*

Namely, Δ is t-reading if any of the open sessions in the mapping prescribe a read action within the time-frame between ν and $\nu + t$. Definition 9 is used in the typing rules for time-consuming processes – [Vrcv], [Drcv], and [Delt] – to 'disallow' derivations when a (urgent) receive may happen.

Definition 10 (Compatibility of configurations). *Configuration $(\nu_1, S_1, \mathtt{M}_1)$ is compatible with $(\nu_2, S_2, \mathtt{M}_2)$, written $(\nu_1, S_1, \mathtt{M}_1) \perp (\nu_2, S_2, \mathtt{M}_2)$, if:*

1. $\mathtt{M}_1 = \varnothing \vee \mathtt{M}_2 = \varnothing$,
2. $\forall i \neq j \in \{1,2\} : \mathtt{M}_i = m;\mathtt{M}_i' \Rightarrow \exists \nu_i', S_i', m' : (\nu_i, S_i) \xrightarrow{?m'} (\nu_i', S_i') \wedge m <: m' \wedge (\nu_i', S_i', \mathtt{M}_i') \bot (\nu_j, S_j, \mathtt{M}_j)$,
3. $\mathtt{M}_1 = \varnothing \wedge \mathtt{M}_2 = \varnothing \Rightarrow \nu_1 = \nu_2 \wedge S_1 = \overline{S_2}$.

By condition (3) initial configurations are compatible when they include dual types, i.e., $(\nu_0, S, \varnothing) \bot (\nu_0, \overline{S}, \varnothing)$. By condition (2) two configurations may temporarily misalign as execution proceeds: one may have read a message from its queue, while the other has not, as long as the former is ready to receive it immediately. Thanks to the particular shape of type's interactions, initial configurations – of the form $(\nu_0, S, \varnothing) \bot (\nu_0, \overline{S}, \varnothing)$ – will only reach systems, say $(\nu_1, S_1, \mathtt{M}_1) \bot (\nu_2, S_2, \mathtt{M}_2)$, in which at least one between $\mathtt{M}_1$ and $\mathtt{M}_2$ is empty. Condition (1) requires compatible configurations to satisfy this basic property.

Typing rules. The typing rules are given in Fig. 5. Rule [Vrcv] is for input processes. The first premise consists of two conditions requiring the time-span $[\nu, \nu + n]$ in which the process can receive the message to *coincide* with δ:

- $\nu + t \models \delta \Rightarrow t \leqslant n$ rules out processes that are not ready to receive a message when prescribed by the type.
- $t \leqslant n \Rightarrow \nu + t \models \delta$ requires that $a^n(b).P$ can read only at times that satisfy the type prescription δ.[2]

The second premise of [Vrcv] requires the continuation P to be well-typed against the continuation of the type, for all possible session environments where the virtual time is somewhere between $[\nu, \nu + n]$, where the virtual valuation ν in the mapping of session a is reset according to λ. Rule [Drcv], for processes receiving delegated sessions, is like [Vrcv] except: (a) the continuation P is typed against a session environment *extended with the received session* S', and (b) the clock valuation ν' of the receiving session must satisfy δ'. Recall that by formation rules (Sect. 2.1) S' is well-formed against all ν' that satisfy δ'.

Rule [Vsend] is for output processes. Send actions are instantaneous, hence the type current ν needs to satisfy δ. As customary, the continuation of the process needs to be well-typed against the continuation of the type (with ν being reset according to λ, and Γ extended with information on the sort of b). [Dsend] for delegation is similar but: (a) the delegated session is removed from the session environment (the process can no longer engage in the delegated session), and (b) valuation ν' of the delegated session must satisfy guard δ'.

Rule [Delδ] checks that P is well-typed against all possible solutions of δ. Rule [Del$t$] shifts the virtual valuations in the session environment of t. This is as the corresponding rule in [12] but with the addition of the check that Δ is not t-reading, needed because of urgent semantics.

Rule [Res] is for processes with scopes.

[2] While not necessary for our safety results, this constraint simplifies our theory. Timing variations between types and programs are all handled in one place: rule [Subt].

Rule [Rec] is for recursive processes. The rule is as usual [21] except that we use a set of session environments Θ (instead of a single Δ) to capture a set of possible scenarios in which a recursion instance may start, which may have different clock valuations. Rule [Var] is also as expected except for the use of Θ.

Rules [Par] and [Subt] straightforward.

Example 8 (Typing with subtyping). Subtyping substantially increases the power of our type system, in particular in the presence of channel passing. Intuitively, without subtyping, the type of any higher-order send action should be an equality constraint (e.g., $x = 1$) rather than more general timeout (e.g., $x < 1$). We illustrate our point using P defined below:

$$P = (\nu a_1 b_1)(\nu a_2 b_2)(P_1 \mid P_2 \mid P_3 \mid Q) \qquad P_1 = \mathtt{delay}(x \leqslant 1).\, \overline{a_1}\, a_2$$
$$P_2 = b_1^1(c).\, c^2(d) \qquad P_3 = \mathtt{delay}(1 \leqslant x \wedge x \leqslant 2).\, \overline{b_2}\, \mathtt{true}$$

where Q contains empty queues of the involved endpoints. Intuitively, P proceeds as follows: (1) P_1 sends channel a_2 to P_2 within one time unit, and terminates; (2) P_2 reads the message as soon as it arrives, and listens for a message across the received channel (a_2) for two time units; (3) P_3 sends value `true` through channel b_2 at a time in between 1 and 2, unaware that now she is communicating with P_2, and then terminates; (4) P_2 reads the message immediately and terminates. See below for one possible reduction:

$$\begin{aligned} P &\longrightarrow^* (\nu a_1 b_1)(\nu a_2 b_2)(\overline{a_1}\, a_2 \mid b_1^0(c).\, c^2(d) \mid \mathtt{delay}(0 \leqslant x \wedge x \leqslant 1).\, \overline{b_2}\, \mathtt{true}) \mid Q) \\ &\longrightarrow^* (\nu a_1 b_1)(\nu a_2 b_2)(0 \mid a_2^2(d) \mid \mathtt{delay}(0.5).\, \overline{b_2}\, \mathtt{true} \mid Q) \\ &\longrightarrow (\nu a_1 b_1)(\nu a_2 b_2)(0 \mid a_2^{1.5}(d) \mid \overline{b_2}\, \mathtt{true} \mid Q) \\ &\longrightarrow^* (\nu a_1 b_1)(\nu a_2 b_2)(0 \mid 0 \mid 0 \mid Q) \end{aligned}$$

Although P executes correctly, the involved processes are well-typed against types that are not dual:

$$\vdash\ P_1 \rhd a_1 : (\nu_0, S_1), a_2 : (\nu_0, S_2) \quad \vdash\ P_2 \rhd b_1 : (\nu_0, S_1') \quad \vdash\ P_3 \rhd b_2 : (\nu_0, \overline{S_2})$$

for $S_1 = !(y \leqslant 1, S_2)(x \leqslant 1)$, $S_2 = ?\mathtt{Bool}(1 \leqslant y \wedge y \leqslant 2)$, $S_1' = ?(y = 0, S_2')(x \leqslant 1)$. In order to type-check P, we need to apply rule [Res], requiring endpoints of the same session to have dual types. But clearly: $S_1' \neq \overline{S_1}$. Without subtyping, P would not be well-typed. By subtyping, however, $(y \leqslant 1, S_2) <: (y = 0, S_2')$ with $S_2' = ?\mathtt{Bool}(y \leqslant 2).\mathtt{end}$, and then $S_1' <: \overline{S_1'}$. Thanks to the subtyping rule [subt] we can derive $\vdash\ P_2 \rhd b_1 :\ (\nu_0, \overline{S_1})$ and, in turn, $\vdash\ P \rhd \varnothing$.

7 Subject Reduction and Time Safety

The main properties of our typing system are Subject Reduction and Time Safety. Time Safety ensures that the execution of well-typed processes will only

$$\frac{\begin{array}{c}\forall t:\quad \nu + t \models \delta \iff t \leqslant n \\ \forall t \leqslant n:\quad \Gamma, b : T \vdash P \rhd \Delta + t, a : (\nu + t\,[\lambda \mapsto 0],\ S) \quad \Delta \text{ not } t\text{-reading}\end{array}}{\Gamma \vdash a^n(b).\, P \rhd \Delta, a : (\nu,\ ?T(\delta, \lambda).S)} \quad [\texttt{Vrcv}]$$

$$\frac{\begin{array}{c}\forall t:\quad \nu + t \models \delta \iff t \leqslant n \quad T = (\delta', S') \quad \nu' \models \delta' \\ \forall t \leqslant n:\quad \Gamma \vdash P \rhd \Delta + t, a : (\nu + t\,[\lambda \mapsto 0],\ S), b : (\nu',\ S') \quad \Delta \text{ not } t\text{-reading}\end{array}}{\Gamma \vdash a^n(b).\, P \rhd \Delta, a : (\nu,\ ?T(\delta, \lambda).S)} \quad [\texttt{Drcv}]$$

$$\frac{\Gamma \vdash b : T \quad \nu \models \delta \quad \Gamma \vdash P \rhd \Delta, a : (\nu\,[\lambda \mapsto 0],\ S)}{\Gamma \vdash \overline{a}\, b.P \rhd \Delta, a : (\nu,\ !T(\delta, \lambda).S)} \quad [\texttt{Vsend}]$$

$$\frac{T = (\delta', S') \quad \nu' \models \delta' \quad \nu \models \delta \quad \Gamma \vdash P \rhd \Delta, a : (\nu\,[\lambda \mapsto 0],\ S)}{\Gamma \vdash \overline{a}\, b.P \rhd \Delta, a : (\nu,\ !T(\delta, \lambda).S), b : (\nu',\ S')} \quad [\texttt{Dsend}]$$

$$\frac{\forall t \in \delta : \Gamma \vdash \texttt{delay}(t).\, P \rhd \Delta}{\Gamma \vdash \texttt{delay}(\delta).\, P \rhd \Delta} \quad \frac{\Gamma \vdash P \rhd \Delta + t \quad \Delta \text{ not } t\text{-reading}}{\Gamma \vdash \texttt{delay}(t).\, P \rhd \Delta} \quad [\texttt{Del}\delta/\texttt{Del}t]$$

$$\frac{(\nu_1, S_1, \mathtt{M}_1) \bot (\nu_2, S_2, \mathtt{M}_2) \quad \Gamma \vdash P \rhd \Delta, a : (\nu_1,\ S_1), b : (\nu_2,\ S_2), ba : \mathtt{M}_1, ab : \mathtt{M}_2}{\Gamma \vdash (\nu ab)P \rhd \Delta} \quad [\texttt{Res}]$$

$$\frac{\Delta \in \Theta \quad \forall i : \Gamma \vdash \boldsymbol{v}_i : \boldsymbol{T}_i}{\Gamma, X : \boldsymbol{T}; \Theta \vdash X\langle \boldsymbol{v}\,;\, \boldsymbol{b}\rangle \rhd \Delta} \quad \frac{\Gamma \vdash P \rhd \Delta_1 \quad \Gamma \vdash Q \rhd \Delta_2}{\Gamma \vdash P \mid Q \rhd \Delta_1, \Delta_2} \quad [\texttt{Var/Par}]$$

$$\frac{\forall (\boldsymbol{\nu}, \boldsymbol{S}) \in \Theta : \quad \Gamma, \boldsymbol{a} : \boldsymbol{T}, X : \boldsymbol{T}; \Theta \vdash P \rhd \boldsymbol{b} : (\boldsymbol{\nu}, \boldsymbol{S}) \quad \Gamma, X : \boldsymbol{T}; \Theta \vdash Q \rhd \Delta}{\Gamma \vdash \texttt{def}\ X(\boldsymbol{a}\,;\, \boldsymbol{b}) = P\ \texttt{in}\ Q \rhd \Delta} \quad [\texttt{Rec}]$$

$$\frac{\Gamma \vdash P \rhd \Delta' \qquad \Delta' <: \Delta}{\Gamma \vdash P \rhd \Delta} \quad \frac{\Gamma \vdash P \rhd \Delta}{\Gamma \vdash P \rhd \Delta, a : (\nu,\ \texttt{end})} \quad [\texttt{Subt/Weak}]$$

Fig. 5. Selected typing rules for processes

reach *fail-free* states. Recall, P is fail-free when none of its sub-terms is the process `failed`. Time Safety builds on a condition that is not related with time, but with the structure of the process interactions. If an untimed process gets stuck due to mismatches in its communication structure, a timed process with the same communication structure may move to a failed state. Consider P below:

$$\begin{array}{l} P = (\nu ab)(\nu cd)\, Q \qquad R = ab : \varnothing \mid ba : \varnothing \mid cd : \varnothing \mid dc : \varnothing \\ Q = a^5(e).\, \overline{d}\, e.0 \mid c^5(e).\, \overline{b}\, e.0 \mid R \end{array} \tag{13}$$

P is well-typed: $\varnothing \vdash P \rhd a : (\nu_0, S), b : (\nu_0, \overline{S}), c : (\nu_0, S), d : (\nu_0, \overline{S})$ with $S = ?\texttt{Int}(x \leqslant 5, \varnothing).\texttt{end}$. However, P can only make time steps, and when, overall, more than 5 time units elapse (e.g., 6 in the reduction below) P reaches a failed state due to a circular dependency between actions of sessions (νab) and (νcd):

$$P \quad \longrightarrow \quad \Phi_6(Q) = (\nu ab)(\nu cd)\,(\texttt{failed} \mid \texttt{failed} \mid R)$$

Our typing system does not check against such circularities across different interleaved sessions. This is common in work on untimed [21] and timed [12] session types. However, in the untimed scenario, progress for interleaved sessions can be guaranteed by means of additional checks on processes [17]. Time Safety builds on the results in [17] by using an assumption (receive liveness) on the underneath structure of the timed processes. This assumptions is formally captured in Definition 11, which is based on an untimed variant of our calculus.

The untimed calculus. We define untimed processes, denoted by $\hat{P}$, as processes obtained from the grammar given for timed processes (Sect. 5) without delays and failed processes. In untimed processes, time annotations of branching/receive processes are immaterial, hence omitted in the rest of the paper.

Given a (timed) process P, one can obtain its untimed counter-part by *erasing* delays and failed processes; we denoted the result of such erasure on P by $\texttt{erase}(P)$. The semantics of untimed processes is defined as the one for timed processes (Sect. 5) except that reduction rules [Delay], [TStr], and [Red2], are removed. Abusing the notation, we write $\hat{P} \longrightarrow \hat{P}'$ when an untimed process $\hat{P}$ moves to a state $\hat{P}'$ using the semantics for untimed processes. The definitions of $\texttt{Wait}(\hat{P})$ and $\texttt{NEQueue}(\hat{P})$ can be derived from the definitions for timed processes in the straightforward way.

Definition 11 (receive liveness) formalises our assumption on the interaction structures of a process.

Definition 11 (Receive liveness). *$\hat{P}$ is said to satisfy receive liveness (or is* live*, for short) if, for all $\hat{P}'$ such that $\hat{P} \longrightarrow^* \hat{P}'$:*

$$\hat{P}' \equiv (\nu ab)\hat{Q} \ \wedge \ a \in \texttt{Wait}(\hat{Q}) \implies \exists \hat{Q}' : \hat{Q} \longrightarrow^* \hat{Q}' \ \wedge \ a \in \texttt{NEQueue}(\hat{Q}')$$

In any reachable state $\hat{P}'$ of a live untimed process $\hat{P}$, if any endpoint a in $\hat{P}'$ is waiting to receive a message ($a \in \texttt{Wait}(\hat{Q})$), then the overall process is able to reach a state $\hat{Q}'$ where a can perform the receive action ($a \in \texttt{NEQueue}(\hat{Q}')$).

Consider process P in (13). The untimed process $\texttt{erase}(P)$ is not live because $\texttt{Wait}(\texttt{erase}(P)) = \{a, c\}$ and $a, c \notin \texttt{NEQueue}(\texttt{erase}(P))$, since $\texttt{NEQueue}(\texttt{erase}(P))$ is the empty set. Syntactically, $\texttt{erase}(P)$ is as P, but it does not have the same behaviour. P can only make time steps, reaching a failed process, while $\texttt{erase}(P)$ is stuck, as untimed processes only make communication steps.

Properties. Time safety relies on Subject Reduction Theorem 4, which establishes a relation (preserved by reduction) of well-typed processes and their types.

Theorem 4 (Subject reduction for closed systems). *Let $\texttt{erase}(P)$ be live. If $\varnothing \vdash P \rhd \varnothing$ and $P \longrightarrow P'$ then $\varnothing \vdash P' \rhd \varnothing$.*

Note that Subject Reduction assumes $\texttt{erase}(P)$ to be live. For instance, the example of P in (13) is well-typed, but $\texttt{erase}(P)$ is not live. The process can reduce to a failed state (as illustrated earlier in this section) that cannot be typed (failed processes are not well-typed). Time Safety establishes that well-typed processes only reduce to fail-free states.

Theorem 5 (Time safety). *If* $\texttt{erase}(P)$ *is live,* $\vdash P \rhd \varnothing$ *and* $P \longrightarrow^* P'$, *then* P' *is fail-free.*

Typing is decidable if one uses processes annotated with the following information: (1) scope restrictions $(\nu ab : S)P$ are annotated with the type S of the session for endpoint a (the type of b is implicitly assumed to be $\overline{S}$ and both endpoints are type checked in the initial clock valuation ν_0); (2) receive actions $a^n(b : T).\, P$ are annotated with the type T of the received message; (3) recursion $X(\boldsymbol{a : T}\, ; \boldsymbol{a : S}, \delta) = P$ are annotated with types for each parameter, and a guard modelling the state of the clocks. We call annotated programs those annotated processes derived without using productions marked as run-time (i.e., $\texttt{failed}$ and $\texttt{delay}(t).\, P$), and where n in $a^n(b : T).\, P$ ranges over $\mathbb{Q}_{\geqslant 0} \cup \{\infty\}$.

Proposition 2. *Type checking for annotated programs is decidable.*

8 Conclusion and Related Work

We introduced duality and subtyping relations for asynchronous timed session types. Unlike for untimed and timed synchronous [6] dualities, the composition of dual types does not enjoy progress in general. Compositions of asynchronous timed dual types enjoy progress *when using an urgent receive semantics*. We propose a behavioural typing system for a timed calculus that features non-blocking and blocking receive primitives (with and without timeout), and time consuming primitives of arbitrary but constrained delays. The main properties of the typing system are Subject Reduction and Time Safety; both results rely on an assumption (receive liveness) of the underneath interaction structure of processes. In related work on timed session types [12], receive liveness is not required for Subject Reduction; this is because the processes in [12] block (rather than reaching a failed state) whenever they cannot progress correctly, hence e.g., missed deadline are regarded as progress violations. By explicitly capturing failures, our calculus paves the way for future work on combining static checking with run-time instrumentation to prevent or handle failures.

Asynchronous timed session types have been introduced in [12], in a multiparty setting, together with a timed π-calculus, and a type system. The direct extension of session types with time introduces unfeasible executions (i.e., types may get stuck), as we have shown in Example 1. [12] features a notion of feasibility for choreographies, which ensures that types enjoy progress. We ensure progress of types by formation and duality. The semantics of types in [12] is different from ours in that receive actions are not urgent. The work in [12] gives one extra condition on types (wait-freedom), because feasible types may still yield undesirable executions in well-typed processes. Thanks to our duality, subtyping, and calculus (in particular the blocking receive primitive with timeout) this condition is unnecessary in this work. As a result, our typing system allows for types that are *not wait-free*. By dropping wait-freedom, we can type a class of common real-world protocols in which processes may be ready to receive messages even before the final deadline of the corresponding senders. Remarkably,

SMTP mentioned in the introduction is *not wait-free*. For some other aspects, our work is less general than the one in [12], as we consider binary sessions rather than multiparty sessions. A theory of timed multiparty asynchronous protocols that encompasses the protocols in [12] and those considered here is an interesting future direction. The work in [6] introduces a theory of synchronous timed session types, based on a decidable notion of compatibility, called *compliance*, that ensures progress of types, and is equivalent to synchronous timed duality and subtyping in a precise sense [6]. Our duality and subtyping are similar to those in [6], but apply to the asynchronous scenario. The work in [15] introduces a typed calculus based on temporal session types. The temporal modalities in [15] can be used as a discrete model of time. Timed session types, thanks to clocks and resets, are able to model complex timed dependencies that temporal session types do not seem able to capture. Other work studies models for asynchronous timed interactions, e.g., Communicating Timed Automata [23] (CTA), timed Message Sequence Charts [2], but not their relationships with processes. The work in [5] introduces a refinement for CTA, and presents a notion of urgency similar to the one used in this paper, preliminary studied also in [29].

Several timed calculi have been introduced outside the context of behavioural types. The work in [32] extends the π- calculus with time primitives inspired in CTA and is closer, in principle, to our types than our processes. Another timed extension of the π-calculus with time-consuming actions has been applied to the analysis the active times of processes [18]. Some works focus on specific aspects of timed behaviour, such as timeouts [9], transactions [24,27], and services [25]. Our calculus does not feature exception handlers, nor timed transactions. Our focus in on detecting time violations via static typing, so that a process only moves to fail-free states.

The calculi in [7,12,15] have been used in combination with session types. The calculus in [12] features a non-blocking receive primitive similar to our $a^0(b).P$, but that never fails (i.e., time is not allowed to flow if a process tries to read from an empty buffer—possibly leading to a stuck process rather than a failed state). The calculus in [7] features a blocking receive primitive without timeout, equivalent to our $a^\infty(b).P$. The calculus in [15], seems able to encode a non-blocking receive primitive like the one of [12] and a blocking receive primitive without timeout like our $a^\infty(b).P$. None of these works features blocking receive primitives with timeouts. Furthermore, existing works feature [7,12] or can encode [15] only precise delays, equivalent to $\texttt{delay}(x = n).P$. Such punctual predictions are often difficult to achieve. Arbitrary but constrained delays are closer abstractions of time-consuming programming primitives (and possibly, of predictions one can derive by cost analysis, e.g., [20]).

As to applications, timed session types have been used for run-time monitoring [7,30] and static checking [12]. A promising future direction is that of integrating static typing with run-time verification and enforcement, towards a theory of hybrid timed session types. In this context, extending our calculus with exception handlers [9,24,27] could allow an extension of the typing system, that introduces run-time instrumentation to handle unexpected time failures.

References

1. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge (2007). https://doi.org/10.1017/CBO9780511814105
2. Akshay, S., Gastin, P., Mukund, M., Kumar, K.N.: Model checking time-constrained scenario-based specifications. In: FSTTCS. LIPIcs, vol. 8, pp. 204–215. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010). https://doi.org/10.4230/LIPIcs.FSTTCS.2010.204
3. Alur, R., Dill, D.L.: A theory of timed automata. TCS **126**, 183–235 (1994)
4. Advanced Message Queuing Protocols (AMQP). https://www.amqp.org/
5. Bartoletti, M., Bocchi, L., Murgia, M.: Progress-preserving refinements of CTA. In: CONCUR. LIPIcs, vol. 118, pp. 40:1–40:19. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018). https://doi.org/10.4230/LIPIcs.CONCUR.2018.40
6. Bartoletti, M., Cimoli, T., Murgia, M.: Timed session types. Log. Methods Comput. Sci. **13**(4) (2017). https://doi.org/10.23638/LMCS-13(4:25)2017
7. Bartoletti, M., Cimoli, T., Murgia, M., Podda, A.S., Pompianu, L.: A contract-oriented middleware. In: Braga, C., Ölveczky, P.C. (eds.) FACS 2015. LNCS, vol. 9539, pp. 86–104. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28934-2_5
8. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
9. Berger, M., Yoshida, N.: Timed, distributed, probabilistic, typed processes. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 158–174. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76637-7_11
10. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_33
11. Bocchi, L., Murgia, M., Vasconcelos, V., Yoshida, N.: Asynchronous timed session types: from duality to time-sensitive processes (2018). https://www.cs.kent.ac.uk/people/staff/lb514/tstp.html
12. Bocchi, L., Yang, W., Yoshida, N.: Timed multiparty session types. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 419–434. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44584-6_29
13. Bruno, E.J., Bollella, G.: Real-Time Java Programming: With Java RTS, 1st edn. Prentice Hall PTR, Upper Saddle River (2009)
14. Chen, T.C., Dezani-Ciancaglini, M., Yoshida, N.: On the preciseness of subtyping in session types. In: PPDP, pp. 135–146. ACM (2014). https://doi.org/10.1145/2643135.2643138
15. Das, A., Hoffmann, J., Pfenning, F.: Parallel complexity analysis with temporal session types. Proc. ACM Program. Lang. **2**(ICFP), 91:1–91:30 (2018). https://doi.org/10.1145/3236786
16. Demangeon, R., Honda, K.: Full abstraction in a subtyped pi-calculus with linear types. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 280–296. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_19

17. Dezani-Ciancaglini, M., de'Liguoro, U., Yoshida, N.: On progress for structured communications. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 257–275. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78663-4_18
18. Fischer, M., Förster, S., Windisch, A., Monjau, D., Balser, B.: A new time extension to π-calculus based on time consuming transition semanticss. In: Grimm, C. (ed.) Languages for System Specification, pp. 271–283. Springer, Boston (2004). https://doi.org/10.1007/1-4020-7991-5_17
19. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Inf. **42**(2–3), 191–225 (2005). https://doi.org/10.1007/s00236-005-0177-z
20. Hoffmann, J., Shao, Z.: Automatic static cost analysis for parallel programs. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 132–157. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_6
21. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL, pp. 273–284. ACM (2008)
22. Klensin, J.: Simple mail transfer protocol. RFC 5321, October 2008. https://tools.ietf.org/html/rfc5321
23. Krcal, P., Yi, W.: Communicating timed automata: the more synchronous, the more difficult to verify. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 249–262. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_24
24. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FoSSaCS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31982-5_18
25. Lapadula, A., Pugliese, R., Tiezzi, F.: CWS: a timed service-oriented calculus. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 275–290. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75292-9_19
26. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. Int. J. Softw. Tools Technolo. Transf. **1**, 134–152 (1997)
27. López, H.A., Pérez, J.A.: Time and exceptional behavior in multiparty structured interactions. In: Carbone, M., Petit, J.-M. (eds.) WS-FM 2011. LNCS, vol. 7176, pp. 48–63. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29834-9_5
28. Milner, R.: Communicating and Mobile Systems: The π-calculus. Cambridge University Press, New York (1999)
29. Murgia, M.: On urgency in asynchronous timed session types. In: ICE. EPTCS, vol. 279, pp. 85–94 (2018). https://doi.org/10.4204/EPTCS.279.9
30. Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. Formal Asp. Comput. **29**(5), 877–910 (2017). https://doi.org/10.1007/s00165-017-0420-8
31. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
32. Saeedloei, N., Gupta, G.: Timed π-calculus. In: Abadi, M., Lluch Lafuente, A. (eds.) TGC 2013. LNCS, vol. 8358, pp. 119–135. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05119-2_8
33. Vinoski, S.: Advanced message queuing protocol. IEEE Internet Comput. **10**(6), 87–89 (2006). https://doi.org/10.1109/MIC.2006.116
34. Yovine, S.: Kronos: a verification tool for real-time systems. (Kronos user's manual release 2.2). Int. J. Softw. Tools Technol. Transf. **1**, 123–133 (1997)

Manifest Deadlock-Freedom for Shared Session Types

Stephanie Balzer[1(✉)], Bernardo Toninho[2(✉)], and Frank Pfenning[1]

[1] Carnegie Mellon University, Pittsburgh, USA
balzers@cs.cmu.edu
[2] NOVA LINCS, Universidade Nova de Lisboa, Lisbon, Portugal
btoninho@fct.unl.pt

Abstract. Shared session types generalize the Curry-Howard correspondence between intuitionistic linear logic and the session-typed π-calculus with adjoint modalities that mediate between linear and shared session types, giving rise to a programming model where shared channels must be used according to a locking discipline of acquire-release. While this generalization greatly increases the range of programs that can be written, the gain in expressiveness comes at the cost of deadlock-freedom, a property which holds for many linear session type systems. In this paper, we develop a type system for logically-shared sessions in which types capture not only the interactive behavior of processes but also constrain the order of resources (i.e., shared processes) they may acquire. This type-level information is then used to rule out cyclic dependencies among acquires and synchronization points, resulting in a system that ensures *deadlock-free communication* for well-typed processes in the presence of shared sessions, higher-order channel passing, and recursive processes. We illustrate our approach on a series of examples, showing that it rules out deadlocks in circular networks of both shared and linear recursive processes, while still being permissive enough to type concurrent implementations of shared imperative data structures as processes.

Keywords: Linear and shared session types · Deadlock-freedom

1 Introduction

Session types [25–27] naturally describe the interaction protocols that arise amongst concurrent processes that communicate via message-passing. This typing discipline has been integrated (with varying static safety guarantees) into several mainstream language such as Java [28,29], F# [43], Scala [49,50], Go [11] and Rust [33]. Session types moreover enjoy a logical correspondence between *linear logic* and the *session-typed* π*-calculus* [8,9,51,55]. Languages building on this correspondence [24,52,55] not only guarantee *session*

Supported by NSF Grant No. CCF-1718267: "Enriching Session Types for Practical Concurrent Programming" and NOVA LINCS (Ref. UID/CEC/04516/2019).

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 611–639, 2019.
https://doi.org/10.1007/978-3-030-17184-1_22

fidelity (i.e., type preservation) but also *deadlock-freedom* (i.e., global progress). The latter is guaranteed even in the presence of interleaved sessions, which are often excluded from the deadlock-free fragments of traditional session-typed frameworks [20,26,27,53]. These logical session types, however, exclude programming scenarios that demand *sharing* of mutable resources (e.g., shared databases or shared output devices) instead of functional resource replication.

To increase their practicality, logical session types have been extended with *manifest sharing* [2]. In the resulting language, linear and shared sessions coexist, but the type system enforces that clients of shared sessions run in mutual exclusion of each other. This separation is achieved by enforcing an *acquire-release* policy, where a client of a shared session must first acquire the session before it can participate in it along a private linear channel. Conversely, when a client releases a session, it gives up its linear channel and only retains a shared reference to the session. Thus, sessions in the presence of manifest sharing can change, or *shift*, between shared and linear execution modes. At the type-level, the acquire-release policy manifests in a stratification of session types into linear and shared with adjoint modalities [5,47,48], connecting the two strata. Operationally, the modality shifting *up* from the linear to the shared layer translates into an *acquire* and the one shifting *down* from shared to linear into a *release*.

Manifest sharing greatly increases the range of programs that can be written because it recovers the expressiveness of the untyped asynchronous π-calculus [3] while maintaining session fidelity. As in the π-calculus, however, the gain in expressiveness comes at the cost of *deadlock-freedom*. An illustrative example is an implementation of the classical dining philosophers problem, shown in Fig. 1, using the language SILL$_S$ [2] that supports manifest sharing (in this setting we often equate a process with the session it offers along a distinguished channel). The code shows the process *fork_proc*, implementing a session of type sfork, and the processes *thinking* and *eating*, implementing sessions of type philosopher. We defer the details of the typing and the definition of the session types sfork and philosopher to Sect. 2 and focus on the programmatic working of the processes for now. For ease of reading, we typeset shared session types and variables denoting shared channel references in red.

A *fork_proc* process represents a fork that can be perpetually acquired and released. The actions accept and detach are the duals of acquire and release, respectively, allowing a process to accept an acquire by a client and to initiate a release by a client, respectively. Process thinking has two shared channel references as arguments, for the forks to the left and right of the philosopher, which the process tries to acquire. If the acquire succeeds, the process recurs as an eating philosopher with two (now) linear channel references of type lfork. Once a philosopher is done eating, it releases both forks and recurs as a thinking philosopher. Let's set a table for three philosopher that share three forks, all spawned as processes executing in parallel:

$f_0 \leftarrow \mathit{fork\_proc}$; $f_1 \leftarrow \mathit{fork\_proc}$; $f_2 \leftarrow \mathit{fork\_proc}$;
$p_0 \leftarrow \mathit{thinking} \leftarrow f_0, f_1$; $p_1 \leftarrow \mathit{thinking} \leftarrow f_1, f_2$; $p_2 \leftarrow \mathit{thinking} \leftarrow f_2, f_0$;

```
fork_proc : {sfork}
c ← fork_proc =
  c′ ← accept c ;
  c ← detach c′ ;
  c ← fork_proc

thinking : {phil ← sfork, sfork}
c ← thinking ← left, right =
  left′ ← acquire left ;
  right′ ← acquire right ;
  c ← eating ← left′, right′ ;

eating : {phil ← lfork, lfork}
c ← eating ← left′, right′ =
  right ← release right′ ;
  left ← release left′ ;
  c ← thinking ← left, right
```

Fig. 1. Dining philosophers in SILL_S [2].

Infamously, this configuration may deadlock because of the *circular* dependency between the acquires. We can break this cycle by changing the last line to $p_2 \leftarrow \mathit{thinking} \leftarrow f_0, f_2$, ensuring that forks are acquired in increasing order.

Perhaps surprisingly, cyclic dependencies between acquire requests are not the only source of deadlocks. Fig. 2 gives an example, defining the processes *owner* and *contester*, which both have a shared channel reference to a common resource that can be perpetually acquired and released. Both processes acquire the shared resource, but additionally exchange the message ping. More precisely, process *owner* spawns the process *contester*, acquires the shared resource, and only releases the resource after having received the message ping from the *contester*. Process *contester*, on the other hand, first attempts to acquire the resource and then sends the message *ping* to the owner. The program deadlocks if process *owner* acquires the resource first. In that case, process *owner* waits for process *contester* to send the message ping while process *contester* waits to acquire the resource held by process *owner*. We note that this deadlock arises in both synchronous and asynchronous semantics.

```
owner : {1 ← sres}
o ← owner ← sr =
  c ← contester ← sr ;
  lr ← acquire sr ;
  case c of
  | ping → wait c ;
           sr ← release lr ; close o

contester : {⊕{ping : 1} ← sres}
c ← contester ← sr =
  lr ← acquire sr ;
  c.ping ;
  sr ← release lr ;
  close c
```

Fig. 2. Circular dependencies among acquire and synchronization actions.

In this paper, we develop a type system for manifest sharing that rules out cycles between acquire requests and interdependencies between acquire requests and synchronization actions, detecting the two kinds of deadlocks explained above. In our type system, session types not only prescribe *when* resources must be acquired and released, but also the *range* of resources that may be acquired. To this end, we equip the type system with the notion of a *world*, an abstract value at which a process resides, and type processes relative to an acyclic *ordering* on worlds, akin to the partial-order based approaches of [34,37]. The contributions of this paper are:

- a characterization of the possible forms of deadlocks that can arise in shared session types;
- the introduction of manifest deadlock-freedom, where resource dependencies are manifest in the type structure via world modalities;
- its elaboration in the programming language $\mathsf{SILL}_{\mathsf{S}^+}$, resulting in a type system, a synchronous operational semantics, and proofs of session fidelity (preservation) and a strong form of progress that excludes all deadlocks;
- the novel abstraction of green and red arrows to reason about the interdependencies between processes;
- an illustration of the concepts on various examples, including an extensive comparison with related work.

This paper is structured as follows: Sect. 2 provides a short introduction to manifest sharing. Sect. 3 develops the type system and dynamics of the language $\mathsf{SILL}_{\mathsf{S}^+}$. Sect. 4 illustrates the introduced concepts on an extended example. Sect. 5 discusses the meta-theoretical properties of $\mathsf{SILL}_{\mathsf{S}^+}$, emphasizing progress. Sect. 6 compares with examples of related work and identifies future work. Sect. 7 discusses related work, and Sect. 8 concludes this paper.

2 Manifest Sharing

In the previous section, we have already explored the programmatic workings of *manifest sharing* [2], which enforces an *acquire-release* policy on shared channel references. In this section, we clarify the typing of shared processes.

A key contribution of manifest sharing is not only to support acquire-release as a programming primitive but also to make it *manifest* in the type system. Generalizing the idea of type *stratification* [5,47,48], session types are partitioned into a linear and shared layer with two *adjoint modalities* connecting the layers:

$$\begin{array}{lcl} A_{\mathsf{S}} & \triangleq & \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}} \\ A_{\mathsf{L}}, B_{\mathsf{L}} & \triangleq & A_{\mathsf{L}} \otimes B_{\mathsf{L}} \mid \oplus\{\overline{l : A_{\mathsf{L}}}\} \mid \&\{\overline{l : A_{\mathsf{L}}}\} \mid A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid \exists x{:}A_{\mathsf{S}}.B_{\mathsf{L}} \mid \Pi x{:}A_{\mathsf{S}}.B_{\mathsf{L}} \mid \mathbf{1} \mid \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}} \end{array}$$

In the linear layer, we get the standard connectives of intuitionistic linear logic ($A_{\mathsf{L}} \otimes B_{\mathsf{L}}$, $A_{\mathsf{L}} \multimap B_{\mathsf{L}}$, $\oplus\{\overline{l : A_{\mathsf{L}}}\}$, $\&\{\overline{l : A_{\mathsf{L}}}\}$, and $\mathbf{1}$). These connectives are extended with the modal operator $\downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}}$, shifting *down* from the shared to the linear layer. Similarly, in the shared layer, we have the operator $\uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}$, shifting *up* from the linear to the shared layer. The former translates into a *release* (and, dually, detach), the latter into an *acquire* (and, dually, accept). As a result, we obtain a system in which session types prescribe all forms of communication, including the acquisition and release of shared processes.

Table 1 provides an overview of $\mathsf{SILL}_{\mathsf{S}}$'s session types and their operational reading. Since $\mathsf{SILL}_{\mathsf{S}}$ is based on an intuitionistic interpretation of linear logic session types [8], types are expressed from the point of view of the *providing process* with the channel along which the process provides the session behavior being characterized by its session type. This choice avoids the explicit duality operation present in original presentations of session types [25,26] and in those based

Table 1. Session types in SILL_S and their operational meaning.

Session type current	cont	Process term current	cont	Description
$c_\mathsf{L} : \oplus\{\overline{l : A_\mathsf{L}}\}$	$c_\mathsf{L} : A_{\mathsf{L}_h}$	$c_\mathsf{L}.l_h\ ; P$	P	sends label l_h along c_L
		$\mathsf{case}\ c_\mathsf{L}\ \mathsf{of}\ \overline{l \Rightarrow Q}$	Q_h	receives label l_h along c_L
$c_\mathsf{L} : \&\{\overline{l : A_\mathsf{L}}\}$	$c_\mathsf{L} : A_{\mathsf{L}_h}$	$\mathsf{case}\ c_\mathsf{L}\ \mathsf{of}\ \overline{l \Rightarrow P}$	P_h	receives label l_h along c
		$c_\mathsf{L}.l_h\ ; Q$	Q	sends label l_h along c_L
$c_\mathsf{L} : A_\mathsf{L} \otimes B_\mathsf{L}$	$c_\mathsf{L} : B_\mathsf{L}$	$\mathsf{send}\ c_\mathsf{L}\ d_\mathsf{L}\ ; P$	P	sends channel $d_\mathsf{L} : A_\mathsf{L}$ along c_L
		$y_\mathsf{L} \leftarrow \mathsf{recv}\ c_\mathsf{L}\ ; Q_{y_\mathsf{L}}$	$[d_\mathsf{L}/y_\mathsf{L}]\, Q_{y_\mathsf{L}}$	receives channel $d_\mathsf{L} : A_\mathsf{L}$ along c_L
$c_\mathsf{L} : A_\mathsf{L} \multimap B_\mathsf{L}$	$c_\mathsf{L} : B_\mathsf{L}$	$y_\mathsf{L} \leftarrow \mathsf{recv}\ c_\mathsf{L}\ ; P_{y_\mathsf{L}}$	$[d_\mathsf{L}/y_\mathsf{L}]\, P_{y_\mathsf{L}}$	receives channel $d_\mathsf{L} : A_\mathsf{L}$ along c_L
		$\mathsf{send}\ c_\mathsf{L}\ d_\mathsf{L}\ ; Q$	Q	sends channel $d_\mathsf{L} : A_\mathsf{L}$ along c_L
$c_\mathsf{L} : \Pi x{:}A_\mathsf{S}.B_\mathsf{L}$	$c_\mathsf{L} : B_\mathsf{L}$	$\mathsf{send}\ c_\mathsf{L}\ d_\mathsf{S}\ ; P$	P	sends channel $d_\mathsf{S} : A_\mathsf{S}$ along c_L
		$y_\mathsf{S} \leftarrow \mathsf{recv}\ c_\mathsf{L}\ ; Q_{y_\mathsf{S}}$	$[d_\mathsf{S}/y_\mathsf{S}]\, Q_{y_\mathsf{S}}$	receives channel $d_\mathsf{S} : A_\mathsf{S}$ along c_L
$c_\mathsf{L} : \exists x{:}A_\mathsf{S}.B_\mathsf{L}$	$c_\mathsf{L} : B_\mathsf{L}$	$y_\mathsf{S} \leftarrow \mathsf{recv}\ c_\mathsf{L}\ ; P_{y_\mathsf{S}}$	$[d_\mathsf{S}/y_\mathsf{S}]\, P_{y_\mathsf{S}}$	receives channel $d_\mathsf{S} : A_\mathsf{S}$ along c_L
		$\mathsf{send}\ c_\mathsf{L}\ d_\mathsf{S}\ ; Q$	Q	sends channel $d_\mathsf{S} : A_\mathsf{S}$ along c_L
$c_\mathsf{L} : \mathbf{1}$	-	$\mathsf{close}\ c_\mathsf{L}$	-	sends "end" along c_L
		$\mathsf{wait}\ c_\mathsf{L}\ ; Q$	Q	receives "end" along c_L
$c_\mathsf{L} : \downarrow^\mathsf{S}_\mathsf{L} A_\mathsf{S}$	$c_\mathsf{S} : A_\mathsf{S}$	$c_\mathsf{S} \leftarrow \mathsf{detach}\ c_\mathsf{L}\ ; P_{x_\mathsf{S}}$	$[c_\mathsf{S}/x_\mathsf{S}]\, P_{x_\mathsf{S}}$	sends "detach c_S" along c_L
		$x_\mathsf{S} \leftarrow \mathsf{release}\ c_\mathsf{L}\ ; Q_{x_\mathsf{S}}$	$[c_\mathsf{S}/x_\mathsf{S}]\, Q_{x_\mathsf{S}}$	receives "detach c_S" along c_L
$c_\mathsf{S} : \uparrow^\mathsf{S}_\mathsf{L} A_\mathsf{L}$	$c_\mathsf{L} : A_\mathsf{L}$	$c_\mathsf{L} \leftarrow \mathsf{acquire}\ c_\mathsf{S}\ ; Q_{x_\mathsf{L}}$	$[c_\mathsf{L}/x_\mathsf{L}]\, Q_{x_\mathsf{L}}$	sends "acquire c_L" along c_S
		$x_\mathsf{L} \leftarrow \mathsf{accept}\ c_\mathsf{S}\ ; P_{x_\mathsf{L}}$	$[c_\mathsf{L}/x_\mathsf{L}]\, P_{x_\mathsf{L}}$	receives "acquire c_L" along c_S

on classical linear logic [55]. Table 1 lists the points of view of the *provider* and *client* of a given connective in the first and second lines, respectively. Moreover, Table 1 gives for each connective its session type before and after the message exchange, along with their respective process terms. We can see that the process terms of a provider and a client for a given connective come in matching pairs, indicating that the participants' views of the session change consistently. We use the subscripts L and S to distinguish between linear and shared channels, respectively.

We are now able to give the session types of the processes *fork_proc*, *thinking*, and *eating* defined in the previous section:

$$\mathsf{lfork} = \downarrow^\mathsf{S}_\mathsf{L} \mathsf{sfork}$$
$$\mathsf{sfork} = \uparrow^\mathsf{S}_\mathsf{L} \mathsf{lfork}$$
$$\mathsf{phil} = \mathbf{1}$$

The mutually recursive session types lfork and sfork represent a fork that can perpetually be acquired and released. We adopt an *equi-recursive* [14] interpretation for recursive session types, silently equating a recursive type with its unfolding and requiring types to be *contractive* [19].

We briefly discuss the typing and the dynamics of acquire-release. The typing and the dynamics of the residual linear connectives are standard, and we detail them in the context of $\mathsf{SILL}_{\mathsf{S}^+}$ (see Sect. 3). As is usual for an intuitionistic

interpretation, each connective gives rise to a left and a right rule, denoting the use and provision, respectively, of a session of the given type:

$$(\mathrm{T}\text{-}\uparrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{R})\ \frac{\Gamma;\cdot \vdash P_{x_{\mathsf{L}}} :: (x_{\mathsf{L}} : A_{\mathsf{L}})}{\Gamma \vdash x_{\mathsf{L}} \leftarrow \mathsf{accept}\, x_{\mathsf{S}}; P_{x_{\mathsf{L}}} :: (x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}})} \qquad (\mathrm{T}\text{-}\uparrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{L})\ \frac{\Gamma, x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}; \Delta, x_{\mathsf{L}} : A_{\mathsf{L}} \vdash Q_{x_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma, x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}; \Delta \vdash x_{\mathsf{L}} \leftarrow \mathsf{acquire}\, x_{\mathsf{S}}; Q_{x_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}$$

$$(\mathrm{T}\text{-}\downarrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{R})\ \frac{\Gamma \vdash P_{x_{\mathsf{S}}} :: (x_{\mathsf{S}} : A_{\mathsf{S}})}{\Gamma;\cdot \vdash x_{\mathsf{S}} \leftarrow \mathsf{detach}\, x_{\mathsf{L}}; P_{x_{\mathsf{S}}} :: (x_{\mathsf{L}} : \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}})} \qquad (\mathrm{T}\text{-}\downarrow^{\mathsf{S}}_{\mathsf{L}}\mathrm{L})\ \frac{\Gamma, x_{\mathsf{S}} : A_{\mathsf{S}}; \Delta \vdash Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma; \Delta, x_{\mathsf{L}} : \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}} \vdash x_{\mathsf{S}} \leftarrow \mathsf{release}\, x_{\mathsf{L}}; Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}$$

The typing judgments $\Gamma \vdash P :: (x_{\mathsf{S}} : A_{\mathsf{S}})$ and $\Gamma; \Delta \vdash P :: (x_{\mathsf{L}} : A_{\mathsf{L}})$ indicate that process P provides a session of type A along channel x, given the typing of the channels specified in typing contexts Γ (and Δ). Γ and Δ consist of hypotheses on the typing of shared and linear channels, respectively, where Γ is a structural and Δ a linear context. To allow for recursive process definitions, the typing judgment depends on a signature Σ that is populated with all process definitions prior to type-checking. The adjoint formulation precludes shared processes from depending on linear channel references [2,47], a restriction motivated from logic referred to as the independence principle [47]. Thus, when a shared session accepts an acquire and shifts to linear, it starts with an empty linear context.

Operationally, the dynamics of $\mathsf{SILL}_{\mathsf{S}}$ is captured by *multiset rewriting rules* [12], which denote computation in terms of state transitions between configurations of processes. Multiset rewriting rules are local in that they only mention the parts of a configuration they rewrite. For acquire-release we have the following:

$(\mathrm{D}\text{-}\uparrow^{\mathsf{S}}_{\mathsf{L}})$
$\mathsf{proc}(a_{\mathsf{S}}, x_{\mathsf{L}} \leftarrow \mathsf{accept}\, a_{\mathsf{S}}\, ; P_{x_{\mathsf{L}}}), \mathsf{proc}(c_{\mathsf{L}}, x_{\mathsf{L}} \leftarrow \mathsf{acquire}\, a_{\mathsf{S}}\, ; Q_{x_{\mathsf{L}}})$
$\longrightarrow \mathsf{proc}(a_{\mathsf{L}}, [a_{\mathsf{L}}/x_{\mathsf{L}}]\, P_{x_{\mathsf{L}}}), \mathsf{proc}(c_{\mathsf{L}}, [a_{\mathsf{L}}/x_{\mathsf{L}}]\, Q_{x_{\mathsf{L}}}), \mathsf{unavail}(a_{\mathsf{S}})$

$(\mathrm{D}\text{-}\downarrow^{\mathsf{S}}_{\mathsf{L}})$
$\mathsf{proc}(a_{\mathsf{L}}, x_{\mathsf{S}} \leftarrow \mathsf{detach}\, a_{\mathsf{L}}\, ; P_{x_{\mathsf{S}}}), \mathsf{proc}(c_{\mathsf{L}}, x_{\mathsf{S}} \leftarrow \mathsf{release}\, a_{\mathsf{L}}\, ; Q_{x_{\mathsf{S}}}), \mathsf{unavail}(a_{\mathsf{S}})$
$\longrightarrow \mathsf{proc}(a_{\mathsf{S}}, [a_{\mathsf{S}}/x_{\mathsf{S}}]\, P_{x_{\mathsf{S}}}), \mathsf{proc}(c_{\mathsf{L}}, [a_{\mathsf{S}}/x_{\mathsf{S}}]\, Q_{x_{\mathsf{S}}})$

Configuration states are defined by the predicates $\mathsf{proc}(c_m, P)$ and $\mathsf{unavail}(a_{\mathsf{S}})$. The former denotes a running process with process term P providing along channel c_m, the latter acts as a placeholder for a shared process providing along channel a_{S} that is currently not available. The above rule exploits the invariant that a process' providing channel a can appear at one of two modes, a linear one, a_{L}, and a shared one, a_{S}. While the process (i.e. the session) is linear, it provides along a_{L}, while it is shared, along a_{S}. When a process shifts between modes, it switches between the two modes of its offering channel. The channel at the appropriate mode is substituted for the variables occurring in process terms.

3 Manifest Deadlock-Freedom

In this section, we introduce our language $\mathsf{SILL}_{\mathsf{S}^+}$, a session-typed language that supports sharing without deadlock. We focus on $\mathsf{SILL}_{\mathsf{S}^+}$'s type system and dynamics in this section and discuss its meta-theoretical properties in Sect. 5.

3.1 Competition and Collaboration

The introduction of acquire-release, to ensure that the multiple clients of a shared process interact with the process in mutual exclusion from each other, gives rise to an obvious source of deadlocks, as acquire-release effectively amounts to a locking discipline. The typical approach to prevent deadlocks in that case is to impose a partial order on the resources and to *"lock-up"*, i.e., to lock the resources in ascending order. We adopted this strategy in Sect. 1 (Fig. 1) to break the cyclic dependencies among the acquires in the dining philosophers.

In Sect. 1, however, we also considered another example (Fig. 2) and discovered that *cyclic acquisitions* are not the only source of deadlocks, but deadlocks can also arise from *interdependent acquisitions and synchronizations*. In that example, we can prevent the deadlock by moving the acquire past the synchronization, in either of the two processes. Whereas in a purely linear session-typed system the sequencing of actions within a process do not affect other processes, the relative placement of acquire requests and synchronizations become relevant in a shared session-typed system.

Based on this observation, we can divide the processes in a shared-session discipline into *competitors* and *collaborators*. The former compete for a set of resources, whereas the latter do not overlap in the set of resources they acquire. For example, in the dining philosophers (Fig. 1), the philosophers p_0, p_1, and p_2 compete with each other for the set of forks f_0, f_1, and f_2, whereas the process that spawns the philosophers and the forks collaborates with either of them.

Transferring this idea to the process graph that emerges at run-time, we note that competitors are siblings whereas collaborators stand in a parent-descendant relationship. We illustrate this outcome on Fig. 3 that shows a possible run-time process graph for the dining philosophers. Linear processes are depicted as solid black circles with a white identifier and shared processes are depicted as dotted filled violet circles with a black identifier. Linear channels are depicted as black lines, shared channel references as dotted violet lines with the arrow head pointing to the shared process being acquired[1]. The identifiers P_0, P_1, and P_2 stand for the three philosophers, F_0, F_1, and F_2 for the three forks, and T for the process that sets the table. The current run-time graph depicts the scenario in which P_1 is eating, while the other two philosophers are still thinking.

Embedded in the graph is a *tree* that arises from the linear processes and the linear channels connecting them. For any two nodes in this tree, the *parent* node denotes the *client* process and the *child* node the *providing* process. We note that the *independence principle* (see Sect. 2), which precludes shared processes from depending on linear channel references, guarantees that there exists exactly one tree in the process graph, with the linear main process as its root. The shape of the tree changes when new processes are spawned, linear channels exchanged (through $\otimes$ and $\multimap$), or shared processes acquired. For example, process P_2 could acquire the shared fork F_0, which then becomes a linear child process of P_2, should the acquire succeed. As indicated by the shared channel references, the

[1] We have made sure to make the different concepts distinguishable in greyscale mode.

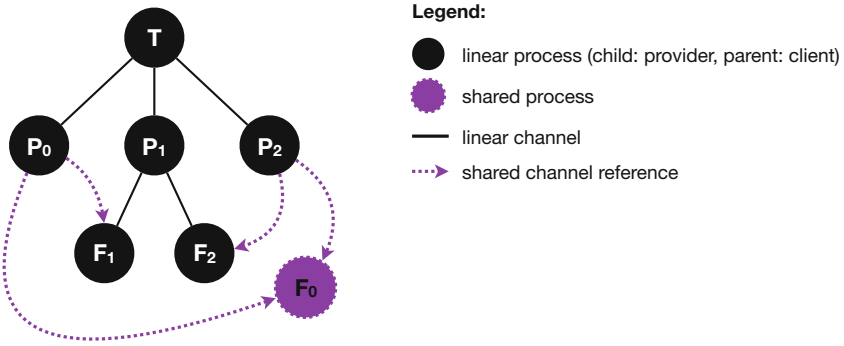


Fig. 3. Run-time process graph for dining philosophers (see Fig. 1).

sibling nodes P_0, P_1, and P_2 compete with each other for the nodes F_0, F_1, and F_2, whereas the node T does not compete for any of the resources acquired by its *descendants* (including F_1 and F_2). Our type system enforces this paradigm, as we discuss in the next section.

3.2 Type System

Invariants. Having identified the notions of *collaborators* and *competitors*, our type system must guarantee: *(i)* that collaborators acquire mutually disjoint sets of resources; *(ii)* that competitors employ a locking-up strategy for the resources they share; and, *(iii)* that competitors have released all acquired resources when synchronizing with other competitors. Invariant *(ii)* rules out cyclic acquisitions and invariants *(i)* and *(iii)* combined rule out interdependent acquisitions and synchronizations.

To express the high-level invariants above in our type system, we introduce the notion of a *world* – an abstract value that is equipped with a partial order – and associate such a world with every process. Programmers can *create* worlds, indicate the world at which a process resides at spawn time, and define an *order* on worlds. Moreover, we associate with each process a *range of worlds* that indicates the worlds of resources that the process may acquire. As a result, we obtain the following typing judgments:

$$\Psi;\, \Gamma \vdash P :: (x_{\mathsf{S}} : A_{\mathsf{S}}[\omega_k \updownarrow_{\omega_l}^{\omega_n}]) \quad (\text{where } \Psi^+ \text{ irreflexive})$$

$$\Psi;\, \Gamma;\, \Phi;\, \Delta \vdash P :: (x_{\mathsf{L}} : A_{\mathsf{L}}[\omega_k \updownarrow_{\omega_l}^{\omega_n}]) \quad (\text{where } \Psi^+ \text{ irreflexive})$$

The typing judgments reveal that we impose worlds at the *judgmental level*, resulting in a *hybrid system*, in which the adjoint modalities for acquire-release are complemented with world modalities that occur as *syntactic objects* in propositions [7]. We use the notation $x_m : A_m[\omega_k \updownarrow_{\omega_l}^{\omega_n}]$ (where m stands for S or L) to associate worlds ω_k, ω_l, and ω_n with a process that offers a session of type A_m along channel x. World ω_k denotes the world at which the process resides.

We refer to this world as the *self* world. Worlds ω_l and ω_n indicate the range of worlds of resources that the process may acquire, with ω_l denoting the *minimal (min)* world in this range and ω_n the *maximal (max)* one.

Process terms are typed relative to the order specified in Ψ and the contexts Γ, Φ, and Δ. As in Sect. 2, Γ is a structural context consisting of hypotheses on the typing of variables bound to shared channel references, augmented with world annotations. We find it necessary to split the linear context "Δ" from Sect. 2 into the two disjoint contexts Φ and Δ, allowing us to separate channels that are possibly aliased (due to sharing) from those that are not, respectively. Both Φ and Δ consist of hypotheses on the typing of variables that are bound to linear channels, augmented with world annotations. Ψ is presupposed to be *acyclic* and defined as: $\Psi \triangleq \cdot \mid \Psi',\ \omega_k < \omega_l \mid \Psi',\ \omega_o$, where ω stands for a concrete *world* w or a *world variable* δ. We allow Ψ to contain single worlds, to support singletons as well as to accommodate world creation prior to order declaration. We define the transitive closure Ψ^+, yielding a *strict partial order*, and the reflexive transitive closure Ψ^*, yielding a *partial order*.

The high-level invariants *(i)*, *(ii)*, and *(iii)* identified earlier naturally transcribe into the following invariants, which we impose on the typing judgments above. We use the notation $\_\langle x_m \rangle; P$ to denote a process term that currently executes an action along channel x_m.

1. $\mathsf{min(parent)} \leq \mathsf{self(acquired\_child)} \leq \mathsf{max(parent)}$:
   $\forall y_{\mathsf{L}} : B_{\mathsf{L}}[\omega_o \updownarrow^{\omega_r}_{\omega_p}] \in \Phi : \Psi^* \vdash \omega_l \leq \omega_o \leq \omega_n$
2. $\mathsf{max(parent)} < \mathsf{min(child)}$:
   $\forall y_{\mathsf{L}} : B_{\mathsf{L}}[\omega_o \updownarrow^{\omega_r}_{\omega_p}] \in \Delta \cup \Phi : \Psi^+ \vdash \omega_n < \omega_p$
3. If $\Psi;\ \Gamma, x_{\mathsf{S}} : A[\omega_t \updownarrow^{\omega_v}_{\omega_u}];\ \Phi;\ \Delta \vdash x_{\mathsf{L}} \leftarrow \mathsf{acquire}\ x_{\mathsf{S}};\ Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_k \updownarrow^{\omega_n}_{\omega_l}])$, then
   $\forall y_{\mathsf{L}} : B_{\mathsf{L}}[\omega_o \updownarrow^{\omega_r}_{\omega_p}] \in \Phi : \Psi^+ \vdash \omega_o < \omega_t$.
4. If $\Psi;\ \Gamma;\ \Phi;\ \Delta \vdash \_\langle x_m \rangle; P :: (x_{\mathsf{L}} : A_{\mathsf{L}}[\omega_k \updownarrow^{\omega_n}_{\omega_l}])$, then $\Phi = (\cdot)$.

Invariants 1 and 2 ensure that, for any node in the tree, the acquired resources reside at smaller worlds than those acquired by any descendant. As a result, the two invariants guarantee high-level invariant *(i)*. Invariant 3, on the other hand, imposes a lock-up strategy on acquires and thus guarantees high-level invariant *(ii)*. To guarantee high-level invariant *(iii)*, we impose Invariant 4, which forces a process to release any acquired resources before communicating along its offering channel. Since sibling nodes cannot be directly connected by a linear channel, the only way for them to synchronize is through a common parent. Finally, to guarantee that world annotations are internally consistent, we require for each annotation $[\omega_k \updownarrow^{\omega_n}_{\omega_l}]$ that $\omega_k < \omega_l \leq \omega_n$.

Rules. We now present select process typing rules, a complete listing is provided in the companion technical report [4]. The only new rules with respect to the language $\mathsf{SILL}_{\mathsf{S}}$ [2] are those pertaining to world creation and order determination. These are extra-logical judgmental rules. We allow both linear and shared processes to create and relate worlds. Rules (T-New$_{\mathsf{L}}$) and (T-New$_{\mathsf{S}}$) create a new world w and make it available to the continuation Q_{w}. Rules (T-Ord$_{\mathsf{L}}$) and (T-Ord$_{\mathsf{S}}$) relate two existing worlds, while preserving acyclicity of the order.

$$\frac{\Psi, \mathsf{w};\ \Gamma;\ \Phi;\ \Delta \vdash Q_{\mathsf{w}} :: (x_{\mathsf{L}} : A_{\mathsf{L}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}{\Psi;\ \Gamma;\ \Phi;\ \Delta \vdash \mathsf{w} \leftarrow \mathsf{new\_world};\ Q_{\mathsf{w}} :: (x_{\mathsf{L}} : A_{\mathsf{L}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}\ (\text{T-NEW}_{\mathsf{L}})$$

$$\frac{\Psi, \mathsf{w};\ \Gamma \vdash Q_{\mathsf{w}} :: (x_{\mathsf{S}} : A_{\mathsf{S}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}{\Psi;\ \Gamma \vdash \mathsf{w} \leftarrow \mathsf{new\_world};\ Q_{\mathsf{w}} :: (x_{\mathsf{S}} : A_{\mathsf{S}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}\ (\text{T-NEW}_{\mathsf{S}})$$

$$\frac{\omega_p, \omega_r \in \Psi \qquad (\Psi, \omega_p < \omega_r)^+ \text{ irreflexive} \qquad \Psi, \omega_p < \omega_r;\ \Gamma;\ \Phi;\ \Delta \vdash Q :: (x_{\mathsf{L}} : A_{\mathsf{L}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}{\Psi;\ \Gamma;\ \Phi;\ \Delta \vdash \omega_p < \omega_r;\ Q :: (x_{\mathsf{L}} : A_{\mathsf{L}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}\ (\text{T-ORD}_{\mathsf{L}})$$

$$\frac{\omega_p, \omega_r \in \Psi \qquad (\Psi, \omega_p < \omega_r)^+ \text{ irreflexive} \qquad \Psi, \omega_p < \omega_r;\ \Gamma \vdash Q :: (x_{\mathsf{S}} : A_{\mathsf{S}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}{\Psi;\ \Gamma \vdash \omega_p < \omega_r;\ Q :: (x_{\mathsf{S}} : A_{\mathsf{S}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}\ (\text{T-ORD}_{\mathsf{S}})$$

We now consider the typing rule for acquire, which must explicitly enforce the various low-level invariants above. Since an acquire results in the addition of a new child node to the executing process, the rule can interfere with Invariants 1 and 2. The first two premises of the rule ensure that the two invariants are preserved. Moreover, the rule has to ensure that the acquiring process is locking-up (Invariant 3), which is achieved by the third premise.

$$\frac{\Psi^* \vdash \omega_k \leq \omega_m \leq \omega_n \qquad \Psi^+ \vdash \omega_n < \omega_u \qquad \forall y_{\mathsf{L}} : B_{\mathsf{L}}[\omega_l \uparrow^{\omega_r}_{\omega_p}] \in \Phi : \omega_l < \omega_m \qquad \Psi;\ \Gamma, x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}[\omega_m \uparrow^{\omega_v}_{\omega_u}];\ \Phi, x_{\mathsf{L}} : A_{\mathsf{L}}[\omega_m \uparrow^{\omega_v}_{\omega_u}];\ \Delta \vdash Q_{x_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \uparrow^{\omega_n}_{\omega_k}])}{\Psi;\ \Gamma, x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}[\omega_m \uparrow^{\omega_v}_{\omega_u}];\ \Phi;\ \Delta \vdash x_{\mathsf{L}} \leftarrow \mathsf{acquire}\, x_{\mathsf{S}}\,;\, Q_{x_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \uparrow^{\omega_n}_{\omega_k}])}\ (\text{T-}\uparrow^{\mathsf{S}}_{\mathsf{L}}\text{L})$$

The remaining shift rules are actually *unchanged* with respect to $\mathsf{SILL}_{\mathsf{S}}$, modulo the world annotations. In particular, low-level Invariant 4 is already satisfied because the conclusion of rule (T-$\uparrow^{\mathsf{S}}_{\mathsf{L}}$R) does not have a context Φ and because the independence principle forces Φ to be empty in rule (T-$\downarrow^{\mathsf{S}}_{\mathsf{L}}$R).

$$\frac{\Psi;\ \Gamma;\ \cdot\,;\ \cdot \vdash P_{x_{\mathsf{L}}} :: (x_{\mathsf{L}} : A_{\mathsf{L}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}{\Psi;\ \Gamma \vdash x_{\mathsf{L}} \leftarrow \mathsf{accept}\, x_{\mathsf{S}}\,;\, P_{x_{\mathsf{L}}} :: (x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}\ (\text{T-}\uparrow^{\mathsf{S}}_{\mathsf{L}}\text{R})$$

$$\frac{\Psi;\ \Gamma, x_{\mathsf{S}} : A_{\mathsf{S}}[\omega_m \uparrow^{\omega_v}_{\omega_u}];\ \Phi;\ \Delta \vdash Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \uparrow^{\omega_n}_{\omega_k}])}{\Psi;\ \Gamma;\ \Phi, x_{\mathsf{L}} : \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}}[\omega_m \uparrow^{\omega_v}_{\omega_u}];\ \Delta \vdash x_{\mathsf{S}} \leftarrow \mathsf{release}\, x_{\mathsf{L}}\,;\, Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \uparrow^{\omega_n}_{\omega_k}])}\ (\text{T-}\downarrow^{\mathsf{S}}_{\mathsf{L}}\text{L})$$

$$\frac{\Psi;\ \Gamma \vdash P_{x_{\mathsf{S}}} :: (x_{\mathsf{S}} : A_{\mathsf{S}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}{\Psi;\ \Gamma;\ \cdot\,;\ \cdot \vdash x_{\mathsf{S}} \leftarrow \mathsf{detach}\, x_{\mathsf{L}}\,;\, P_{x_{\mathsf{S}}} :: (x_{\mathsf{L}} : \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}}[\omega_m \uparrow^{\omega_v}_{\omega_u}])}\ (\text{T-}\downarrow^{\mathsf{S}}_{\mathsf{L}}\text{R})$$

We now consider the linear connectives, starting with $\mathbf{1}$. Rule (T-$\mathbf{1}_{\mathrm{L}}$) reveals that only processes that have never been acquired may be terminated. This restriction is important to guarantee progress because existing clients of a shared process may wait indefinitely otherwise. We impose the restriction as a well-formedness condition on a session type, giving rise to a *strictly equi-synchronizing* session type. The notion of an *equi-synchronizing* session type [2] has been defined for $\mathsf{SILL}_{\mathsf{S}}$ and guarantees that a process that has been acquired at a type A_{s} is released back to the type A_{s}, should it ever be released. A *strictly* equi-synchronizing session type additionally requires that an acquired resource *must* be released. The corresponding rules can be found in [4]. Linearity enforces Invariant 4 in rule (T-$\mathbf{1}_{\mathrm{R}}$), making sure that no linear channels are left behind.

$$\frac{\Psi;\ \Gamma;\ \Phi;\ \Delta \vdash Q :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow_{\omega_k}^{\omega_n}])}{\Psi;\ \Gamma;\ \Phi;\ \Delta, x_{\mathsf{L}} : \mathbf{1}[\omega_m \updownarrow_{\omega_u}^{\omega_v}] \vdash \mathsf{wait}\, x_{\mathsf{L}}\,;Q :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow_{\omega_k}^{\omega_n}])}\ (\text{T-}\mathbf{1}_{\mathrm{L}})$$

$$\frac{}{\Psi;\ \Gamma;\ \cdot\,;\ \cdot \vdash \mathsf{close}\, x_{\mathsf{L}} :: (x_{\mathsf{L}} : \mathbf{1}[\omega_m \updownarrow_{\omega_u}^{\omega_v}])}\ (\text{T-}\mathbf{1}_{\mathrm{R}})$$

Next, we consider internal and external choice. Since internal and external choice cannot alter the linear process tree of a process graph, the rules are very similar to the ones in SILL$_{\mathsf{S}}$. The only differences are that we get two left rules for each connective and that the Φ-context of each right rule must be empty to satisfy Invariant 4. The former is merely due to the tracking of possibly aliased sessions in the Φ context. We only list rules for internal choice, those for external choice are dual and can be found in [4].

$$\frac{(\forall i)\ \ \Psi;\ \Gamma;\ \Phi;\ \Delta, x_{\mathsf{L}} : A_{\mathsf{L}_i}[\omega_m \updownarrow_{\omega_u}^{\omega_v}] \vdash Q_i :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow_{\omega_k}^{\omega_n}])}{\Psi;\ \Gamma;\ \Phi;\ \Delta, x_{\mathsf{L}} : \oplus\{\overline{l : A_{\mathsf{L}}}\}[\omega_m \updownarrow_{\omega_u}^{\omega_v}] \vdash \mathsf{case}\ x_{\mathsf{L}}\ \mathsf{of}\ \overline{l \Rightarrow Q} :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow_{\omega_k}^{\omega_n}])}\ (\text{T-}\oplus_{\mathrm{L}_1})$$

$$\frac{(\forall i)\ \ \Psi;\ \Gamma;\ \Phi, x_{\mathsf{L}} : A_{\mathsf{L}_i}[\omega_m \updownarrow_{\omega_u}^{\omega_v}];\ \Delta \vdash Q_i :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow_{\omega_k}^{\omega_n}])}{\Psi;\ \Gamma;\ \Phi, x_{\mathsf{L}} : \oplus\{\overline{l : A_{\mathsf{L}}}\}[\omega_m \updownarrow_{\omega_u}^{\omega_v}];\ \Delta \vdash \mathsf{case}\ x_{\mathsf{L}}\ \mathsf{of}\ \overline{l \Rightarrow Q} :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow_{\omega_k}^{\omega_n}])}\ (\text{T-}\oplus_{\mathrm{L}_2})$$

$$\frac{\Psi;\ \Gamma;\ \cdot\,;\ \Delta \vdash P :: (x_{\mathsf{L}} : A_{\mathsf{L}\,h}[\omega_m \updownarrow_{\omega_u}^{\omega_v}])}{\Psi;\ \Gamma;\ \cdot\,;\ \Delta \vdash x_{\mathsf{L}}.l_h\,;P :: (x_{\mathsf{L}} : \oplus\{\overline{l : A_{\mathsf{L}}}\}[\omega_m \updownarrow_{\omega_u}^{\omega_v}])}\ (\text{T-}\oplus_{\mathrm{R}})$$

More interesting are linear channel output and input, since these alter the linear process tree of a process graph. Moreover, additional world annotations are needed to indicate the worlds of the channel that is exchanged. For the latter we use the notation $@\omega_l \updownarrow_{\omega_p}^{\omega_r}$, indicating that the exchanged channel has the worlds ω_l, ω_p, and ω_r for self, min, and max, respectively. To account for induced changes in the process graph, the rules that type an input of a linear channel must guard against any disturbance of Invariants 1 and 2. Because the two invariants guarantee that parents do not overlap with their descendants in terms of acquired resources, they prevent any exchange of acquired channels. We thus restrict $\otimes$ and $\multimap$ to the exchange of channels that have not yet been acquired. This is not a limitation since, as we will see below, shared channel output and input are unrestricted.

Even with the above restriction in place, we still have to make sure that a received channel satisfies Invariant 2. If we were to state a corresponding premise on the receiving rules, invertibility of the rules would be disturbed. To uphold invertibility, we impose a well-formedness condition on session types that ensures for a session of type $A_{\mathsf{L}}@\omega_l \updownarrow_{\omega_p}^{\omega_r} \otimes B_{\mathsf{L}}[\omega_m \updownarrow_{\omega_u}^{\omega_v}]$ that $\omega_v < \omega_p$ and, analogously, for a session of type $A_{\mathsf{L}}@\omega_l \updownarrow_{\omega_p}^{\omega_r} \multimap B_{\mathsf{L}}[\omega_m \updownarrow_{\omega_u}^{\omega_v}]$ that $\omega_v < \omega_p$. Session types are checked to be well-formed upon process definition. Given type well-formedness, we obtain the following rules for $\multimap$, noting that the right rule enforces Invariant 4 by requiring an empty Φ-context. The rules for $\otimes$ are dual.

$$\frac{\Psi;\ \Gamma;\ \Phi;\ \Delta, x_{\mathsf{L}} : B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}] \vdash Q :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}])}{\Psi;\ \Gamma;\ \Phi;\ \Delta, x_{\mathsf{L}} : A_{\mathsf{L}}@\omega_l \updownarrow^{\omega_r}_{\omega_p} \multimap B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}], y_{\mathsf{L}} : A_{\mathsf{L}}[\omega_l \updownarrow^{\omega_r}_{\omega_p}] \vdash \mathsf{send}\ x_{\mathsf{L}}\ y_{\mathsf{L}}\ ;\ Q :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}])}\ (\text{T-}\multimap_{\mathrm{L}_1})$$

$$\frac{\Psi;\ \Gamma;\ \Phi, x_{\mathsf{L}} : B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}];\ \Delta \vdash Q :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}])}{\Psi;\ \Gamma;\ \Phi, x_{\mathsf{L}} : A_{\mathsf{L}}@\omega_l \updownarrow^{\omega_r}_{\omega_p} \multimap B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}];\ \Delta, y_{\mathsf{L}} : A_{\mathsf{L}}[\omega_l \updownarrow^{\omega_r}_{\omega_p}] \vdash \mathsf{send}\ x_{\mathsf{L}}\ y_{\mathsf{L}}\ ;\ Q :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}])}\ (\text{T-}\multimap_{\mathrm{L}_2})$$

$$\frac{\Psi;\ \Gamma;\ \cdot\ ;\ \Delta, y_{\mathsf{L}} : A_{\mathsf{L}}[\omega_l \updownarrow^{\omega_r}_{\omega_p}] \vdash P_{y_{\mathsf{L}}} :: (x_{\mathsf{L}} : B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}])}{\Psi;\ \Gamma;\ \cdot\ ;\ \Delta \vdash y_{\mathsf{L}} \leftarrow \mathsf{recv}\ x_{\mathsf{L}}\ ;\ P_{y_{\mathsf{L}}} :: (x_{\mathsf{L}} : A_{\mathsf{L}}@\omega_l \updownarrow^{\omega_r}_{\omega_p} \multimap B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}])}\ (\text{T-}\multimap_{\mathrm{R}})$$

Since there are no invariants imposed on the shared context Γ, the rules for shared channel output and input are identical to those in $\mathsf{SILL}_{\mathsf{S}}$. The only differences are that we have two left rules and that the Φ-context of the right rule must be empty to satisfy Invariant 4. The former is merely due to the tracking of possibly aliased sessions in the Φ context.

$$\frac{\Psi;\ \Gamma, y_{\mathsf{S}} : A_{\mathsf{S}}[\omega_l \updownarrow^{\omega_r}_{\omega_p}];\Phi;\ \Delta, x_{\mathsf{L}} : B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}] \vdash Q_{y_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}])}{\Psi;\ \Gamma;\Phi;\ \Delta, x_{\mathsf{L}} : \exists x{:}A_{\mathsf{S}}@\omega_l \updownarrow^{\omega_r}_{\omega_p}\ .\ B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}] \vdash y_{\mathsf{S}} \leftarrow \mathsf{recv}\ x_{\mathsf{L}}\ ;Q_{y_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}])}\ (\text{T-}\exists_{\mathrm{L}_1})$$

$$\frac{\Psi;\ \Gamma, y_{\mathsf{S}} : A_{\mathsf{S}}[\omega_l \updownarrow^{\omega_r}_{\omega_p}];\Phi, x_{\mathsf{L}} : B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}];\ \Delta \vdash Q_{y_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}])}{\Psi;\ \Gamma;\Phi, x_{\mathsf{L}} : \exists x{:}A_{\mathsf{S}}@\omega_l \updownarrow^{\omega_r}_{\omega_p}\ .\ B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}];\ \Delta \vdash y_{\mathsf{S}} \leftarrow \mathsf{recv}\ x_{\mathsf{L}}\ ;Q_{y_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}])}\ (\text{T-}\exists_{\mathrm{L}_2})$$

$$\frac{\Psi;\ \Gamma, y_{\mathsf{S}} : A_{\mathsf{S}}[\omega_l \updownarrow^{\omega_r}_{\omega_p}];\cdot\ ;\ \Delta \vdash P :: (x_{\mathsf{L}} : B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}])}{\Psi;\ \Gamma, y_{\mathsf{S}} : A_{\mathsf{S}}[\omega_l \updownarrow^{\omega_r}_{\omega_p}];\cdot\ ;\ \Delta \vdash \mathsf{send}\ x_{\mathsf{L}}\ y_{\mathsf{S}}\ ;\ P :: (x_{\mathsf{L}} : \exists x{:}A_{\mathsf{S}}@\omega_l \updownarrow^{\omega_r}_{\omega_p}\ .\ B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}])}\ (\text{T-}\exists_{\mathrm{R}})$$

We finally consider the rules for forwarding and spawning. We allow a shared forward between processes that offer the same session at the same worlds. Because forwards have to be *world-invariant*, however, no well-typed program could ever have a linear forward. The process being forwarded to must be in either of the contexts Φ or Δ, and thus satisfies Invariant 2, making it impossible for the world annotations of the forwarder and forwardee to match. We omit linear forwarding and discuss possible future extensions in Sect. 6.

$$\frac{}{\Psi;\ \Gamma, y_{\mathsf{S}} : A_{\mathsf{S}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}] \vdash \mathsf{fwd}\ x_{\mathsf{S}}\ y_{\mathsf{S}} :: (x_{\mathsf{S}} : A_{\mathsf{S}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}])}\ (\text{T-Id}_{\mathsf{S}})$$

The rules for spawning depend on the possible modes of the spawning and spawned processes: (T-Spawn$_{\mathsf{LL}}$) specifies how a linear process can spawn another linear process; (T-Spawn$_{\mathsf{SS}}$) specifies how a shared processes can spawn another shared process. The rules are checked relative to a process definition found in the signature Σ and to a world substitution mapping $\gamma : |\Psi| \to |\Psi'|$, such that for each $\delta \in \Psi'$ we have $\Psi \vdash \gamma(\delta)$, where $|\Psi|$ denotes the *field* of Ψ (i.e., the union of its domain and range). As usual, we lift substitution to types $\hat{\gamma}(A_m)$, contexts $\hat{\gamma}(\Gamma)$, and orders $\hat{\gamma}(\Psi)$. Both rules ensure that, given the mapping γ, the order Ψ of the spawning process entails the one of the process definition ($\Psi \vdash \hat{\gamma}(\Psi')$). The linear spawn rule (T-Spawn$_{\mathsf{LL}}$) further enforces Invariant 2 for the spawned child. We note that the spawned child enters the linear context Δ in the spawning process' continuation since no aliases to such a process can exist at this point.

$$\frac{\begin{array}{c}\Delta_1 = \overline{y_{\mathsf{L}} : B_{\mathsf{L}}[\omega_m \updownarrow^{\omega_v}_{\omega_u}]} \qquad \Phi_1 = \overline{\tilde{y}_{\mathsf{L}} : \tilde{B}_{\mathsf{L}}[\tilde{\omega}_m \updownarrow^{\tilde{\omega}_v}_{\tilde{\omega}_u}]} \qquad \Gamma_1 = \overline{z_{\mathsf{S}} : C_{\mathsf{S}}[\omega_l \updownarrow^{\omega_r}_{\omega_p}]} \\ (\Psi' \vdash x'_{\mathsf{L}} : A'_{\mathsf{L}}[\delta_j \updownarrow^{\delta_n}_{\delta_k}] \leftarrow X_{\mathsf{L}} \leftarrow \Delta', \Phi', \Gamma' = P_{x'_{\mathsf{L}}, \mathsf{dom}(\Delta'), \mathsf{dom}(\Phi'), \mathsf{dom}(\Gamma'), \Psi''}) \in \Sigma \\ \hat{\gamma}(A'_{\mathsf{L}}[\delta_j \updownarrow^{\delta_n}_{\delta_k}]) = A_{\mathsf{L}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}] \qquad \hat{\gamma}(\Delta') = \Delta_1 \qquad \hat{\gamma}(\Phi') = \Phi_1 \qquad \hat{\gamma}(\Gamma') = \Gamma_1 \qquad \Psi \vdash \hat{\gamma}(\Psi') \\ \Psi^+ \vdash \omega_t < \omega_k \\ \Psi;\ \Gamma_1, \Gamma_2;\ \Phi_2;\ \Delta_2, x_{\mathsf{L}} : A_{\mathsf{L}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}] \vdash Q_{x_{\mathsf{L}}} :: (z'_{\mathsf{L}} : D_{\mathsf{L}}[\omega_i \updownarrow^{\omega_t}_{\omega_q}])\end{array}}{\Psi;\ \Gamma_1, \Gamma_2;\ \Phi_1, \Phi_2;\ \Delta_1, \Delta_2 \vdash x_{\mathsf{L}} : A_{\mathsf{L}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}] \leftarrow X_{\mathsf{L}} \leftarrow \overline{y_{\mathsf{L}}}, \overline{\tilde{y}_{\mathsf{L}}}, \overline{z_{\mathsf{S}}};\ Q_{x_{\mathsf{L}}} :: (z'_{\mathsf{L}} : D_{\mathsf{L}}[\omega_i \updownarrow^{\omega_t}_{\omega_q}])} \ (\text{T-SPAWN}_{\mathsf{LL}})$$

$$\frac{\begin{array}{c}\Gamma_1 = \overline{z_{\mathsf{S}} : C_{\mathsf{S}}[\omega_l \updownarrow^{\omega_r}_{\omega_p}]} \qquad (\Psi' \vdash x'_{\mathsf{S}} : A'_{\mathsf{S}}[\delta_j \updownarrow^{\delta_n}_{\delta_k}] \leftarrow X_{\mathsf{S}} \leftarrow \Gamma' = P_{x'_{\mathsf{S}}, \mathsf{dom}(\Gamma'), \Psi''}) \in \Sigma \\ \hat{\gamma}(A'_{\mathsf{S}}[\delta_j \updownarrow^{\delta_n}_{\delta_k}]) = A_{\mathsf{S}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}] \qquad \hat{\gamma}(\Gamma') = \Gamma_1 \qquad \Psi \vdash \hat{\gamma}(\Psi') \\ \Psi;\ \Gamma_1, \Gamma_2, x_{\mathsf{S}} : A_{\mathsf{S}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}] \vdash Q_{x_{\mathsf{S}}} :: (z'_{\mathsf{S}} : D_{\mathsf{S}}[\omega_i \updownarrow^{\omega_t}_{\omega_q}])\end{array}}{\Psi;\ \Gamma_1, \Gamma_2 \vdash x_{\mathsf{S}} : A_{\mathsf{S}}[\omega_j \updownarrow^{\omega_n}_{\omega_k}] \leftarrow X_{\mathsf{S}} \leftarrow \overline{z_{\mathsf{S}}};\ Q_{x_{\mathsf{S}}} :: (z'_{\mathsf{S}} : D_{\mathsf{S}}[\omega_i \updownarrow^{\omega_t}_{\omega_q}])} \ (\text{T-SPAWN}_{\mathsf{SS}})$$

In the companion technical report [4], we provide a variant of rule (T-SPAWN$_{\mathsf{LL}}$) for the case of a linear recursive tail call. Without linear forwarding, a linear tail call can no longer be implicitly "de-sugared" into a spawn and a linear forward [2,22,52], but must be accounted for explicitly. In the report, we also provide the rules for checking process definitions. Those rules make sure that the process' world order is acyclic, that the types of the providing session and argument sessions are well-formed, and that the process satisfies Invariants 1 and 2.

3.3 Dining Philosophers in SILL$_{\mathsf{S}+}$

Having introduced our type system, we revisit the dining philosophers from Sect. 1 and show how to program the example in SILL$_{\mathsf{S}+}$, ensuring that the program will run without deadlocks. The code is given in Fig. 4. We note the world annotations in the signature of the process definitions. For instance,

$$\mathit{thinking} : \{\delta_0 < \delta_1, \delta_1 < \delta_2, \delta_2 < \delta_3 \vdash \mathsf{phil}[\delta_0 \updownarrow^{\delta_2}_{\delta_1}] \leftarrow \mathsf{sfork}[\delta_1 \updownarrow^{\delta_3}_{\delta_3}], \mathsf{sfork}[\delta_2 \updownarrow^{\delta_3}_{\delta_3}]; \cdot; \cdot\}$$

indicates that, given the order $\delta_0 < \delta_1 < \delta_2 < \delta_3$, process *thinking* provides a session of type $\mathsf{phil}[\delta_0 \updownarrow^{\delta_2}_{\delta_1}]$ and uses two shared channel references of type $\mathsf{sfork}[\delta_1 \updownarrow^{\delta_3}_{\delta_3}]$ and $\mathsf{sfork}[\delta_2 \updownarrow^{\delta_3}_{\delta_3}]$. The two $\cdot$ signify that neither acquired nor linear channel references are given as arguments. The signature indicates that the two shared fork references reside at different worlds, such that the world of the first one is smaller than the one of the second.

Let's briefly convince ourselves that the two acquires in process *thinking* in Fig. 4 are type-correct. For each acquire we have to show that: the world of the resource to be acquired is within the acquiring process' range; the max of the acquiring process is smaller than the min of the acquired resource; and, that the self of the acquired resource is larger than those of all already acquired resources. We can convince ourselves that all those conditions are readily met.

$$
\begin{array}{l}
\mathit{thinking} : \{\delta_0 < \delta_1, \delta_1 < \delta_2, \delta_2 < \delta_3 \vdash \\
\quad \mathsf{phil}[\delta_0 \uparrow^{\delta_2}_{\delta_1}] \leftarrow \mathsf{sfork}[\delta_1 \uparrow^{\delta_3}_{\delta_3}], \mathsf{sfork}[\delta_2 \uparrow^{\delta_3}_{\delta_3}]; \cdot; \cdot\} \\
c[\delta_0 \uparrow^{\delta_2}_{\delta_1}] \leftarrow \mathit{thinking} \leftarrow \mathit{left}[\delta_1 \uparrow^{\delta_3}_{\delta_3}], \mathit{right}[\delta_2 \uparrow^{\delta_3}_{\delta_3}] = \\
\quad \mathit{left}' \leftarrow \mathsf{acquire}\ \mathit{left}\ ; \\
\quad \mathit{right}' \leftarrow \mathsf{acquire}\ \mathit{right}\ ; \\
\quad c \leftarrow \mathit{eating} \leftarrow \mathit{left}', \mathit{right}'\ ; \\
\mathit{eating} : \{\delta_0 < \delta_1, \delta_1 < \delta_2, \delta_2 < \delta_3 \vdash \\
\quad \mathsf{phil}[\delta_0 \uparrow^{\delta_2}_{\delta_1}] \leftarrow \cdot; \mathsf{lfork}[\delta_1 \uparrow^{\delta_3}_{\delta_3}], \mathsf{lfork}[\delta_2 \uparrow^{\delta_3}_{\delta_3}]; \cdot\} \\
c[\delta_0 \uparrow^{\delta_2}_{\delta_1}] \leftarrow \mathit{eating} \leftarrow \mathit{left}'[\delta_1 \uparrow^{\delta_3}_{\delta_3}], \mathit{right}'[\delta_2 \uparrow^{\delta_3}_{\delta_3}] = \\
\quad \mathit{right} \leftarrow \mathsf{release}\ \mathit{right}'\ ; \\
\quad \mathit{left} \leftarrow \mathsf{release}\ \mathit{left}'\ ; \\
\quad c \leftarrow \mathit{thinking} \leftarrow \mathit{left}, \mathit{right}
\end{array}
\qquad
\begin{array}{l}
\mathsf{lfork} = \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{sfork} \\
\mathsf{sfork} = \uparrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{lfork} \\
\mathsf{phil} = \mathbf{1} \\
\\
\mathit{fork\_proc} : \{\delta_0 < \delta_1 \vdash \mathsf{sfork}[\delta_0 \uparrow^{\delta_1}_{\delta_1}]\} \\
c[\delta_0 \uparrow^{\delta_1}_{\delta_1}] \leftarrow \mathit{fork\_proc} = \\
\quad c' \leftarrow \mathsf{accept}\ c\ ; \\
\quad c \leftarrow \mathsf{detach}\ c'\ ; \\
\quad c'' : \mathsf{sfork}[\delta_0 \uparrow^{\delta_1}_{\delta_1}] \leftarrow \mathit{fork\_proc}\ ; \\
\quad \mathsf{fwd}\ c\ c''
\end{array}
$$

Fig. 4. Deadlock-free version of dining philosophers in $\mathsf{SILL}_{\mathsf{S}^+}$.

We note, however, that if we were to swap the two acquires, the program would not type-check.

Let us once more set the table for three philosophers and three forks. We execute this code in a process with world annotations $[\delta_a \uparrow^{\delta_b}_{\delta_b}]$ such that $\delta_a < \delta_b$. We first create new worlds and define their order:

$$
\begin{array}{l}
\mathsf{w}_1 \leftarrow \mathsf{new\_world}; \mathsf{w}_2 \leftarrow \mathsf{new\_world}; \mathsf{w}_3 \leftarrow \mathsf{new\_world}; \mathsf{w}_4 \leftarrow \mathsf{new\_world}; \\
\delta_a < \mathsf{w}_1; \delta_a < \mathsf{w}_2; \delta_b < \mathsf{w}_1; \mathsf{w}_1 < \mathsf{w}_2; \mathsf{w}_1 < \mathsf{w}_3; \mathsf{w}_1 < \mathsf{w}_4; \mathsf{w}_2 < \mathsf{w}_3; \mathsf{w}_2 < \mathsf{w}_4; \mathsf{w}_3 < \mathsf{w}_4;
\end{array}
$$

We then spawn the forks, each residing at a different world, such that the max world of a fork is higher than the self of the highest fork, ensuring Invariant 2 for the philosopher processes that we spawn afterwards:

$$
\begin{array}{l}
f_1 : \mathsf{sfork}[\mathsf{w}_1 \uparrow^{\mathsf{w}_4}_{\mathsf{w}_4}] \leftarrow \mathit{fork\_proc}\ ;\ f_2 : \mathsf{sfork}[\mathsf{w}_2 \uparrow^{\mathsf{w}_4}_{\mathsf{w}_4}] \leftarrow \mathit{fork\_proc}\ ; \\
f_3 : \mathsf{sfork}[\mathsf{w}_3 \uparrow^{\mathsf{w}_4}_{\mathsf{w}_4}] \leftarrow \mathit{fork\_proc}\ ;
\end{array}
$$

When we spawn the philosophers, we ensure that P_0 is going to pick up fork F_1 and then F_2, P_1 is going to pick up F_2 and then F_3, and P_2 is going to pick up F_1 and then F_3.

$$
\begin{array}{l}
p_0 : \mathsf{phil}[\delta_a \uparrow^{\mathsf{w}_2}_{\mathsf{w}_1}] \leftarrow \mathit{thinking} \leftarrow \cdot; \cdot; f_1, f_2\ ;\ p_1 : \mathsf{phil}[\delta_a \uparrow^{\mathsf{w}_3}_{\mathsf{w}_2}] \leftarrow \mathit{thinking} \leftarrow \cdot; \cdot; f_2, f_3\ ; \\
p_2 : \mathsf{phil}[\delta_a \uparrow^{\mathsf{w}_3}_{\mathsf{w}_1}] \leftarrow \mathit{thinking} \leftarrow \cdot; \cdot; f_1, f_3\ ;
\end{array}
$$

We note that the deadlocking spawn

$$
p_2 : \mathsf{phil}[\delta_a \uparrow^{\mathsf{w}_3}_{\mathsf{w}_1}] \leftarrow \mathit{thinking} \leftarrow \cdot; \cdot; f_3, f_1\ ;
$$

is type-incorrect since we would substitute both w_1 and w_3 for δ_1 and w_3 and w_1 for δ_2, which violates the ordering constraints put in place by typing.

3.4 Dynamics

We now give the *dynamics* of $\mathsf{SILL}_{\mathsf{S}^+}$. Our current system is based on a *synchronous* dynamics. While this choice is more conservative, it allows us to narrow the complexity of the problem at hand.

As in $\mathsf{SILL}_{\mathsf{S}}$, we use *multiset rewriting rules* [12] to capture the dynamics of $\mathsf{SILL}_{\mathsf{S}^+}$ (see Sect. 2). Multiset rewriting rules represent computation in terms of local state transitions between configurations of processes, only mentioning the parts of a configuration they rewrite. We use the predicates $\mathsf{proc}(a_m, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P_{a_m})$ and $\mathsf{unavail}(a_{\mathsf{S}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}})$ to define the states of a configuration (see Sect. 5.1). The former denotes a process executing term P that provides along channel a_m at mode m with worlds $\mathsf{w}_{\mathsf{a}_1}$, $\mathsf{w}_{\mathsf{a}_2}$, and $\mathsf{w}_{\mathsf{a}_3}$ for self, min, and max, respectively. The latter acts as a placeholder for a shared process providing along channel a_{S} with worlds $\mathsf{w}_{\mathsf{a}_1}$, $\mathsf{w}_{\mathsf{a}_2}$, and $\mathsf{w}_{\mathsf{a}_3}$ for self, min, and max, respectively, that is currently unavailable. We note that since worlds are also run-time artifacts, they must occur as part of the state-defining predicates.

Fig. 5 lists selected rules of the dynamics. Since the rules remain largely the same as those of $\mathsf{SILL}_{\mathsf{S}}$, apart from the world annotations that are "threaded through" unchanged, we only discuss the rules that actually differ from the $\mathsf{SILL}_{\mathsf{S}}$ rules. The interested reader can find the remaining rules in the companion technical report [4].

(D-Spawn$_{\mathsf{LL}}$)
$\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, x_{\mathsf{L}} : A_{\mathsf{L}}[\mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}}] \leftarrow X_{\mathsf{L}} \leftarrow \overline{c_{\mathsf{L}}}, \overline{\overline{c_{\mathsf{L}}}}, \overline{d_{\mathsf{S}}}; Q_{x_{\mathsf{L}}}),$
$!\mathsf{def}(\Psi' \vdash x'_{\mathsf{L}} : A'_{\mathsf{L}}[\delta_j \updownarrow^{\delta_n}_{\delta_k}] \leftarrow X_{\mathsf{L}} \leftarrow \Delta', \Phi', \Gamma' = P_{x'_{\mathsf{L}}, \mathsf{dom}(\Delta'), \mathsf{dom}(\Phi'), \mathsf{dom}(\Gamma'), \Psi''})$
$\longrightarrow \mathsf{proc}(b_{\mathsf{L}}, \mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}}, [b_{\mathsf{L}}/x'_{\mathsf{L}}, \overline{c_{\mathsf{L}}}/\mathsf{dom}(\Delta'), \overline{\overline{c_{\mathsf{L}}}}/\mathsf{dom}(\Phi'), \overline{d_{\mathsf{S}}}/\mathsf{dom}(\Gamma')]\hat{\gamma}(P_{x'_{\mathsf{L}}, \mathsf{dom}(\Delta'), \mathsf{dom}(\Phi'), \mathsf{dom}(\Gamma'), \Psi''})),$
$\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, [b_{\mathsf{L}}/x_{\mathsf{L}}]Q_{x_{\mathsf{L}}}),$
$\mathsf{unavail}(b_{\mathsf{S}}, \mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}})$ (b *fresh*)

(D-New)
$\mathsf{proc}(a, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, \mathsf{w} \leftarrow \mathsf{new\_world}; Q_{\mathsf{w}}) \longrightarrow \mathsf{proc}(a, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, Q_{\mathsf{w}})$ (w *fresh*)

(D-Ord)
$\mathsf{proc}(a, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, \mathsf{w} < \mathsf{w}'; Q) \longrightarrow \mathsf{proc}(a, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, Q)$

Fig. 5. Selected multiset rewriting rules of $\mathsf{SILL}_{\mathsf{S}^+}$.

Noteworthy are the rules D-New and D-Ord for creating and relating worlds, respectively. Rule D-New creates a fresh world, which will be globally available in the configuration. Rule D-Ord, on the other hand, updates the configuration's order with the pair $\mathsf{w} < \mathsf{w}'$. Rule D-Spawn$_{\mathsf{LL}}$, lastly, substitutes actual worlds for world variables in the body of the spawned process, using the substitution mapping γ defined earlier. It relies on the existence of a corresponding definition predicate for each process definition contained in the signature Σ. We note that the substitution γ in rule D-Spawn$_{\mathsf{LL}}$ instantiates the appropriate world variables in the spawned process P.

4 Extended Example: An Imperative Shared Queue

We now develop a typical imperative-style implementation of a queue that uses a list data structure internally to store the queue's elements and has shared references to the front and the back of the list for concurrent dequeueing and enqueueing, respectively. The session types for the queue and the list are[2]

$$\begin{aligned}\mathsf{queue}\ A_\mathsf{s} = \uparrow^\mathsf{s}_\mathsf{L} \& \{&\mathsf{enq} : \Pi x{:}A_\mathsf{s}.\ \downarrow^\mathsf{s}_\mathsf{L} \mathsf{queue}\ A_\mathsf{s},\\ &\mathsf{deq} : \oplus\{\mathsf{none} : \downarrow^\mathsf{s}_\mathsf{L} \mathsf{queue}\ A_\mathsf{s}, \mathsf{some} : \exists x{:}A_\mathsf{s}.\ \downarrow^\mathsf{s}_\mathsf{L} \mathsf{queue}\ A_\mathsf{s}\}\}\end{aligned}$$

$$\begin{aligned}\mathsf{list}\ A_\mathsf{s} = \uparrow^\mathsf{s}_\mathsf{L} \& \{&\mathsf{ins} : \Pi x{:}A_\mathsf{s}.\ \exists y{:}\mathsf{list}\ A_\mathsf{s}.\ \downarrow^\mathsf{s}_\mathsf{L} \mathsf{list}\ A_\mathsf{s},\\ &\mathsf{del} : \oplus\{\mathsf{none} : \downarrow^\mathsf{s}_\mathsf{L} \mathsf{list}\ A_\mathsf{s}, \mathsf{some} : \exists x{:}A_\mathsf{s}.\ \downarrow^\mathsf{s}_\mathsf{L} \mathsf{list}\ A_\mathsf{s}\}\end{aligned}$$

The list is implemented in terms of processes *empty* and *elem*, denoting the empty list and a cons cell, respectively. We show the more interesting case of a cons cell (Fig. 6). The queue is defined by processes *head* (Fig. 7) and *queue_proc* (Fig. 8), the latter being the queue's interface to its clients.

$$\begin{aligned}&elem : \{\delta_1 < \delta_2, \delta_2 < \delta_3, \delta_3 < \delta_4 \vdash \mathsf{list}[\delta_1 \updownarrow^{\delta_2}_{\delta_2}] A_\mathsf{s}[\delta_3 \updownarrow^{\delta_4}_{\delta_4}] \leftarrow A_\mathsf{s}[\delta_3 \updownarrow^{\delta_4}_{\delta_4}], \mathsf{list}[\delta_1 \updownarrow^{\delta_2}_{\delta_2}] A_\mathsf{s}[\delta_3 \updownarrow^{\delta_4}_{\delta_4}]\}\\ &c[\delta_1 \updownarrow^{\delta_2}_{\delta_2}][\delta_3 \updownarrow^{\delta_4}_{\delta_4}] \leftarrow elem \leftarrow x[\delta_3 \updownarrow^{\delta_4}_{\delta_4}], next[\delta_1 \updownarrow^{\delta_2}_{\delta_2}][\delta_3 \updownarrow^{\delta_4}_{\delta_4}] =\\ &\quad c' \leftarrow \mathsf{accept}\ c\ ;\\ &\quad \mathsf{case}\ c'\ \mathsf{of}\\ &\quad |\ \mathsf{ins} \rightarrow y \leftarrow \mathsf{recv}\ c'\ ;\ n \leftarrow elem \leftarrow y,\ next\ ;\ \mathsf{send}\ c'\ n\ ;\\ &\qquad\quad c \leftarrow \mathsf{detach}\ c'\ ;\\ &\qquad\quad c'' : \mathsf{list}[\delta_1 \updownarrow^{\delta_2}_{\delta_2}] A_\mathsf{s}[\delta_3 \updownarrow^{\delta_4}_{\delta_4}] \leftarrow elem \leftarrow x,\ n\ ;\ \mathsf{fwd}\ c\ c''\\ &\quad |\ \mathsf{del} \rightarrow c'.\mathsf{some}\ ;\ \mathsf{send}\ c'\ x\ ;\\ &\qquad\quad c \leftarrow \mathsf{detach}\ c'\ ;\ \mathsf{fwd}\ c\ next\end{aligned}$$

Fig. 6. Imperative queue – *elem* process.

We can now define a client (Fig. 8) for the queue, assuming existence of a corresponding shared session type item and a process *item_proc* offering a session of type $\mathsf{item}[\delta_3 \updownarrow^{\delta_4}_{\delta_4}]$. The client instantiates the queue at world δ_b, allowing it to acquire resources at world w_1, which is exactly the world at which process *queue_proc* instantiates the list. Given that the client itself resides at world δ_a, which is smaller than the queue's world δ_b, the client is allowed to acquire the queue, which in turn will acquire the list to satisfy any requests by the client.

The example showcases a paradigmatic use of several collaborators, where collaborators can hold resources while they "talk down" in the tree. In particular, as illustrated in Fig. 9, the clients C_1, C_2, and C_3 compete for resources at world δ_b, i.e., the queue Q. On the other hand, a client C_i collaborates with the queue Q, the list elements L_i, and the items I_i, since they do not overlap in

[2] We adopt polymorphism for the example without formal treatment since it is orthogonal and has been studied for session types in [23,46].

$$
\begin{array}{l}
\mathit{head} : \{\delta_0 < \delta_1, \delta_1 < \delta_2, \delta_2 < \delta_3, \delta_3 < \delta_4 \vdash \mathsf{queue}[\delta_0 \updownarrow^{\delta_1}_{\delta_1}] A_S[\delta_3 \updownarrow^{\delta_4}_{\delta_4}] \leftarrow \mathsf{list}[\delta_1 \updownarrow^{\delta_2}_{\delta_2}] A_S[\delta_3 \updownarrow^{\delta_4}_{\delta_4}], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{list}[\delta_1 \updownarrow^{\delta_2}_{\delta_2}] A_S[\delta_3 \updownarrow^{\delta_4}_{\delta_4}]\} \\
c[\delta_0 \updownarrow^{\delta_1}_{\delta_1}][\delta_3 \updownarrow^{\delta_4}_{\delta_4}] \leftarrow \mathit{head} \leftarrow \mathit{front}[\delta_1 \updownarrow^{\delta_2}_{\delta_2}][\delta_3 \updownarrow^{\delta_4}_{\delta_4}],\ \mathit{back}[\delta_1 \updownarrow^{\delta_2}_{\delta_2}][\delta_3 \updownarrow^{\delta_4}_{\delta_4}] = \\
\quad c' \leftarrow \mathsf{accept}\ c\ ; \\
\quad \mathsf{case}\ c'\ \mathsf{of} \\
\quad \mid \mathsf{enq} \rightarrow x \leftarrow \mathsf{recv}\ c'\ ; \\
\qquad\qquad \mathit{back}' \leftarrow \mathsf{acquire}\ \mathit{back}\ ; \\
\qquad\qquad \mathit{back}'.\mathsf{ins}\ ;\ \mathsf{send}\ \mathit{back}'\ x\ ;\ e \leftarrow \mathsf{recv}\ \mathit{back}'\ ; \\
\qquad\qquad \mathit{back} \leftarrow \mathsf{release}\ \mathit{back}'\ ; \\
\qquad\qquad c \leftarrow \mathsf{detach}\ c'\ ;\ c'' : \mathsf{queue}[\delta_0 \updownarrow^{\delta_1}_{\delta_1}] A_S[\delta_3 \updownarrow^{\delta_4}_{\delta_4}] \leftarrow \mathit{head} \leftarrow \mathit{front},\ e\ ;\ \mathsf{fwd}\ c\ c'' \\
\quad \mid \mathsf{deq} \rightarrow \mathit{front}' \leftarrow \mathsf{acquire}\ \mathit{front}\ ; \\
\qquad\qquad \mathit{front}'.\mathsf{del}\ ; \\
\qquad\qquad (\mathsf{case}\ \mathit{front}'\ \mathsf{of} \\
\qquad\qquad \mid \mathsf{none} \rightarrow \mathit{front} \leftarrow \mathsf{release}\ \mathit{front}'\ ;\ c'.\mathsf{none}\ ;\ c \leftarrow \mathsf{detach}\ c'\ ; \\
\qquad\qquad\qquad\qquad c'' : \mathsf{queue}[\delta_0 \updownarrow^{\delta_1}_{\delta_1}] A_S[\delta_3 \updownarrow^{\delta_4}_{\delta_4}] \leftarrow \mathit{head} \leftarrow \mathit{front},\ \mathit{back}\ ;\ \mathsf{fwd}\ c\ c'' \\
\qquad\qquad \mid \mathsf{some} \rightarrow x \leftarrow \mathsf{recv}\ \mathit{front}'\ ; \\
\qquad\qquad\qquad\qquad \mathit{front} \leftarrow \mathsf{release}\ \mathit{front}'\ ; \\
\qquad\qquad\qquad\qquad c'.\mathsf{some}\ ;\ \mathsf{send}\ c'\ x\ ;\ c \leftarrow \mathsf{detach}\ c'\ ; \\
\qquad\qquad\qquad\qquad c'' : \mathsf{queue}[\delta_0 \updownarrow^{\delta_1}_{\delta_1}] A_S[\delta_3 \updownarrow^{\delta_4}_{\delta_4}] \leftarrow \mathit{head} \leftarrow \mathit{front},\ \mathit{back}\ ;\ \mathsf{fwd}\ c\ c'')
\end{array}
$$

Fig. 7. Imperative queue – *head* process.

$$
\begin{array}{l}
\mathit{queue\_proc} : \{\delta_0 < \delta_1, \delta_1 < \delta_3, \delta_3 < \delta_4 \\
\quad \vdash \mathsf{queue}[\delta_0 \updownarrow^{\delta_1}_{\delta_1}] A_S[\delta_3 \updownarrow^{\delta_4}_{\delta_4}]\} \\
c[\delta_0 \updownarrow^{\delta_1}_{\delta_1}][\delta_3 \updownarrow^{\delta_4}_{\delta_4}] \leftarrow \mathit{queue\_proc} = \\
\quad \mathsf{w}_2 \leftarrow \mathsf{new\_world}\ ; \\
\quad \delta_1 < \mathsf{w}_2\ ;\ \mathsf{w}_2 < \delta_3\ ; \\
\quad e : \mathsf{list}[\delta_1 \updownarrow^{\mathsf{w}_2}_{\mathsf{w}_2}] A_S[\delta_3 \updownarrow^{\delta_4}_{\delta_4}] \leftarrow \mathit{empty}\ ; \\
\quad c'' : \mathsf{queue}[\delta_0 \updownarrow^{\delta_1}_{\delta_1}] A_S[\delta_3 \updownarrow^{\delta_4}_{\delta_4}] \leftarrow \mathit{head} \\
\qquad \leftarrow e, e\ ; \\
\quad \mathsf{fwd}\ c\ c''
\end{array}
\qquad
\begin{array}{l}
\mathit{client} : \{\delta_a < \delta_b \vdash \mathbf{1}[\delta_a \updownarrow^{\delta_b}_{\delta_b}]\} \\
c[\delta_a \updownarrow^{\delta_b}_{\delta_b}] \leftarrow \mathit{client} = \\
\quad \mathsf{w}_1 \leftarrow \mathsf{new\_world}\ ;\ \mathsf{w}_3 \leftarrow \mathsf{new\_world}\ ; \\
\quad \mathsf{w}_4 \leftarrow \mathsf{new\_world}\ ; \\
\quad \delta_b < \mathsf{w}_1\ ;\ \mathsf{w}_1 < \mathsf{w}_3\ ;\ \mathsf{w}_3 < \mathsf{w}_4\ ; \\
\quad i_0 : \mathsf{item}[\mathsf{w}_3 \updownarrow^{\mathsf{w}_4}_{\mathsf{w}_4}] \leftarrow \mathit{item\_proc}\ ; \\
\quad q : \mathsf{queue}[\delta_b \updownarrow^{\mathsf{w}_1}_{\mathsf{w}_1}] A_S[\mathsf{w}_3 \updownarrow^{\mathsf{w}_4}_{\mathsf{w}_4}] \leftarrow \mathit{queue\_proc}\ ; \\
\quad q' \leftarrow \mathsf{acquire}\ q\ ;\ q'.\mathsf{enq}\ ;\ \mathsf{send}\ q'\ i_0\ ; \\
\quad q \leftarrow \mathsf{release}\ q'\ ;\ \mathsf{close}\ c
\end{array}
$$

Fig. 8. Imperative queue – *queue_proc* process and *client* process.

the set of resources they may acquire: a client acquires resources at δ_b, a queue resources at w_1, a list resources at w_2, and an item resources at w_4, and we have $\delta_a < \delta_b < \mathsf{w}_1 < \mathsf{w}_2 < \mathsf{w}_3 < \mathsf{w}_4$. We note in particular that the setup prevents a list element from acquiring its successor, forcing linear access through the queue.

5 Semantics

In this section, we discuss the meta-theoretical properties of SILL_{S^+}, focusing on deadlock-freedom. The companion technical report [4] provides further details.

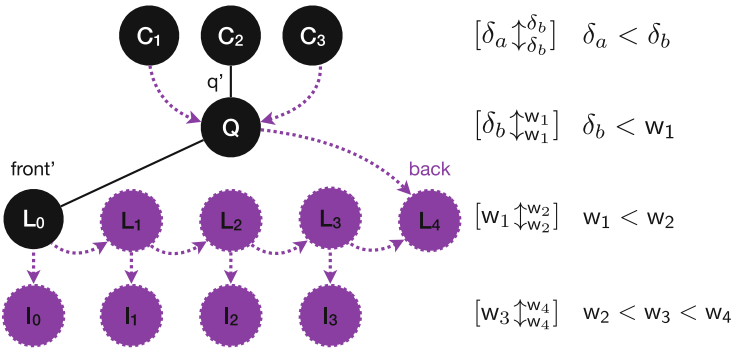


Fig. 9. Run-time process graph for imperative queue (see Fig. 3 for legend).

5.1 Configuration Typing and Preservation

Given the hierarchy between mode S and L and the fact that shared processes cannot depend on linear processes, we divide a configuration into a *shared* part Λ and a linear part Θ. We use the typing judgment $\Psi; \Gamma \vDash \Lambda; \Theta :: \Gamma; \Phi, \Delta$ to type configurations. The judgment expresses that a well-formed configuration $\Lambda; \Theta$ provides the shared channels in Γ and the linear channels in Φ and Δ. A configuration is type-checked relative to all shared channel references and a global order Ψ. While type-checking is compositional insofar as each process definition can be type-checked separately, solely relying on the process' local Ψ (and Γ), at run-time, the entire order that a configuration relies upon is considered. We give the configuration typing rules in Fig. 10.

Our progress theorem crucially depends on the guarantee that the Invariants 1 and 2 from Sect. 3 hold for every linear process in a configuration's tree. This is expressed by the premises $\mathsf{Inv}_1(\mathsf{proc}(a_\mathsf{L}, w_{a_1} \updownarrow^{w_{a_3}}_{w_{a_2}}, P_{a_\mathsf{L}}))$ and $\mathsf{Inv}_2(\mathsf{proc}(a_\mathsf{L}, w_{a_1} \updownarrow^{w_{a_3}}_{w_{a_2}}, P_{a_\mathsf{L}}))$ in rule (T-Θ_2), based on the Definitions 1 and 2 below that restate Invariants 1 and 2 for an entire configuration. We note that Invariant 2 is based on the set of all transitive children (i.e., *descendants*) of a process. We formally define the notion of a descendant inductively over a well-typed linear configuration. The interested reader can find the definition in the companion technical report [4].

Invariant 1 ($\mathsf{min}(\mathsf{parent}) \leq \mathsf{self}(\mathsf{acquired\_child}) \leq \mathsf{max}(\mathsf{parent})$)**.** *If* $\Psi; \Gamma \vDash \Theta :: \Phi, \Delta$ *and for any* $\mathsf{proc}(a_\mathsf{L}, w_{a_1} \updownarrow^{w_{a_3}}_{w_{a_2}}, P_{a_\mathsf{L}}) \in \Theta$ *such that* $\Psi; \Gamma; \Phi_1; \Delta_1 \vdash P_{a_\mathsf{L}} :: (a_\mathsf{L} : A_\mathsf{L}[w_{a_1} \updownarrow^{w_{a_3}}_{w_{a_2}}])$, $\mathsf{Inv}_1(\mathsf{proc}(a_\mathsf{L}, w_{a_1} \updownarrow^{w_{a_3}}_{w_{a_2}}, P_{a_\mathsf{L}}))$ *holds if an only if for every acquired resource* $b_\mathsf{L} : B_\mathsf{L}[w_{b_1} \updownarrow^{w_{b_3}}_{w_{b_2}}] \in \Phi_1$ *it holds that* $\Psi^* \vdash w_{a_2} \leq w_{b_1} \leq w_{a_3}$*. Moreover, if* $P_{a_\mathsf{L}} = x_\mathsf{L} \leftarrow \mathsf{acquire}\, c_\mathsf{S}; Q_{x_\mathsf{L}}$*, for a* $(c_\mathsf{S} : \uparrow^\mathsf{S}_\mathsf{L} C_\mathsf{L}[w_{c_1} \updownarrow^{w_{c_3}}_{w_{c_2}}]) \in \Gamma$*, then, for every acquired resource* $b_\mathsf{L} : B_\mathsf{L}[w_{b_1} \updownarrow^{w_{b_3}}_{w_{b_2}}] \in \Phi_1$*, it holds that* $\Psi^+ \vdash w_{b_1} < w_{c_1}$ *and that* $\Psi^* \vdash w_{a_2} \leq w_{c_1} \leq w_{a_3}$.

$$\frac{}{\Psi; \Gamma \vDash (\cdot) :: (\cdot)}\ (\text{T-}\Theta_1)$$

$$\frac{\begin{array}{c}(a_{\mathsf{S}} : \hat{B}[\mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}]) \in \Gamma \qquad \vdash (A_{\mathsf{L}}, \hat{B})\ \mathsf{sesync} \qquad \Psi \vdash A_{\mathsf{L}}[\mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}]\ \mathsf{type} \\ \Psi^* \vdash \mathsf{w}_{\mathsf{a}_2} \leq \mathsf{w}_{\mathsf{a}_3} \qquad \mathsf{Inv}_1(\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P_{a_{\mathsf{L}}})) \qquad \mathsf{Inv}_2(\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P_{a_{\mathsf{L}}})) \\ \Psi; \Gamma \vDash \Theta :: \Phi, \Phi_1, \Delta, \Delta_1 \qquad \Psi; \Gamma; \Phi_1; \Delta_1 \vdash P_{a_{\mathsf{L}}} :: (a_{\mathsf{L}} : A_{\mathsf{L}}[\mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}])\end{array}}{\Psi; \Gamma \vDash \Theta, \mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P_{a_{\mathsf{L}}}) :: (\Phi, \Delta, a_{\mathsf{L}} : A_{\mathsf{L}}[\mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}])}\ (\text{T-}\Theta_2)$$

$$\frac{}{\Psi; \Gamma \vDash (\cdot) :: (\cdot)}\ (\text{T-}\Lambda_1) \qquad \frac{\begin{array}{c}\vdash (\uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}, \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}})\ \mathsf{sesync} \qquad \Psi \vdash \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}[\mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}]\ \mathsf{type} \\ \Psi^* \vdash \mathsf{w}_{\mathsf{a}_2} \leq \mathsf{w}_{\mathsf{a}_3} \qquad \Psi; \Gamma \vdash P_{a_{\mathsf{S}}} :: (a_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}[\mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}])\end{array}}{\Psi; \Gamma \vDash \mathsf{proc}(a_{\mathsf{S}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P_{a_{\mathsf{S}}}) :: (a_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}[\mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}])}\ (\text{T-}\Lambda_2)$$

$$\frac{}{\Psi; \Gamma \vDash \mathsf{unavail}(a_{\mathsf{S}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}) :: (a_{\mathsf{S}} : \hat{A}[\mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}])}\ (\text{T-}\Lambda_3) \qquad \frac{\Psi; \Gamma \vDash \Lambda :: \Gamma_1 \qquad \Psi; \Gamma \vDash \Lambda' :: \Gamma_2}{\Psi; \Gamma \vDash \Lambda, \Lambda' :: \Gamma_1, \Gamma_2}\ (\text{T-}\Lambda_4)$$

$$\frac{\Psi; \Gamma \vDash \Lambda :: \Gamma \qquad \Psi; \Gamma \vDash \Theta :: \Phi, \Delta}{\Psi; \Gamma \vDash \Lambda; \Theta :: \Gamma; \Phi, \Delta}\ (\text{T-}\Omega)$$

Fig. 10. Configuration typing

Invariant 2 (max(parent) < minima(descendants)). *If* $\Psi; \Gamma \vDash \Theta :: \Phi, \Delta$ *and for any* $\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P_{a_{\mathsf{L}}}) \in \Theta$ *and that process' descendants* $(\Psi; \Gamma \vDash \Theta :: \Phi, \Delta) \rhd a_{\mathsf{L}} = (\Phi', \Delta')$, $\mathsf{Inv}_2(\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P_{a_{\mathsf{L}}}))$ *holds iff for every descendant* $b_{\mathsf{L}} : B_{\mathsf{L}}[\mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}}] \in (\Phi', \Delta')$ *it holds that* $\Psi^+ \vdash \mathsf{w}_{\mathsf{a}_3} < \mathsf{w}_{\mathsf{b}_2}$.

Our preservation theorem states that Invariants 1 and 2 are preserved for every linear process in the configuration along transitions. Moreover, the theorem expresses that the types of the providing linear channels Φ and Δ are maintained along transitions and that new shared channels and worlds may be allocated. The proof relies, in particular, on session types being strictly equi-synchronizing, on a process' type well-formedness and assurance that the process' min world is less than or equal to its max world.

Theorem 5.1 (Preservation). *If* $\Psi; \Gamma \vDash \Lambda; \Theta :: \Gamma; \Phi, \Delta$ *and* $\Lambda; \Theta \longrightarrow \Lambda'; \Theta'$, *then* $\Psi'; \Gamma' \vDash \Lambda'; \Theta' :: \Gamma'; \Phi, \Delta$, *for some* Λ', Θ', Ψ', *and* Γ'.

5.2 Progress

In our development so far we have distilled the two scenarios of interdependencies between processes that can lead to deadlocks: *cyclic acquisitions* and *interdependent acquisitions and synchronizations*. This has lead to the development of a type system that ingrains the notions of *competitors* and *collaborators*, such that the former compete for a set of resources whereas the latter do not overlap in the set of resources they acquire. Our type system then ties these notions to a configuration's linear process tree such that collaborators stand in a parent-descendant relationship to each other and competitors in a sibling/cousin relationship. In this section, we prove that this orchestration is sufficient to rule out any of the aforementioned interdependencies.

To this end we introduce the notions of *red* and *green arrows* that allow us to reason about process interdependencies in a configuration's tree. A red arrow points from a linear $\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, Q)$ to a linear $\mathsf{proc}(b_{\mathsf{L}}, \mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}}, P)$, if the former is attempting to acquire a resource held by the latter and, consequently, is waiting for the latter to release that resource. A green arrow points from a linear $\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, Q)$ to a linear $\mathsf{proc}(b_{\mathsf{L}}, \mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}}, P)$, if the former is waiting to synchronize with the latter. We define these arrows formally as follows:

Definition 5.2 (Acquire Dependency — "Red Arrow"). *Given a well-formed and well-typed configuration* $\Psi; \Gamma \vDash \Lambda; \Theta :: \Gamma; \Phi, \Delta$*, there exists a waiting-due-to-acquire relation* $\mathcal{A}(\Theta)$ *among linear processes in* Θ *at run-time such that*

$$\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, x_{\mathsf{L}} \leftarrow \mathsf{acquire}\ c_{\mathsf{S}};\ Q_{x_{\mathsf{L}}}) <_{\mathcal{A}} \mathsf{proc}(b_{\mathsf{L}}, \mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}}, P\langle c_{\mathsf{L}}\rangle)$$

where $P\langle c_{\mathsf{L}}\rangle$ *denotes a process term with an occurrence of channel* c_{L}*.*

Definition 5.3 (Synchronization Dependency — "Green Arrow"). *Given a well-formed and well-typed configuration* $\Psi; \Gamma \vDash \Lambda; \Theta :: \Gamma; \Phi, \Delta$*, there exists a waiting-due-to-synchronization relation* $\mathcal{S}(\Theta)$ *among linear processes in* Θ *at run-time such that*

$$\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, \_\langle b_{\mathsf{L}}\rangle; Q) <_{\mathcal{S}} \mathsf{proc}(b_{\mathsf{L}}, \mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}}, \_\langle \neg b_{\mathsf{L}}\rangle; P)$$
$$\mathsf{proc}(b_{\mathsf{L}}, \mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}}, \_\langle b_{\mathsf{L}}\rangle; P) <_{\mathcal{S}} \mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, \_\langle \neg b_{\mathsf{L}}\rangle; Q\langle b_{\mathsf{L}}\rangle)$$

where $P\langle a_{\mathsf{L}}\rangle$ *denotes a process term with an occurrence of channel* b_{L}*,* $\_\langle a\rangle; P$ *a process term that currently executes an action along channel* a*, and* $\_\langle \neg a\rangle; P$ *a process term whose currently executing action does not involve the channel* a*.*

It may be helpful to consult Fig. 3 at this point and note the semantic difference between the violet arrows in that figure and the red arrows discussed here. Whereas violet arrows point from the acquiring process to the resource being acquired, red arrows point from the acquiring process to the process that is holding the resource. Thus, violet arrows can go out of the tree, while red arrows stay within. Given the definitions of red and green arrows, we can define the relation $\mathcal{W}(\Theta)$ on the configuration's tree, which contains all process pairs that are in some way waiting for each other:

Definition 5.4 (Waiting Dependency). *Given a well-formed and well-typed configuration* $\Psi; \Gamma \vDash \Lambda; \Theta :: \Gamma; \Phi, \Delta$*, there exists a waiting relation* $\mathcal{W}(\Theta)$ *among processes in* Θ *at run-time such that* $\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P) <_{\mathcal{W}} \mathsf{proc}(b_{\mathsf{L}}, \mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}}, Q)$*,*

- *if* $\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P) <_{\mathcal{A}} \mathsf{proc}(b_{\mathsf{L}}, \mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}}, Q)$*, or*
- *if* $\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P) <_{\mathcal{S}} \mathsf{proc}(b_{\mathsf{L}}, \mathsf{w}_{\mathsf{b}_1} \updownarrow^{\mathsf{w}_{\mathsf{b}_3}}_{\mathsf{w}_{\mathsf{b}_2}}, Q)$*.*

Having defined the relation $\mathcal{W}(\Theta)$, we can now state the key lemma underlying our progress theorem, indicating that $\mathcal{W}(\Theta)$ is acyclic in a well-formed and well-typed configuration.

Lemma 5.5 (Acyclicity of $\mathcal{W}(\Theta)$). *If $\Psi; \Gamma \vDash \Lambda; \Theta :: \Gamma; \Phi, \Delta$, then $\mathcal{W}(\Theta)$ is acyclic.*

We focus on explaining the main idea of the proof here. The proof proceeds by induction on $\Psi; \Gamma \vDash \Theta :: \Phi, \Delta$, assuming for the non-empty case $\Psi; \Gamma \vDash \Theta, \mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P_{a_{\mathsf{L}}}) :: (\Phi, \Delta, a_{\mathsf{L}} : A_{\mathsf{L}}[\mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}])$ that $\mathcal{W}(\Theta)$ is acyclic, by the inductive hypothesis. We then know that there cannot exist any paths of green and red arrows in Θ that form a cycle, and we have to show that there is no way of introducing such a cyclic path by adding node $\mathsf{proc}(a_{\mathsf{L}}, \mathsf{w}_{\mathsf{a}_1} \updownarrow^{\mathsf{w}_{\mathsf{a}_3}}_{\mathsf{w}_{\mathsf{a}_2}}, P_{a_{\mathsf{L}}})$ to the configuration Θ. In particular, the proof considers all possible new arrows that may be introduced by adding the node and that are necessary for creating a cycle, showing that such arrows cannot come about in a well-typed configuration.

We illustrate the reasoning for the two selected cases shown in Fig. 11. Case **(a)** represents a case in which process $P_{a_{\mathsf{L}}}$ is waiting to synchronize with its child $P_{b_{\mathsf{L}}}$ while holding a resource a descendant of $P_{b_{\mathsf{L}}}$ or $P_{b_{\mathsf{L}}}$ itself wants to acquire. However, this scenario cannot come about in a well-typed configuration because $P_{a_{\mathsf{L}}}$ and $P_{b_{\mathsf{L}}}$ are collaborators and thus cannot overlap in resources they acquire. Case **(b)** represents a case in which process $P_{a_{\mathsf{L}}}$ is waiting to synchronize with its child $P_{b_{\mathsf{L}}}$ while another child, process $P_{c_{\mathsf{L}}}$, is waiting to synchronize with $P_{a_{\mathsf{L}}}$. Given acyclicity of $\mathcal{W}(\Theta)$, a necessary condition for a cycle to form is that there already must exist a red arrow **C** in the configuration that connects the subtrees in which the siblings $P_{b_{\mathsf{L}}}$ and $P_{c_{\mathsf{L}}}$ reside. However, this scenario cannot come about in a well-typed configuration because $P_{b_{\mathsf{L}}}$ and $P_{c_{\mathsf{L}}}$ are competitors, forcing $P_{c_{\mathsf{L}}}$ or any of its descendant to release a resource before synchronizing with $P_{a_{\mathsf{L}}}$. These arguments are made precise in various lemmas in [4].

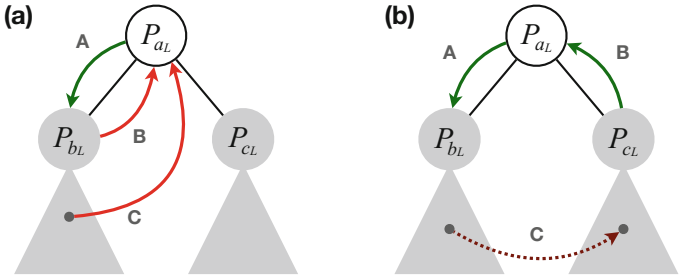


Fig. 11. Two prototypical cases in proof of acyclicty of $\mathcal{W}(\Theta)$.

Given acyclicity of $\mathcal{W}(\Theta)$, we can state and prove the following strong progress theorem. The theorem relies on the notion of a *poised* process, a process currently executing an action along its offering channel, and distinguishes a configuration only consisting of the top-level, linear "main" process from one that consists of several linear processes. We use $|\Theta|$ to denote the cardinality of Θ:

Theorem 5.6 (Progress). *If* $\Psi; \Gamma \vDash \Lambda; \Theta :: (\Gamma; c_{\mathsf{L}} : \mathbf{1}[\mathsf{w}_{c_1} \updownarrow_{\mathsf{w}_{c_2}}^{\mathsf{w}_{c_3}}])$*, then either*

- $\Lambda \longrightarrow \Lambda'$*, for some* Λ'*, or*
- Λ *is poised and*
  - *if* $|\Theta| = 1$*, then either* $\Lambda; \Theta \longrightarrow \Lambda'; \Theta'$*, for some* Λ' *and* Θ'*, or* Θ *is poised, or*
  - *if* $|\Theta| > 1$*, then* $\Lambda; \Theta \longrightarrow \Lambda'; \Theta'$*, for some* Λ' *and* Θ'*.*

The theorem indicates that, as long as there exist at least two linear processes in the configuration, the configuration can always step. If the configuration only consists of the main process, then this process will become poised (i.e., ready to close), once all sub-computations are finished. The proof of the theorem relies on the acyclicity of $\mathcal{W}(\Theta)$ and the fact that all sessions must be strictly equi-synchronizing.

6 Additional Discussion

Linear Forwarding. Our current formalization does not include linear forwarding because a forward changes the process tree and thus endangers the invariants imposed on it. This means that certain programs from the purely linear fragment may not type-check in our system. However, the correspondingly η-expanded versions of these programs should be expressible and type-checkable in $\mathsf{SILL}_{\mathsf{S}^+}$. As part of future work, we want to explore the addition of the linear forward

$$\frac{\Psi^+ \vdash \omega_n < \omega_u}{\Psi;\, \Gamma;\, \cdot;\, y_{\mathsf{L}} : A_{\mathsf{L}}[\omega_m \updownarrow_{\omega_u}^{\omega_v}] \vdash \mathsf{fwd}\ x_{\mathsf{L}}\, y_{\mathsf{L}} :: (x_{\mathsf{L}} : A_{\mathsf{L}}[\omega_j \updownarrow_{\omega_k}^{\omega_n}])}\ (\text{T-ID}_{\mathsf{L}})$$

which allows forwarding to processes that are known to not yet be aliased and whose world annotations meet the premise $\Psi^+ \vdash \omega_n < \omega_u$. Restricting to processes in Δ should uphold Invariant 1, while the premise of the rule should uphold Invariant 2. However, this change will affect the inner working of the proofs, the use of inversion in particular, which might have far-reaching consequences that need to be carefully explored.

Unbounded Process Networks and World Polymorphism. The typing discipline presented in the previous sections, while rich enough to account for a wide range of interesting programs, cannot type programs that spawn a statically undetermined number of shared sessions that are then to be used. For instance, while we can easily type a configuration of any given number of dining philosophers (Sect. 3.3), we cannot type a recursive process in which the number of philosophers (and forks) is potentially unbounded (as done in [21,38]), due to the way worlds are created and propagated across processes.

The general issue lies in implementing a statically unbounded network of processes that interact with each other. These interactions require the processes to be spawned at different worlds which must be generated dynamically as needed.

To interact with such a statically unknown number of processes uniformly, their offering channels must be stored in a list-like structure for later use. However, in our system, recursive types have to be invariant with respect to worlds. For instance, in a recursive type such as $T = A_{\mathsf{L}}@\omega_l \uparrow_{\omega_p}^{\omega_r} \otimes T$, the worlds ω_l, ω_p, ω_r are fixed in the unfoldings of T. Thus, we cannot type a world-heterogeneous list and cannot form such process networks.

Given that the issues preventing us from typing such unbounded networks lie in problems of world invariance, the natural solution is to explore some form of *world polymorphism*, where types can be parameterized by worlds which are instantiated at a later stage. Such techniques have been studied in the context of hybrid logical processes in [7] by considering session types of the form $\forall\delta.A$ and $\exists\delta.A$, sessions that are parametric in the world variable δ, that is instantiated by a concrete reachable world at runtime. While their development cannot be mapped directly to our setting, it is a promising avenue of future work.

7 Related Work

Behavioral Type Analysis of Deadlocks. The addition of channel usage information to types in a concurrent, message-passing setting was pioneered by Kobayashi and Igarashi [30,34], who applied the idea to deadlock prevention in the π-calculus and later to more general properties [31,32], giving rise to a generic system that can be instantiated to produce a variety of concrete typing disciplines for the π-calculus (e.g., race detection, deadlock detection, etc.).

This line of work types π-calculus processes with a simplified form of *process* (akin to CCS [42] terms without name restriction) that characterizes the input/output behavior of processes. These types are augmented with abstract data that pertain to the relative ordering of channel actions, with the type system ensuring that the transitive closure of such orderings forms a strict partial order, ensuring deadlock-freedom (i.e., communication succeeds unless a process diverges). Building on this, Kobayashi et al. proposed type systems that ensure a stronger property dubbed lock-freedom [35] (i.e., communication always succeeds), and variants that are amenable to type inference [36,39]. Kobayashi [37] extended this latter system to more accurately account for recursive processes while preserving the existence of a type inference algorithm.

Our system draws significant inspiration from this line of work, insofar as we also equip types with abstract ordering data on certain communication actions, which is then statically enforced to form a strict partial order. We note that our $\mathsf{SILL}_{\mathsf{S}^+}$ language differs sufficiently from the pure π-calculus in terms of its constructs and semantics to make the formulation of a direct comparison or an immediate application of their work unclear (e.g., [37] uses replication to encode recursive processes). Moreover, we integrate this style of order-based reasoning with both linear and shared session typing, which interact in non-trivial ways (especially in the presence of recursive types and recursive process definitions).

In terms of typability, enforcing session fidelity can be a double-edged sword: some examples of the works above can be transposed to $\mathsf{SILL}_{\mathsf{S}^+}$ with mostly

cosmetic changes and without making use of shared sessions (e.g., a parallel implementation of factorial that recurses via replication but always answers on a private channel); others are incompatible with linear sessions and require the use of shared sessions via the acquire-release discipline, which entails a more indirect but still arguably faithful modelling of the original π-calculus behavior; some examples, however, cannot be easily adapted to the shared session discipline (e.g., $*c?(x,y).x?(z).y?(z) \mid *c?(x,y).y?(z).x?(z)$ is typable in [37], where $x?(z)$ denotes input on x and $*c?(x,y)$ denotes replicated input) and their transcription, while possible, would be too far removed from the original term to be deemed a faithful representation. Recursive processes are known to produce patterns that can be challenging to analyze using such order-based techniques. The work of [21,38] specializes Kobayashi's system to account for potentially unbounded process networks with non-trivial forms of sharing. Such systems are not typable in our work (see Sect. 6 for additional discussion on this topic).

The work of Padovani [44] develops techniques inspired by [35,37] to develop a typing system for deadlock (and lock) freedom for the linear π-calculus where (linear) channels must be used exactly once. By enforcing this form of linearity, the resulting system uses only one piece of ordering data per channel usage and can easily integrate a form of channel polymorphism that accounts for intricate cyclic interleavings of recursive processes. The combination of manifest sharing and linear session typing does not seem possible without the use of additional ordering data, and the lack of single-use linear channels make the robust channel polymorphism of [44] not feasible in our setting.

Dardha and Gay [15] recently integrated a system of Kobayashi-style orderings in a logical session π-calculus based on classical linear logic, extended with the ability to form *cyclic dependencies* of actions on *linear* session channels (Atkey et al. [1] study similar cycles but do not consider deadlock-freedom), without the need for new process constructs or an acquire-release discipline. Their work considers only a restricted form of replication common in linear logic-based works, not including recursive types nor recursive process definitions. This reduces the complexity of their system, at the cost of expressiveness. We also note that the cycles enabled by their system are produced by processes sharing multiple *linear* names. Since linearity is still enforced, they cannot represent the more general form of cycles that exploit shared channels, as we do.

A comparative study of session typing and Kobayashi-style systems in terms of sharing was developed by Dardha and Pérez [16], showing that such order-based techniques can account for sharing in ways that are out of reach of both classical session typing and pure logic-based session typing. Our system (and that of [15]) aims to combine the heightened power of Kobayashi-style systems with the benefits of session typing, which seems to be better suited as a typing discipline for a high-level programming language [18].

Progress and Session Typing. To address limitations of classical binary session types, Honda et al. [27] introduced *multiparty* session types, where sessions are described by so-called global types that capture the interactions between an arbitrary number of session participants. Under some well-formedness

constraints, global types can be used to ensure that a collection of processes correctly implements the global behavior in a deadlock-free way. However, these global type-based approaches do not ensure deadlock freedom in the presence of higher-order channel passing or interleaved multiparty sessions. Coppo et al. [13] and Bettini et al. [6] develop systems that track usage orders among interleaved multiparty sessions, ruling out cyclic dependencies that can lead to deadlocks. The resulting system is quite intricate, since it combines the full multiparty session theory with the order tracking mechanism, interacts negatively with recursion (essentially disallowing interleaving with recursion) and, by tracking order at the multiparty session-level, ends up rejecting various benign configurations that can be accounted for by our more fine-grained analysis. We also highlight the analyses of Vieira and Vasconcelos [54] and Padovani et al. [45] that are more powerful than the approaches above, at the cost of a more complex analysis based on conversation types [10] (themselves a partial-order based technique).

Static Analysis of Concurrent Programs. Lange et al. [40,41] develop a deadlock detection framework applied to the Go programming language. Their work distills CCS processes from programs which are then checked for deadlocks by a form of symbolic execution [40] and *model-checked* against modal μ-calculus formulae [41] which encode deadlock-freedom of the abstracted process (among other properties of interest). Their abstraction introduces some distance between the original program and the analysed process and so the analysis is sound only for certain restricted program fragments, excluding any combination of recursion and process spawning. Our direct approach does not suffer from this limitation.

de'Liguoro and Padovani [17] develop a typing discipline for deadlock-freedom in a setting where processes exchange messages via unordered mailboxes. Their calculus subsumes the actor model and their analysis combines both so-called mailbox types and specialized dependency graphs to track potential cycles between mailboxes in actor-based systems. The unordered nature of actor-based communication introduces significant differences wrt our work, which crucially exploits the ordering of exchanged messages.

8 Concluding Remarks

In this paper we have developed the concept of manifest deadlock-freedom in the context of the language $\mathsf{SILL}_{\mathsf{S}^+}$, a shared session-typed language, showcasing both the programming methodology and the expressiveness of our framework with a series of examples. Deadlock-freedom of well-typed programs is established by a novel abstraction of so-called green and red arrows to reason about the interdependencies between processes in terms of linear and shared channel references.

In future work, we plan to address some of the limitations of the interactions of deadlock-free shared sessions with recursion, by considering promising notions of world polymorphism and world communication. We also plan to study the problem of world inference and the inclusion of a linear forwarding construct.

References

1. Atkey, R., Lindley, S., Morris, J.G.: Conflation confers concurrency. In: Lindley, S., McBride, C., Trinder, P., Sannella, D. (eds.) A List of Successes That Can Change the World. LNCS, vol. 9600, pp. 32–55. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30936-1_2
2. Balzer, S., Pfenning, F.: Manifest sharing with session types. Proc. ACM Program. Lang. (PACMPL) **1**(ICEP), 37:1–37:29 (2017)
3. Balzer, S., Pfenning, F., Toninho, B.: A universal session type for untyped asynchronous communication. In: 29th International Conference on Concurrency Theory (CONCUR). LIPIcs, pp. 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
4. Balzer, S., Toninho, B., Pfenning, F.: Manifest deadlock-freedom for shared session types. Technical report CMU-CS-19-102, Carnegie Mellon University (2019)
5. Benton, P.N.: A mixed linear and non-linear logic: proofs, terms and models. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933, pp. 121–135. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0022251
6. Bettini, L., Coppo, M., D'Antoni, L., Luca, M.D., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_33
7. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Logic-based domain-aware session types, unpublished draft
8. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_16
9. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Math. Struct. Comput. Sci. **26**(3), 367–423 (2016)
10. Caires, L., Vieira, H.T.: Conversation types. Theor. Comput. Sci. **411**(51–52), 4399–4440 (2010)
11. Castro, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. PACMPL **3**(POPL), 29:1–29:30 (2019)
12. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. Inf. Comput. **207**(10), 1044–1077 (2009)
13. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. Math. Struct. Comput. Sci. **26**(2), 238–302 (2016)
14. Crary, K., Harper, R., Puri, S.: What is a recursive module? In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 50–63 (1999)
15. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Baier, C., Dal Lago, U. (eds.) FoSSaCS 2018. LNCS, vol. 10803, pp. 91–109. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89366-2_5
16. Dardha, O., Pérez, J.A.: Comparing deadlock-free session typed processes. In: EXPRESS/SOS, pp. 1–15 (2015)
17. de'Liguoro, U., Padovani, L.: Mailbox types for unordered interactions. In: 32nd European Conference on Object-Oriented Programming, ECOOP 2018, pp. 15:1–15:28 (2018)

18. Gay, S.J., Gesbert, N., Ravara, A.: Session types as generic process types. In: 21st International Workshop on Expressiveness in Concurrency and 11th Workshop on Structural Operational Semantics, EXPRESS/SOS 2014, pp. 94–110 (2014)
19. Gay, S.J., Hole, M.: Subtyping for session types in the π-calculus. Acta Informatica **42**(2–3), 191–225 (2005)
20. Gay, S.J., Vasconcelos, V.T., Ravara, A., Gesbert, N., Caldeira, A.Z.: Modular session types for distributed object-oriented programming. In: 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 299–312 (2010)
21. Giachino, E., Kobayashi, N., Laneve, C.: Deadlock analysis of unbounded process networks. In: Baldan, P., Gorla, D. (eds.) CONCUR 2014. LNCS, vol. 8704, pp. 63–77. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44584-6_6
22. Gommerstadt, H., Jia, L., Pfenning, F.: Session-typed concurrent contracts. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 771–798. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_27
23. Griffith, D.: Polarized substructural session types. Ph.D. thesis, University of Illinois at Urbana-Champaign (2016)
24. Griffith, D., Pfenning, F.: SILL (2015). https://github.com/ISANobody/sill
25. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_35
26. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053567
27. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 273–284. ACM (2008)
28. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 401–418. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_24
29. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 116–133. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_7
30. Igarashi, A., Kobayashi, N.: Type-based analysis of communication for concurrent programming languages. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 187–201. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0032742
31. Igarashi, A., Kobayashi, N.: A generic type system for the Pi-calculus. In: Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 128–141 (2001)
32. Igarashi, A., Kobayashi, N.: A generic type system for the Pi-calculus. Theor. Comput. Sci. **311**(1–3), 121–163 (2004)
33. Jespersen, T.B.L., Munksgaard, P., Larsen, K.F.: Session types for rust. In: 11th ACM SIGPLAN Workshop on Generic Programming, WGP 2015, pp. 13–22 (2015)
34. Kobayashi, N.: A partially deadlock-free typed process calculus. In: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science, pp. 128–139 (1997)
35. Kobayashi, N.: A type system for lock-free processes. Inf. Comput. **177**(2), 122–159 (2002)
36. Kobayashi, N.: Type-based information flow analysis for the π-calculus. Acta Inf. **42**(4–5), 291–347 (2005)

37. Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006). https://doi.org/10.1007/11817949_16
38. Kobayashi, N., Laneve, C.: Deadlock analysis of unbounded process networks. Inf. Comput. **252**, 48–70 (2017)
39. Kobayashi, N., Saito, S., Sumii, E.: An implicitly-typed deadlock-free process calculus. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 489–504. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44618-4_35
40. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off go: liveness and safety for channel-based programming. In: 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 748–761. ACM (2017)
41. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in go using behavioural types. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 1137–1148 (2018)
42. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
43. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in F#. In: Proceedings of the 27th International Conference on Compiler Construction, CC 2018, pp. 128–138 (2018)
44. Padovani, L.: Deadlock and lock freedom in the linear π-calculus. In: Computer Science Logic - Logic in Computer Science (CSL-LICS), pp. 72:1–72:10 (2014)
45. Padovani, L., Vasconcelos, V.T., Vieira, H.T.: Typing liveness in multiparty communicating systems. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 147–162. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43376-8_10
46. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations and observational equivalences for session-based concurrency. Inf. Comput. **239**, 254–302 (2014)
47. Pfenning, F., Griffith, D.: Polarized substructural session types. In: Pitts, A. (ed.) FoSSaCS 2015. LNCS, vol. 9034, pp. 3–22. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46678-0_1
48. Reed, J.: A judgmental deconstruction of modal logic, January 2009. http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf, unpublished manuscript
49. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: 31st European Conference on Object-Oriented Programming, ECOOP 2017, pp. 24:1–24:31 (2017)
50. Scalas, A., Yoshida, N.: Lightweight session programming in scala. In: 30th European Conference on Object-Oriented Programming, ECOOP 2016, pp. 21:1–21:28 (2016)
51. Toninho, B.: A logical foundation for session-based concurrent computation. Ph.D. thesis, Carnegie Mellon University and New University of Lisbon (2015)
52. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: a monadic integration. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 350–369. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_20

53. Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. **217**, 52–70 (2012)
54. Vieira, H.T., Vasconcelos, V.T.: Typing progress in communication-centred systems. In: De Nicola, R., Julien, C. (eds.) COORDINATION 2013. LNCS, vol. 7890, pp. 236–250. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38493-6_17
55. Wadler, P.: Propositions as sessions. In: 17th ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 273–286. ACM (2012)

A Categorical Model of an i/o-typed π-calculus

Ken Sakayori(✉) and Takeshi Tsukada

The University of Tokyo, Tokyo, Japan
sakayori@kb.is.s.u-tokyo.ac.jp

Abstract. This paper introduces a new categorical structure that is a model of a variant of the **i**/**o**-typed π-calculus, in the same way that a cartesian closed category is a model of the λ-calculus. To the best of our knowledge, no categorical model has been given for the **i**/**o**-typed π-calculus, in contrast to session-typed calculi, to which corresponding logic and categorical structure were given. The categorical structure introduced in this paper has a simple definition, combining two well-known structures, namely, closed Freyd category and compact closed category. The former is a model of effectful computation in a general setting, and the latter describes connections via channels, which cause the effect we focus on in this paper. To demonstrate the relevance of the categorical model, we show by a semantic consideration that the π-calculus is equivalent to a core calculus of Concurrent ML.

Keywords: π-calculus · Categorical type theory · Compact closed category · Closed Freyd category

1 Introduction

The Curry-Howard-Lambek correspondence reveals the trinity of the simply-typed λ-calculus, propositional intuitionistic logic and cartesian closed category. Via the correspondence, a type of the calculus can be seen as a formula of the logic, and as an object of a category; a term can be seen as a proof and as a morphism (see, e.g., [23]). Since its discovery, a number of variations have been proposed and studied.

In concurrency theory, a correspondence between a process calculus and logic was established by Caires, Pfenning and Toninho [8,9] and later by Wadler [48]. What they found is that session types [18,20] can be seen as formulas of linear logic [14], and processes as proofs. This remarkable result has inspired lots of work (e.g. [3,4,10,25,45,46]).

This correspondence is, however, not completely satisfactory as pointed out in [3,26], as well as by Wadler himself [48]. The session-typed calculi in [9,48] corresponding to linear logic have only well-behaved processes, because the session type systems guarantee deadlock-freedom and race-freedom of well-typed processes. This strong guarantee is often useful for programmers writing processes

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 640–667, 2019.
https://doi.org/10.1007/978-3-030-17184-1_23

in the typed calculus, but can be seen as a significant limitation of expressive power. For example, it prevents us from modelling wild concurrent systems or programs that might fall into deadlocks or race conditions.

This paper describes an approach to a Curry-Howard-Lambek correspondence for concurrency in the presence of deadlocks and race conditions, from the viewpoint of categorical type theory.

What Is the Categorical Model of the π-calculus? We focus on the π-calculus [30,31] in this paper. This is not only because the π-calculus is widely used and powerful, but also because of a classical result by Sangiorgi [39,42], which is the starting point of our development.

Sangiorgi, in the early 90s, gave translations between the conventional, first-order π-calculus and its higher-order variant [39,42]. This translation allows us to regard the π-calculus as a higher-order programming language.

Let us review the observation by Sangiorgi, using a core of the asynchronous π-calculus: $P ::= \mathbf{0} \mid (P|Q) \mid \bar{a}\langle x\rangle \mid a(x).P$.[1] The idea is to decompose the input-prefixing $a(x).P$ into a and $(x).P$. Let us write $a[(x).P]$ for $a(x).P$ to emphasise the decomposition. Then a reduction can also be decomposed as

$$\bar{a}\langle x\rangle \mid a[(y).P] \mid Q \longrightarrow [(y).P]\langle x\rangle \mid Q \longrightarrow P\{x/y\} \mid Q,$$

where the first step is the communication and the second step is the β-reduction (i.e. $(\lambda y.P)\, x \longrightarrow P\{x/y\}$ in the λ-calculus notation). Hence we regard

- an output $\bar{a}\langle x\rangle$ as an application of a function $\bar{a}$ to x, and
- an input $a(x).P$ as an abstraction $(x).P$ (or $\lambda x.P$) "located" at $a[-]$.

Now, ignoring the mysterious operator $a[-]$, what we had are the core operations of functional programming languages (i.e. abstraction and application). This functional programming language is effectful; in fact, communication via channels is a side effect.

This observation leads us to base our categorical model for the π-calculus on a model for effectful functional programs. Among several models, we choose *closed Freyd category* [37] for modelling the functional part.

Then what is the categorical counterpart of $a[-]$? As this operation seems responsible for communication, this question can be rephrased as: what is the categorical structure for communication? An observation by Abramsky et al. [2] answered this question. They pointed out the importance of *compact closed category* [21] in concurrency theory, which nicely describes CCS-like processes interconnected via ports.

By combining the two structures described above, this paper introduces a categorical structure, which we call *compact closed Freyd category*, as a categorical model of the π-calculus.[2] Despite its simplicity, compact closed Freyd

[1] This calculus slightly differs from the calculus we shall introduce in Sect. 2, but the differences are not important here.

[2] Here is the reason why we do not use a monad for modelling the effect: it is unclear for us how to integrate a monad with the compact closed structure. On the contrary, a Freyd category has a (pre)monoidal category as its component; we can simply require that it is compact closed.

category captures the strong expressive power of the π-calculus. The compact closed structure allows us to connect ports in an arbitrary way, in return for the possibility of deadlocks; the Freyd structure allows us to duplicate objects, and duplication of input channels introduces the possibility of race conditions.

Reconstructing Calculi. This paper introduces two calculi that are sound and complete with respect to the compact closed Freyd category model. One is a variant of the π-calculus, named π_F; the design of π_F is based on the observations described above. The other is a higher-order programming language λ_{ch} defined as an instance of the computational λ-calculus [33]. Designing λ_{ch} is not so difficult because we can make use of the correspondence between computational λ-calculus and closed Freyd category (see Sect. 4). The λ_{ch}-calculus have operations for creating a channel and for sending a value via the channel and, therefore, can be seen as a core calculus of *Concurrent ML* (or *CML*) [38].

Since the higher-order calculus λ_{ch} and π_F correspond to the same categorical model, we can obtain translations between these calculi by simple semantic computations. These translations are "correct by definition" and, interestingly, coincide with those between higher-order and first-order π-calculus [39,42].

On β- vs. $\beta\eta$-theories. The categorical analysis of this paper reveals that many conventional behavioural equivalences for the π-calculus are problematic from a viewpoint of categorical type theory. The problem is that they induce only *semicategories*, which may not have identities for some objects. This is a reminiscent of the β-theory of the λ-calculus, of which categorical model is given by semi-categorical notions [16].

Adding a single rule (which we call the *η-rule*) resolves the problem. Our categorical type theory deals with only equivalences that admits the η-rule, and the simplicity of the theory of this paper essentially relies on the η-rule.

Interestingly the η-rule seems to explain some phenomenon in the literature. For example, Sangiorgi observed that a syntactic constraint called *locality* [28,49] is essential for his translation [39,42]. The correctness of the translation can be proved without using the η-rule, when one restricts the calculus local; we expect that Sangiorgi's observation can be related to this phenomenon.

Contributions. This paper introduces a new variant of the **i**/**o**-typed π-calculus, which we call π_F. A remarkable feature of π_F is that it has a categorical counterpart, called compact closed Freyd category. The correspondence is fairly firm; the categorical semantics is sound and complete, and the term model is the classifying category. The relevance of the model is demonstrated by a semantic reconstruction of Sangiorgi's translation [39,42]. These results open a new frontier in the Curry-Howard-Lambek correspondence for concurrency; session-type is not the only base for a Curry-Howard-Lambek correspondence for π-calculi.

Organisation of this Paper. Section 2 introduces the calculus π_F and discuss equivalences on processes. Section 3 gives the categorical semantics of π_F and

shows soundness and completeness. A connection to a higher-order programming language with channels is studied in Sect. 4. In Sect. 5, we (1) discuss how our work relates to linear logic and (2) present some ideas for how to extend the application range of our model. We discuss related work in Sect. 6 and conclude in Sect. 7. Omitted proofs, as well as detailed definitions, are available in the full version.

2 A Polyadic, Asynchronous π-calculus with i/o-types

This section introduces a variant of π-calculus, named π_F. It is based on a fairly standard calculus, namely polyadic and asynchronous π-calculus with **i**/**o**-types, but the details are carefully designed so that π_F has a categorical model.

2.1 The π_F-calculus

This subsection defines the calculus π_F, which is based on an asynchronous variant of the polyadic π-calculus with **i**/**o**-types in [35]. The aim of this subsection is to explain what are the differences from the conventional π-calculus. Although π_F has some uncommon features, each of them was studied in the literature; see Related Work (Sect. 6) for related ideas and calculi.

Types. The set of *types*, ranged over by S and T, is given by

$$S, T ::= \mathbf{ch}^o[T_1, \ldots, T_n] \mid \mathbf{ch}^i[T_1, \ldots, T_n] \qquad (n \geq 0).$$

The type $\mathbf{ch}^o[T_1, \ldots, T_n]$ is for output channels sending n arguments of types $T_1, \ldots, T_n$. The type $\mathbf{ch}^i[T_1, \ldots, T_n]$ is for input channels. The *dual* $T^\perp$ of type T is defined by $\mathbf{ch}^o[\vec{T}]^\perp \stackrel{\text{def}}{=} \mathbf{ch}^i[\vec{T}]$ and $\mathbf{ch}^i[\vec{T}]^\perp \stackrel{\text{def}}{=} \mathbf{ch}^o[\vec{T}]$. For a sequence $\vec{T} \stackrel{\text{def}}{=} T_1, \ldots, T_n$ of types, we write $\vec{T}^\perp$ for $T_1^\perp, \ldots, T_n^\perp$.

An important difference from [35] is that no channel allows both input and output operations. We will refer this feature of π_F as **i**/**o**-*separation*.

Processes. Let $\mathcal{N}$ be a denumerable set of *names*, ranged over by x, y and z. Each name is either input-only or output-only, because of **i**/**o**-separation.

The set of *processes*, ranged over by P, Q and R, is defined by

$$P, Q, R ::= \mathbf{0} \mid (P|Q) \mid (\boldsymbol{\nu}_{\mathbf{ch}^o[\vec{T}]}\, xy)P \mid x\langle\vec{y}\rangle \mid !x(\vec{y}).P.$$

The notion of *free names*, as well as *bound names*, is defined as usual. The set of free names (resp. bound names) of P is written as $\mathbf{fn}(P)$ (resp. $\mathbf{bn}(P)$). We allow tacit renaming of bound names, and identify α-equivalent processes.

The meaning of the constructs should be clear, except for $(\boldsymbol{\nu}_T\, xy)P$ which is less common. The process $\mathbf{0}$ is the inaction; $P \mid Q$ is a parallel composition; $x\langle\vec{y}\rangle$ is an output; and $!x(\vec{x}).P$ is a replicated input. The restriction $(\boldsymbol{\nu}_T\, xy)P$ hides the names x and y of type T and $T^\perp$ and, at the same time, establishes a connection between x and y. Communication takes place only over bound names explicitly connected by $\boldsymbol{\nu}$. This is in contrast to the conventional π-calculus, in which input-output correspondence is *a priori* (i.e. $\bar{a}$ is the output to a).

$$\frac{}{\Gamma \vdash \mathbf{0} : \diamond} \qquad \frac{\Gamma \vdash P : \diamond \qquad \Gamma \vdash Q : \diamond}{\Gamma \vdash P \mid Q : \diamond} \qquad \frac{\Gamma, x : \mathbf{ch}^o[\vec{T}], y : \mathbf{ch}^i[\vec{T}] \vdash P : \diamond}{\Gamma \vdash (\boldsymbol{\nu}_{\mathbf{ch}^o[\vec{T}]}\, xy)P : \diamond}$$

$$\frac{(x : \mathbf{ch}^i[\vec{T}]) \in \Gamma \qquad \Gamma, \vec{y} : \vec{T} \vdash P : \diamond}{\Gamma \vdash !x(\vec{y}).P : \diamond} \qquad \frac{(x : \mathbf{ch}^o[\vec{T}]) \in \Gamma \qquad \vec{y} : \vec{T} \subseteq \Gamma}{\Gamma \vdash x\langle \vec{y} \rangle : \diamond}$$

Fig. 1. Typing rules for processes

The π_F-calculus does not have non-replicated input $x(\vec{y}).P$.

Typing Rules. A *type environment* Γ is a finite sequence of type bindings of the form $x\colon T$. We assume the names in Γ are pairwise distinct. If $\vec{x} = x_1, \ldots, x_n$ and $\vec{T} = T_1, \ldots, T_n$, we write $\vec{x}\colon \vec{T}$ for $x_1\colon T_1, \ldots, x_n\colon T_n$. We write $(\vec{x}\colon \vec{T}) \subseteq \Gamma$ to mean $x_i\colon T_i \in \Gamma$ for every i.

A *type judgement* is of the form $\Gamma \vdash P : \diamond$, meaning that P is a well-typed process under Γ. The typing rules are listed in Fig. 1.

Notation 1. We define $(\boldsymbol{\nu}_{\mathbf{ch}^i[\vec{T}]}\, xy)P$ as $(\boldsymbol{\nu}_{\mathbf{ch}^o[\vec{T}]}\, yx)P$; then $(\boldsymbol{\nu}_T\, xy)P$ is defined for every T. We abbreviate $(\boldsymbol{\nu}_{T_1}\, x_1y_1)\ldots(\boldsymbol{\nu}_{T_n}\, x_ny_n)P$ as $(\boldsymbol{\nu}_{\vec{T}}\, \vec{x}\vec{y})P$. We often omit type annotations and write $(\boldsymbol{\nu}xy)$ for $(\boldsymbol{\nu}_T\, xy)$ and $(\boldsymbol{\nu}\vec{x}\vec{y})$ for $(\boldsymbol{\nu}_{\vec{T}}\, \vec{x}\vec{y})$. We use a and b for names of input channel types and $\bar{a}$ and $\bar{b}$ for output. Note that a and $\bar{a}$ are connected only if they are bound by the same occurrence of $\boldsymbol{\nu}$. □

Operational Semantics. *Structural congruence*, written $\equiv$, is the smallest congruence relation on processes that satisfies the following rules:

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$(\boldsymbol{\nu}xy)(P \mid Q) \equiv ((\boldsymbol{\nu}xy)P) \mid Q \qquad (\boldsymbol{\nu}wx)(\boldsymbol{\nu}yz)P \equiv (\boldsymbol{\nu}yz)(\boldsymbol{\nu}wx)P$$

where $x, y \notin \mathbf{fn}(Q)$ in the fourth rule and w, x, y, z are distinct in the fifth rule.

The *reduction relation* on processes, written $\longrightarrow$, is defined by the base rule

$$(\boldsymbol{\nu}\vec{w}\vec{z})(\boldsymbol{\nu}\bar{a}a)(!a(\vec{x}).P \mid \bar{a}\langle \vec{y} \rangle \mid Q) \longrightarrow (\boldsymbol{\nu}\vec{w}\vec{z})(\boldsymbol{\nu}\bar{a}a)(!a(\vec{x}).P \mid P\{\vec{y}/\vec{x}\} \mid Q)$$

(where $P\{\vec{x}/\vec{y}\}$ is the capture-avoiding substitution) and the structural rule which concludes $P \longrightarrow Q$ from $\exists P'\, Q'.\ P \equiv P' \longrightarrow Q' \equiv Q$. Note that, unlike conventional π-calculi, communication only occurs over bound names connected by ν. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

It should be clear that deadlocks and racy communications can be expressed in π_F. An example of race is $(\boldsymbol{\nu}\bar{a}a)(\bar{a}\langle \vec{y} \rangle \mid !a(\vec{x}).P \mid !a(\vec{x}).Q)$, where two input actions are trying to consume the output regarded as a resource. A similar process $(\boldsymbol{\nu}\bar{a}a)(!a(\vec{x}).P \,|\, \bar{a}\langle \vec{y} \rangle \mid \bar{a}\langle \vec{z} \rangle)$ does not have a race since the receiver $!a(\vec{x}).P$ is replicated. In general, race conditions on output actions do not occur in π_F.

2.2 Equivalences on Processes

To establish a Curry-Howard-Lambek correspondence is to find a nice algebraic or categorical structure of terms. For example, the original Curry-Howard-Lambek correspondence reveals the cartesian closed structure of λ-terms.

Such a nice structure would become visible only when appropriate notions of composition and of equivalence could be identified, such as substitution and $\beta\eta$-equivalence for the λ-calculus.

As for process calculi, so-called "parallel composition + hiding" paradigm [17] has been used to compose processes. Given typed processes

$$\vec{x} : \vec{T},\ \vec{y} : \vec{S} \vdash P : \diamond \quad \text{and} \quad \vec{w} : \vec{S}^{\perp},\ \vec{u} : \vec{U} \vdash Q : \diamond,$$

their composite via $(\vec{y}, \vec{w})$ is defined as

$$\vec{x} : \vec{T},\ \vec{u} : \vec{U} \vdash (\boldsymbol{\nu}_{\vec{S}}\, \vec{y}\vec{w})(P \mid Q) : \diamond.$$

This kind of composition appears quite often in logical studies of π-calculi [1, 5, 19]. It also plays a central role in *interaction category paradigm* proposed by Abramsky, Gay and Nagarajan [2].

So it remains to determine an equivalence on π-calculus processes, appropriate for our purpose. This subsection approaches the problem from two directions:

- Examining behavioural equivalences proposed and studied in the literature
- Developing a new equivalence based on categorical considerations

Let us clarify the notion of equivalence discussed below. An *equation-in-context* is a judgement of the form $\Gamma \vdash P = Q$, where $\Gamma \vdash P : \diamond$ and $\Gamma \vdash Q : \diamond$. An *equivalence* $\mathcal{E}$ is a set of equations-in-context that is reflexive, transitive and symmetric (e.g. $(\Gamma \vdash P = P) \in \mathcal{E}$ for every $\Gamma \vdash P : \diamond$).

Behavioural Equivalences. As mentioned above, we are interested in the structure of π_F-processes modulo existing behavioural equivalences. Among the various behavioural equivalence, we start with studying *barbed congruence* [32], which is one of the most widely used equivalences.

We define (asynchronous and weak) barbed congruence for π_F. For each name $\bar{a}$, we write $P{\downarrow_{\bar{a}}}$ if $P \equiv (\boldsymbol{\nu}\vec{x}\vec{y})(\bar{a}\langle\vec{z}\rangle \mid Q)$ and $\bar{a}$ is free, and $P{\Downarrow_{\bar{a}}}$ if $\exists Q.\, P \longrightarrow^{*} Q{\downarrow_{\bar{a}}}$. A (Γ/Δ)-*context* is a context C such that $\Gamma \vdash C[P] : \diamond$ for every $\Delta \vdash P : \diamond$.

Definition 1. *A* barbed bisimulation *is a symmetric relation $\mathcal{R}$ on processes such that, whenever $P\ \mathcal{R}\ Q$, (1) $P{\downarrow_{\bar{a}}}$ implies $Q{\Downarrow_{\bar{a}}}$ and (2) $P \longrightarrow P'$ implies $\exists Q'.\, (Q \longrightarrow^{*} Q') \wedge (P'\ \mathcal{R}\ Q')$. Barbed bisimilarity $\overset{\bullet}{\approx}$ is the largest barbed bisimulation. Typed processes $\Delta \vdash P : \diamond$ and $\Delta \vdash Q : \diamond$ are* barbed congruent at Δ, *written $\Delta \vdash P \cong^{c} Q$, if $C[P] \overset{\bullet}{\approx} C[Q]$ for every (Γ/Δ)-context C.* □

Let us consider a category-like structure $\mathcal{C}$ in which an object is a type and a morphism is an equivalence class of π_F-processes modulo barbed congruence. More precisely, a morphism from T to S is a process $x\colon T,\ y\colon S^{\perp} \vdash P\colon \diamond$ modulo

barbed congruence (and renaming of free names x and y). Then the composition (i.e. "parallel composition + hiding") is well-defined on equivalence classes, because barbed congruence is a congruence. This is a fairly natural setting.

We have a strikingly negative result.

Theorem 1. *$\mathcal{C}$ is not a category.*

Proof. In every category, if $f : A \longrightarrow A$ is a left-identity on A (i.e. $f \circ g = g$ for every $g : A \longrightarrow A$), then f is the identity on A. The process $a : \mathbf{ch}^o[], \bar{b} : \mathbf{ch}^i[] \vdash !a().\bar{b}\langle\rangle : \diamond$ seen as a morphism $(\mathbf{ch}^o[]) \longrightarrow (\mathbf{ch}^o[])$ is a left-identity but not the identity. The former means that $c\colon \mathbf{ch}^o[], \bar{b}\colon \mathbf{ch}^i[] \vdash \big((\boldsymbol{\nu}\bar{a}a)(!a().\bar{b}\langle\rangle \mid P)\big) \mathrel{\approx^c} P\{\bar{b}/\bar{a}\}$ for every $c\colon \mathbf{ch}^o[], \bar{a}\colon \mathbf{ch}^i[] \vdash P\colon \diamond$, which is a consequence of the *replicator theorems* [35]. To prove the latter, observe that $(\boldsymbol{\nu}\bar{b}b)(!a().\bar{b}\langle\rangle \mid \mathbf{0})$ and $\mathbf{0}$ are not barbed congruent. Indeed the context $C \stackrel{\text{def}}{=} (\boldsymbol{\nu}\bar{a}a)(\bar{a}\langle\rangle \mid !a().\bar{o}\langle\rangle \mid [])$ distinguishes the processes, where $\bar{o}$ is the observable. □

Note that race condition is essential for the proof, specifically, for the part proving that the process $!a().\bar{b}\langle\rangle$ is not the identity. A race condition occurs in $C[(\boldsymbol{\nu}\bar{b}b)(!a().\bar{b}\langle\rangle \mid \mathbf{0})]$, where $\bar{a}$ in C has two receivers.

The process $!a().\bar{b}\langle\rangle$ is called *forwarder*, and forwarders will play a central role in this paper. Its general form is $a \hookrightarrow \bar{b} \stackrel{\text{def}}{=} !a(\vec{x}).\bar{b}\langle\vec{x}\rangle$. When $x : T$ and $y : T^{\perp}$, we write $x \leftrightharpoons y$ to mean $x \hookrightarrow y$ if $T = \mathbf{ch}^i[\vec{S}]$ and otherwise $y \hookrightarrow x$.

Remark 1. The argument in the proof of Theorem 1 is widely applicable to **i**/**o**-typed calculi, not specific to π_F. In particular, **i**/**o**-separation (i.e. absence of $\mathbf{ch}^{i/o}[\vec{T}]$) is not the cause, but the existence of $\mathbf{ch}^o[\vec{T}]$ or $\mathbf{ch}^i[\vec{T}]$ is. □

Remark 2. Session-typed calculi in Caires, Pfenning and Toninho [8,9], which correspond to linear logic, do not seem to suffer from this problem. In our understanding, this is because of race-freedom of their calculi. □

To obtain a category, we should think of a coarser equivalence that identifies $(\boldsymbol{\nu}\bar{b}b)(!a().\bar{b}\langle\rangle \mid \mathbf{0})$ with $\mathbf{0}$. Such an equivalence should be very coarse; even *must-testing equivalence* [11] fails to equate them. As long as we have checked, only *may-testing equivalence* [11] defined below satisfies the requirement.

Definition 2. *Typed processes $\Delta \vdash P : \diamond$ and $\Delta \vdash Q : \diamond$ are* may-testing equivalent at Δ, *written $\Delta \vdash P =_{may} Q$, if $C[P]\Downarrow_{\bar{a}} \Leftrightarrow C[Q]\Downarrow_{\bar{a}}$ for every (Γ/Δ)-context C and name $\bar{a}$.* □

As we shall see, π_F-processes modulo may-testing equivalence behaves well. May-testing equivalence is, however, often too coarse.

Category-Driven Approach. In this approach, we first guess an appropriate categorical structure sufficient for interpreting π_F, based on intuitions discussed in Introduction (see also Sect. 3.1), and then design an equivalence so that it is sound and complete with respect to the categorical semantics.

Figure 2 defines the equivalence, described as a set of rules. A *π_F-theory* is an equivalence that behaves well from the categorical perspective.

$$\frac{a \notin \mathbf{fn}(P, C) \qquad \bar{a} \notin \mathbf{bn}(C)}{\Gamma \vdash (\boldsymbol{\nu} \bar{a}a)(!a(\vec{x}).P \mid C[\bar{a}\langle \vec{y} \rangle]) = (\boldsymbol{\nu} \bar{a}a)(!a(\vec{x}).P \mid C[P\{\vec{y}/\vec{x}\}])} \text{ (E-Beta)}$$

$$\frac{a, \bar{a} \notin \mathbf{fn}(P)}{\Gamma \vdash (\boldsymbol{\nu} \bar{a}a)!a(\vec{y}).P = \mathbf{0}} \text{ (E-GC)} \qquad \frac{\bar{a}, a \notin \mathbf{fn}(\bar{c}\langle \vec{x} \rangle)}{\Gamma \vdash \bar{c}\langle \vec{x} \rangle = (\boldsymbol{\nu} \bar{a}a)(a \hookrightarrow \bar{b} \mid \bar{c}\langle \vec{x}\{\bar{a}/\bar{b}\} \rangle)} \text{ (E-FOut)}$$

$$\frac{b, \bar{a} \notin \mathbf{fn}(P)}{\Gamma \vdash (\boldsymbol{\nu} \bar{a}a)(b \hookrightarrow \bar{a} \mid P) = P\{b/a\}} \text{ (E-Eta)}$$

$$\frac{P \equiv Q}{\Gamma \vdash P = Q} \text{ (E-SCong)} \qquad \frac{\Delta \vdash P = Q \qquad C \colon \Gamma/\Delta\text{-context}}{\Gamma \vdash C[P] = C[Q]} \text{ (E-Ctx)}$$

Fig. 2. Inference rules of equations-in-context. Each rule has implicit assumptions that the both sides of the equation are well-typed processes.

Definition 3. *An equivalence $\mathcal{E}$ is a* π_F-theory *if it is closed under the rules in Fig. 2. Any set Ax of equations-in-context has the minimum theory $Th(Ax)$ that contains Ax. We write $Ax \rhd \Gamma \vdash P = Q$ if $(\Gamma \vdash P = Q) \in Th(Ax)$.* □

Let us examine each rule in Fig. 2.

The rule (E-Beta) should be compared with the reduction relation. When $C = ([\,] \mid Q)$, then (E-Beta) claims

$$(\boldsymbol{\nu} \bar{a}a)(!a(\vec{x}).P \mid \bar{a}\langle \vec{y} \rangle \mid Q) = (\boldsymbol{\nu} \bar{a}a)(!a(\vec{x}).P \mid P\{\vec{y}/\vec{x}\} \mid Q)$$

provided that $a \notin \mathbf{fn}(P, Q)$, which is indeed an instance of the reduction.

A significant difference from reduction is the side condition. It is essential in the presence of race conditions. Without the side condition, every π_F-theory would be forced to contain the symmetric and transitive closure of the reduction relation; thus it would identify $P \mid (\boldsymbol{\nu} \bar{a}a)(!a().P \mid !a().Q)$ with $Q \mid (\boldsymbol{\nu} \bar{a}a)(!a().P \mid !a().Q)$ for every processes P and Q (where $\bar{a}, a$ are fresh), because

$$\begin{aligned} (\boldsymbol{\nu} \bar{a}a)(\bar{a}\langle\rangle \mid !a().P \mid !a().Q) &\longrightarrow P \mid (\boldsymbol{\nu} \bar{a}a)(!a().P \mid !a().Q) \\ (\boldsymbol{\nu} \bar{a}a)(\bar{a}\langle\rangle \mid !a().P \mid !a().Q) &\longrightarrow Q \mid (\boldsymbol{\nu} \bar{a}a)(!a().P \mid !a().Q). \end{aligned}$$

The side condition prevents π_F-theories from collapsing.

Another, relatively minor, difference is that application of (E-Beta) is not limited to the contexts of the form $[\,] \mid Q$. This kind of extension can be found in, for example, work by Honda and Laurent [19] studying π-calculus from a logical perspective.

The rule (E-GC) runs "garbage-collection". Because no one can send a message to the hidden name a, the process $!a(\vec{x}).P$ will never be invoked and thus is safely discarded. This rule is sound with respect to many behavioural equivalences, including barbed congruence. Rules of this kind often appear in the literature studying logical aspects of concurrent calculi (as in Honda and Laurent [19] and Wadler [48]). There is, however, a subtle difference in the side condition: (E-GC) requires that a and $\bar{a}$ do not appear at all in P.

The rule (E-FOUT) can be seen as the η-rule of abstractions, as in the λ-calculus and in the higher-order π-calculus [39]. In the latter, an output name $\bar{b}$ can be identified with an abstraction $(\vec{y}).\bar{b}\langle\vec{y}\rangle$. Then we have, for example,

$$(\boldsymbol{\nu}\bar{a}a)(a \hookrightarrow \bar{b} \mid \bar{c}\langle\bar{a}\rangle) \;=\; (\boldsymbol{\nu}\bar{a}a)(a \hookrightarrow \bar{b} \mid \bar{c}\langle\,(\vec{y}).\bar{a}\langle\vec{y}\rangle\,\rangle) \;=\; \bar{c}\langle\,(\vec{y}).\bar{b}\langle\vec{y}\rangle\,\rangle \;=\; \bar{c}\langle\bar{b}\rangle$$

where we use (E-BETA) and (E-GC) in the second step. An important usage of (E-FOUT) is to replace an output of free names with that of bound names. This kind of operation has been studied in [7,28] as a part of translations from the π-calculus to its local/internal fragments.[3]

The rule (E-ETA) requires the forwarders are left-identities, directly describing the requirement discussed above.[4]

The rules (E-SCONG) and (E-CTX) are easy to understand. The former requires that structurally congruent processes should be identified; the latter says that a π_F-theory is a congruence.

These rules can be justified from the operational viewpoint, as well. A well-known result on the **i**/**o**-typed π-calculus (see, e.g., [35,43]) shows the following propositions.

Proposition 1. *Barbed congruence is closed under all rules but* (E-ETA). □

Proposition 2. *May-testing equivalence is a π_F-theory.* □

In particular, the latter means that may-testing equivalence is in the scope of the categorical framework of this paper; see Theorem 5.

3 Categorical Semantics

This section introduces the class of *compact closed Freyd categories* and discusses the interpretation of the π_F-calculus in the categories. We show that the categorical semantics is sound and complete with respect to the equational theory given in Sect. 2.2, and that the syntax of the π_F-calculus induces a model.

This section, by its nature, is slightly theoretical compared with other sections. Section 3.1 explains the ideas of this section without heavily using categorical notions; the subsequent subsections require familiarity with categorical type theory.

3.1 Overview

As mentioned in Sect. 1, the categorical model of π_F is *compact closed Freyd category*, which has both closed Freyd and compact closed structures. Here we

[3] Free outputs can be eliminated from π_F-processes by using the rules (E-FOUT) and (E-ETA), i.e. external mobility can be encoded by internal mobility [7,40]. If the calculus is local [28,49], then we do not need (E-ETA) to eliminate free outputs.

[4] A forwarder behaves as a right-identity with respect to every π_F-theory. This is a consequence of rules (E-BETA), (E-GC) and (E-FOUT).

informally discuss what is a compact closed Freyd category and how to interpret π_F by using syntactic representation.

A *closed Freyd category* is a model of higher-order programs with side effects. It has, among others, the structures to interpret the function type $A \Rightarrow B$ and its constructor and destructor, namely, abstraction $\lambda x.t$ and application $t\,u$. It also has a mechanism for unrestricted duplication of variables; in terms of logic, contraction is admissible.

A *compact closed category* can be seen as MLL [14] with the left rule:

$$\frac{\Gamma, A^*, A \vdash I}{\Gamma \vdash I} \qquad \left[\frac{\Gamma \vdash A^* \qquad \Delta \vdash A}{\Gamma, \Delta \vdash I}\right].$$

(The right rule is the companion, which itself is derivable in MLL.)

A *compact closed Freyd category* has all the constructs. It has the structures corresponding to the following type constructors:

$$\text{(closed Freyd)} \quad I, A \otimes B, A \Rightarrow B \qquad \text{(compact closed)} \quad I, A \otimes B, A^*.$$

Note that the pair type $A \otimes B$ (as well as the unit I) coming from the closed Freyd structure is identified with that from the compact closed structure. Inference rules for a compact closed Freyd category is those for functional languages and the above rules of the compact closed structure.

Interpreting π_F in a compact closed Freyd category is to interpret it by using these constructs. As mentioned in Sect. 1, following Sangiorgi [39], we regard

- an output $\bar{a}\langle\vec{x}\rangle$ as an application of a function $\bar{a}$ to a tuple $\langle\vec{x}\rangle$, and
- an input $!a(\vec{x}).P$ as an abstraction $(\vec{x}).P$ (or $\lambda\vec{x}.P$) located at a.

We interpret the output action by using the function application. Hence the type $\mathbf{ch}^o[T]$ is regarded as a function type $T \Rightarrow I$ (where the unit type I is the type for processes i.e. $\diamond$); then the typing rule for output actions becomes

$$\frac{\Gamma, \bar{a}\colon (T \Rightarrow I), x\colon T \vdash \bar{a} : T \Rightarrow I \qquad \Gamma, \bar{a}\colon (T \Rightarrow I), x\colon T \vdash x : T}{\Gamma, \bar{a}\colon (T \Rightarrow I), x\colon T \vdash \bar{a}\langle x\rangle : I}$$

The type $\mathbf{ch}^i[T]$ is understood as $(T \Rightarrow I)^*$; the input-prefixing rule becomes

$$\frac{\dfrac{}{\Gamma, a\colon (T \Rightarrow I)^* \vdash a : (T \Rightarrow I)^*} \qquad \dfrac{\Gamma, a\colon (T \Rightarrow I)^*, x\colon T \vdash P : I}{\Gamma, a\colon (T \Rightarrow I)^* \vdash (x).P : T \Rightarrow I}}{\Gamma, a\colon (T \Rightarrow I)^* \vdash !a(x).P : I}$$

This derivation directly expresses the intuition that an input-prefixing is abstraction followed by allocation; here allocation is interpreted by using the compact closed structure, i.e. connection of ports. The name restriction also has a natural derivation:

$$\frac{\Gamma, a\colon (T \Rightarrow I)^*, \bar{a}\colon (T \Rightarrow I) \vdash P : I}{\Gamma \vdash (\boldsymbol{\nu}\bar{a}a)P : I}$$

3.2 Compact Closed Freyd Category

Let us formalise the ideas given in Sect. 3.1. Hereafter in this section, we assume basic knowledge of category theory and of categorical type theory.

We recall the definitions of compact closed category and closed Freyd category. For simplicity, the structures below are strict and chosen; a functor is required to preserve the chosen structures on the nose.

Definition 4 (Compact closed category [21]). *Let* $(\mathcal{C}, \otimes, I)$ *be a symmetric strict monoidal category. The* dual *of an object* A *in* $\mathcal{C}$ *is an object* A^* *equipped with* unit $\eta_A \colon I \longrightarrow A \otimes A^*$ *and* counit $\epsilon_A \colon A^* \otimes A \longrightarrow I$ *that satisfy the "triangle identities"* $(\eta_A \otimes \mathrm{id}_A); (\mathrm{id}_A \otimes \epsilon_A) = \mathrm{id}_A$ *and* $(\mathrm{id}_{A^*} \otimes \eta_A); (\epsilon_A \otimes \mathrm{id}_{A^*}) = \mathrm{id}_{A^*}$. *The category* $\mathcal{C}$ *is* compact closed *if each object is equipped with a chosen dual.* □

Definition 5 (Closed Freyd category [37]). *A* Freyd category *is given by (1) a category with chosen finite products* $(\mathcal{C}, \otimes, I)$*, called* value category*, (2) a symmetric strict monoidal category* $(\mathcal{K}, \otimes, I, \mathbf{symm})$*, called* producer category*, and (3) an identity-on-object strict symmetric monoidal functor* $J \colon \mathcal{C} \to \mathcal{K}$. *A Freyd category is a* closed Freyd category *if the functor* $J(-) \otimes A \colon \mathcal{C} \to \mathcal{K}$ *has the (chosen) right adjoint* $A \Rightarrow - \colon \mathcal{K} \to \mathcal{C}$ *for every object* A. *We write* $\Lambda_{A,B,C}$ *for the natural bijection* $\mathcal{K}(J(A) \otimes B, C) \longrightarrow \mathcal{C}(A, B \Rightarrow C)$ *and* $\mathbf{eval}_{A,B}$ *for* $\Lambda^{-1}(\mathrm{id}_{A \Rightarrow B}) \colon (A \Rightarrow B) \otimes A \longrightarrow B$ *in* $\mathcal{K}$. □

Remark 3. The above definition is a restriction of the original one [37], in which $\mathcal{K}$ is a *premonoidal* [36] category. This change reflects concurrency of the calculus. In fact, it validates the following law, expressed by the syntax of the computational λ-calculus [33],

$$\mathbf{let}\, x = M \,\mathbf{in}\, \mathbf{let}\, y = N \,\mathbf{in}\, L \quad = \quad \mathbf{let}\, y = N \,\mathbf{in}\, \mathbf{let}\, x = M \,\mathbf{in}\, L.$$

Then one can evaluate M by using the left form and N by using the right form. This law allows us to evaluate M and N in arbitrary order, or concurrently. □

We now introduce the categorical structure corresponding to the π_F-calculus.

Definition 6 (Compact closed Freyd category). *A* compact closed Freyd category *is a Freyd category* $J : \mathcal{C} \longrightarrow \mathcal{K}$ *such that (1)* $\mathcal{K}$ *is compact closed, and (2)* J *has the (chosen) right adjoint* $I \Rightarrow - \colon \mathcal{K} \to \mathcal{C}$. □

We shall often write J for a compact closed Freyd category $J \colon \mathcal{C} \circlearrowright \mathcal{K}$.

A compact closed Freyd category is a closed Freyd category:

$$\mathcal{K}(J(A) \otimes B, C) \;\cong\; \mathcal{K}(J(A), B^* \otimes C) \;\cong\; \mathcal{C}(A, I \Rightarrow (B^* \otimes C)).$$

Example 1. The most basic example of a compact closed Freyd category is (the strict monoidal version of) $J \colon \mathbf{Sets} \circlearrowright \mathbf{Rel} \colon \mathcal{P}$. Here J is the identity-on-object functor that maps a function to its graph and $\mathcal{P}$ is the "power set functor"

$$
\begin{aligned}
[\![\mathbf{ch}^i[T_1,\ldots,T_n]]\!] &\stackrel{\text{def}}{=} (([\![T_1]\!]\otimes\cdots\otimes[\![T_n]\!])\Rightarrow I)^*\\
[\![\mathbf{ch}^o[T_1,\ldots,T_n]]\!] &\stackrel{\text{def}}{=} ([\![T_1]\!]\otimes\cdots\otimes[\![T_n]\!])\Rightarrow I\\
[\![\Gamma\vdash \mathbf{0}:\diamond]\!] &\stackrel{\text{def}}{=} J(!_\Gamma)\\
[\![\Gamma\vdash !a(\vec{x}).P:\diamond]\!] &\stackrel{\text{def}}{=} J(\langle \pi_a^\Gamma, \Lambda_{\Gamma,\vec{T},I}([\![\Gamma,\vec{x}:\vec{T}\vdash P:\diamond]\!])\rangle);\epsilon_{\mathbf{ch}[\vec{T}]}\\
[\![\Gamma\vdash \bar{a}\langle\vec{x}\rangle:\diamond]\!] &\stackrel{\text{def}}{=} J(\langle \pi_{\bar{a}}^\Gamma, \pi_{x_1}^\Gamma,\ldots,\pi_{x_n}^\Gamma\rangle);\mathbf{eval}_{\vec{T},I}\\
[\![\Gamma\vdash P\mid Q:\diamond]\!] &\stackrel{\text{def}}{=} J(\Delta_\Gamma);([\![\Gamma\vdash P:\diamond]\!]\otimes[\![\Gamma\vdash Q:\diamond]\!])\\
[\![\Gamma\vdash (\boldsymbol{\nu} xy)P:\diamond]\!] &\stackrel{\text{def}}{=} (\mathrm{id}_\Gamma\otimes\eta_T);[\![\Gamma,x:T,y:T^\perp\vdash P:\diamond]\!]
\end{aligned}
$$

Fig. 3. Interpretation of types and processes. Here $!_\Gamma$, Δ_Γ and π_y^Γ are maps in $\mathcal{C}$ induced by the cartesian structure, namely, $!_\Gamma\colon [\![\Gamma]\!]\longrightarrow I$ is the terminal map, $\Delta_\Gamma\colon [\![\Gamma]\!]\longrightarrow [\![\Gamma]\!]\otimes[\![\Gamma]\!]$ is the diagonal map and, when $\Gamma=(y_1\colon T_1,\ldots,y_n\colon T_n)$ and $x=y_j$, the morphism $\pi_x^\Gamma\colon [\![\Gamma]\!]\longrightarrow [\![T_j]\!]$ is the j-th projection. The interpretation of a type environment $x_1\colon T_1,\ldots,x_n\colon T_n$ is $[\![T_1]\!]\otimes\cdots\otimes[\![T_n]\!]$.

that maps a relation $\mathcal{R}\subseteq A\times B$ to a function $\mathcal{P}(\mathcal{R})\stackrel{\text{def}}{=}\{(S_A,S_B)\mid S_B=\{b\mid a\in S_A, a\ \mathcal{R}\ b\}\}$. Another example is obtained by replacing sets with posets, functions with monotone functions and relations with downward closed relations. □

Example 2. A more sophisticated example is taken from Laird's game-semantic model of π-calculus [22]. Precisely speaking, the model in [22] itself is not compact closed Freyd, but its variant (with non-negative arenas) is. This model is important since it is fully abstract w.r.t. may-testing equivalence [22, Theorem 1]; hence our framework has a model that captures the may-testing equivalence. □

3.3 Interpretation

Given a compact closed Freyd category $J\colon \mathcal{C}\rightleftarrows\mathcal{K}$, this section defines the interpretation $[\![-]\!]_J$. It maps types and type environments to objects as usual, and a well-typed process $\Gamma\vdash P:\diamond$ to a morphism $[\![P]\!]\colon [\![\Gamma]\!]\to I$ in $\mathcal{K}$ (recall that the tensor unit I is the interpretation of the type for processes).

Figure 3 defines the interpretation of types and processes. It simply formalises the ideas presented in Sect. 3.1: for example, the interpretation of $!a(\vec{x}).P$ is the abstraction Λ (from the closed Freyd structure) followed by location ϵ (from the compact closed structure). There are some points worth noting.

- $(A\Rightarrow I)^*$ is *not* isomorphic to $A^*\Rightarrow I$, $A\Rightarrow I$ nor $I\Rightarrow A$. Indeed $(A\Rightarrow I)^*$ cannot be simplified. Do not confuse it with a valid law $I\Rightarrow(A^*)\cong A\Rightarrow I$.
- A parallel composition is interpreted as a pair. Recall that two components of a pair are evaluated in parallel in this setting (cf. Remark 3).
- All but the last rule use the cartesian structure of $\mathcal{C}$ in order to duplicate or discard the environment.

Example 3. Let us consider $y : T \vdash (\boldsymbol{\nu}\bar{a}a)(\bar{a}\langle y\rangle \mid !a(x).P) : \diamond$, where $\bar{a}, a, y \notin \mathbf{fn}(P)$ and $a\colon \mathbf{ch}^i[T]$. By (E-Beta) and (E-GC), this process is equal to $P\{y/x\}$. It is natural to expect that the interpretations of the two processes coincide; indeed it is. As the following calculation indicates, our semantics factorises the reduction into two steps: (1) the "transmission" of the closure $\lambda\vec{x}.P$ by the triangle identity of the compact closed structure, and (2) the β-reduction modelled by **eval** of the closed Freyd structure:

$$
\begin{aligned}
&[\![y : T \vdash (\boldsymbol{\nu}\bar{a}a)(\bar{a}\langle y\rangle \mid !a(x).P) : \diamond]\!] \\
&= (\mathrm{id}_T \otimes \eta_{\mathbf{ch}^o[T]}); [\![y : T, \bar{a} : \mathbf{ch}^o[T], a : \mathbf{ch}^i[T] \vdash \bar{a}\langle y\rangle \mid !a(x).P : \diamond]\!] \\
&= (\mathrm{id} \otimes \eta); ([\![y : T, \bar{a} : \mathbf{ch}^o[T] \vdash \bar{a}\langle y\rangle : \diamond]\!] \otimes [\![a : \mathbf{ch}^i[T] \vdash !a(x).P : \diamond]\!]) \\
&= (\mathrm{id} \otimes \eta); ((\mathbf{symm}_{T,\mathbf{ch}^o[T]}; \mathbf{eval}_{T,I}) \otimes (\mathrm{id}_{\mathbf{ch}[T]^*} \otimes J(\Lambda([\![x : T \vdash P : \diamond]\!])))); \epsilon_{T \Rightarrow I} \\
&= (\mathrm{id}_T \otimes J(\Lambda([\![x : T \vdash P : \diamond]\!]))); \mathbf{symm}_{T,\mathbf{ch}^o[T]}; \mathbf{eval}_{T,I} \quad \text{(By triangle identity)} \\
&= (J(\Lambda([\![x : T \vdash P : \diamond]\!])) \otimes \mathrm{id}_T); \mathbf{eval}_{T,I} \\
&= [\![x : T \vdash P]\!] \qquad \text{(By the universality of } \mathbf{eval}) \\
&= [\![y : T \vdash P\{y/x\} : \diamond]\!].
\end{aligned}
$$

(Here we implicitly use derived rules for weakening and exchange.) □

Example 4. The interpretation of a forwarder $a : \mathbf{ch}^i[\vec{T}], \bar{b} : \mathbf{ch}^o[\vec{T}] \vdash a \hookrightarrow \bar{b} : \diamond$ is the counit $\epsilon_{\mathbf{ch}^o[\vec{T}]} : [\![\mathbf{ch}^o[\vec{T}]]\!]^* \otimes [\![\mathbf{ch}^o[\vec{T}]]\!] \longrightarrow I$ in $\mathcal{K}$, which is the one-sided form of the identity. Recall that a forwarder is the identity in every π_F-theory. □

The semantics is sound and complete. That means, a judgement $Ax \rhd \Gamma \vdash P = Q$ is provable if and only if $\Gamma \vdash P = Q$ is valid in all models J of Ax.

Here we define the related notions and prove soundness; completeness is the topic of the next subsection.

Definition 7. *An equational judgement $\Gamma \vdash P = Q$ is* valid in J *if $[\![\Gamma \vdash P : \diamond]\!]_J = [\![\Gamma \vdash Q : \diamond]\!]_J$. Given a set Ax of non-logical axioms, J is a* model of Ax, *written $J \models Ax$, if it validates all judgements in Ax. We write $Ax \rhd \Gamma \Vdash P = Q$ if $\Gamma \vdash P = Q$ is valid in every J such that $J \models Ax$.* □

Theorem 2 (Soundness). *If $Ax \rhd \Gamma \vdash P = Q$, then $Ax \rhd \Gamma \Vdash P = Q$.* □

3.4 Term Model

A *term model* is a category whose objects are type environments and whose morphisms are terms (i.e. processes in this setting). This section gives a construction of the term model, by which we show completeness. This subsection basically follows the standard arguments in categorical type theory; we mainly focus on the features unique to our model, giving a sketch to the common part.

Given a set Ax of axioms, we define the term model $J_{Ax} : \mathcal{C}_{Ax} \overset{\perp}{\leftrightarrows} \mathcal{K}_{Ax}$, which we also write as $Cl(Ax)$.

The definition of the producer category $\mathcal{K}_{Ax}$ follows the standard recipe. As usual, its objects are finite lists of types. The monoidal product $\vec{T} \otimes \vec{S}$ is the concatenation of the lists and the dual $\vec{T}^*$ is $\vec{T}^\perp$. Given objects $\vec{T}$ and $\vec{S}$, a morphism from $\vec{T}$ to $\vec{S}$ is a process $\vec{x}\colon \vec{T}, \vec{y}\colon \vec{S}^\perp \vdash P : \diamond$ (modulo renaming of variables $\vec{x}$ and $\vec{y}$). If $Ax \rhd \vec{x}\colon \vec{T}, \vec{y}\colon \vec{S}^\perp \vdash P = Q$ is provable, then P and Q are regarded as the same morphism. Composition of morphisms is defined as "parallel composition plus hiding": For morphisms $P : \vec{T} \longrightarrow \vec{S}$ and $Q : \vec{S} \longrightarrow \vec{U}$, i.e. processes such that $\vec{x}\colon \vec{T},\ \vec{y}\colon \vec{S}^\perp \vdash P : \diamond$ and $\vec{z}\colon \vec{S},\ \vec{w}\colon \vec{U}^\perp \vdash Q : \diamond$, their composite is $\vec{x}\colon \vec{T}, \vec{w}\colon \vec{U}^\perp \vdash (\boldsymbol{\nu}\vec{y}\vec{z})(P \mid Q) : \diamond$. The monoidal product $P \otimes Q$ of morphisms is the parallel composition $P \mid Q$. The identity, as well as the symmetry of the monoidal product and the unit and counit of the compact closed structure, is a parallel composition of forwarders: for example, the identity on $\vec{S}$ is $\vec{x} : \vec{S}, \vec{y} : \vec{S}^\perp \vdash x_1 \leftrightharpoons y_1 \mid \cdots \mid x_n \leftrightharpoons y_n : \diamond$ where n is the length of $\vec{S}$. The facts that most structural morphisms are forwarders and that forwarders compose are the keys to show that $\mathcal{K}_{Ax}$ is a compact closed category.

We then see the definition of $\mathcal{C}_{Ax}$, of which the definition of morphisms has a subtle point. The objects of $\mathcal{C}_{Ax}$ are by definition the same as $\mathcal{K}_{Ax}$, i.e. lists of types. The definition of morphisms relies on the notion of *values*. The values are defined by the grammar $V ::= x \mid (\vec{x}).P$, where P is a process and $(\vec{x}).P$ is called an *abstraction*. Typing rules for values are as follows:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, \vec{x}\colon \vec{T} \vdash P}{\Gamma \vdash (\vec{x}).P : \mathbf{ch}^o[\vec{T}]}.$$

(To understand the right rule, recall that $[\![\mathbf{ch}^o[\vec{T}]]\!] = [\![\vec{T}]\!] \Rightarrow I$.) A morphism from $\vec{T}$ to $\vec{S} = (S_1, \ldots, S_n)$ is an n-tuple $(V_1, \ldots, V_n)$ of values of type $\vec{x}\colon \vec{T} \vdash V_i\colon S_i$ for each i (modulo renaming of $\vec{x}$). Composition is intuitively defined by "substitution followed by β-reduction" whose definition is omitted here.[5]

The functor J_{Ax} places the values to the channels. For example, let $\vec{T} = (\mathbf{ch}^i[U_1], \mathbf{ch}^o[U_2])$ and consider the morphism in $\mathcal{C}_{Ax}$ given by

$$a\colon \mathbf{ch}^i[T_1], \bar{b}\colon \mathbf{ch}^o[T_2] \vdash (a, \bar{b}, (\vec{x}).P) : (\mathbf{ch}^i[T_1], \mathbf{ch}^o[T_2], \mathbf{ch}^o[\vec{S}])$$

where $\vec{S}$ is the type for $\vec{x}$. The image of this morphism by the functor J_{Ax} is

$$a\colon \mathbf{ch}^i[T_1], \bar{b}\colon \mathbf{ch}^o[T_2], \bar{c}\colon \mathbf{ch}^o[T_1], d\colon \mathbf{ch}^i[T_2], e\colon \mathbf{ch}^i[\vec{S}] \vdash a \hookrightarrow \bar{c} \mid d \hookrightarrow \bar{b} \mid !e(\vec{x}).P\colon \diamond.$$

This example contains all the three ways to place a value to a given channel.

Theorem 3. *$Cl(Ax)$ is a compact closed Freyd category for every Ax.* □

In the model $Cl(Ax)$, the interpretation of a process $\Gamma \vdash P : \diamond$ is the equivalence class that P belongs to. This fact leads to completeness.

[5] Here is a subtle technical issue that we shall not address in this paper; see the long version for the formal definition. We think, however, that this paragraph conveys a precise intuition.

Theorem 4 (Completeness). *If* $Ax \rhd \Gamma \Vdash P = Q$*, then* $Ax \rhd \Gamma \vdash P = Q$*.* □

Theorem 5. *There exists a compact closed Freyd category* J *that is fully abstract w.r.t. may-testing equivalence, i.e.* $\Gamma \vdash P =_{may} Q$ *iff* $[\![P]\!]_J = [\![Q]\!]_J$*.*

Proof. Let J be the term model $Cl(=_{may})$ and use Proposition 2. □

3.5 Theory/Model Correspondence

It is natural to expect that $Cl(Ax)$ is the *classifying category* as in the standard categorical type theory. This means, to give a model of Ax in J is equivalent to give a structure-preserving functor $Cl(Ax) \longrightarrow J$. This subsection clarifies and studies this claim.

The set $\mathrm{Mod}(Ax, J)$ of models of Ax in J is defined as follows. If $J \models Ax$, then $\mathrm{Mod}(Ax, J)$ is a singleton set[6]; otherwise $\mathrm{Mod}(Ax, J)$ is the empty set.

We then define the notion of structure-preserving functors.

Definition 8. *A* strict compact closed Freyd functor *from* $J \colon \mathcal{C} \overset{\curvearrowright}{\underset{\curvearrowleft}{\perp}} \mathcal{K} \colon I \Rightarrow (-)$ *to* $J' \colon \mathcal{C}' \overset{\curvearrowright}{\underset{\curvearrowleft}{\perp}} \mathcal{K}' \colon I \Rightarrow' (-)$ *is a pair of functor* (Φ, Ψ) *such that*

- Φ *is a strict finite product preserving functor from* $\mathcal{C}$ *to* $\mathcal{C}'$*,*
- Ψ *is a strict symmetric monoidal functor from* $\mathcal{K}$ *to* $\mathcal{K}'$ *that preserves the chosen compact closed structures (i.e. units and counits) on the nose, and*
- (Φ, Ψ) *is a map of adjoints between* $J \dashv I \Rightarrow (-)$ *and* $J' \dashv I \Rightarrow' (-)$*.*

□

The collection of (small) compact closed Freyd categories and strict compact closed Freyd functors form a 1-category, which we write as $CCFC$.

Now the question is whether $\mathrm{Mod}(Ax, J) \overset{?}{\cong} CCFC(Cl(Ax), J)$ in **Set**.

Unfortunately this does not hold. More precisely, the left-to-right inclusion does not hold in general. This means that the term model satisfies some additional axioms reflecting some aspects of the π_F-calculus.

The additional axioms reflect the definition of the dual $\vec{T}^*$ in the term model; we have $\vec{T}^* \overset{\mathrm{def}}{=} \vec{T}^\perp$ by definition, and thus $\vec{T}^{**} = \vec{T}$ and $(\vec{T} \otimes \vec{S})^* = \vec{T}^* \otimes \vec{S}^*$. It might be surprising that these equations are harmful because isomorphisms $A^{**} \cong A$ and $(A \otimes B)^* \cong A^* \otimes B^*$ exist in every compact closed category. The point is that the equations also require $\mathcal{C}$ to have isomorphisms $A^{**} \cong A$ and $(A \otimes B)^* \cong A^* \otimes B^*$ (witnessed by the respective identities).

We formally define the additional axioms, which we call **(I)** and **(D)**:

(I) The canonical isomorphism $A^{**} \longrightarrow A$ in $\mathcal{K}$ is the identity.
(D) The canonical isomorphism $(A \otimes B)^* \longrightarrow A^* \otimes B^*$ in $\mathcal{K}$ is the identity.

Theorem 6. $\mathrm{Mod}(Ax, J) \cong CCFC(Cl(Ax), J)$ *if* J *satisfies* ***(I)*** *and* ***(D)****.* □

[6] Because we consider only the empty signature, the set of valuations is singleton.

$$\sigma ::= \tau \to \tau' \qquad \xi ::= \sigma \qquad \tau ::= (\xi_1, \ldots, \xi_n)$$
$$V ::= x \mid \lambda\langle\vec{x}\rangle.M$$
$$M ::= \langle\vec{V}\rangle \mid V\ \langle\vec{V}\rangle \mid \mathbf{let}\ \langle\vec{x}\rangle = M\ \mathbf{in}\ M'$$

(a) λ_c

$$\xi ::= \cdots \mid \sigma^* \qquad V ::= \cdots \mid \mathbf{channel}_\sigma \mid \mathbf{send}_\sigma$$

(b) λ_{ch} (difference from λ_c)

Fig. 4. Syntax of types and terms of the λ_c- and λ_{ch}-calculi. The syntax of λ_c is adapted to the setting of this paper.

4 A Concurrent λ-calculus and (de)compilation

In order to demonstrate the relevance of our semantic framework, this section tries to give a semantic reconstruction of fully-abstract compilation and decompilation from a higher-order calculus to the (first-order) π-calculus, such as [39,42]. We first design an instance of the computational λ-calculus [33], named λ_{ch}, that is sound and complete with respect to compact closed Freyd categories. It is obtained by a straightforward extension of the coincidence between the computational λ-calculus and closed Freyd categories (Sect. 4.1). There are translations between π_F and λ_{ch} since both are sound and complete with respect to compact closed Freyd categories. Section 4.2 actually calculates the translations, and compare them with those in [39,42].

4.1 The λ_{ch}-calculus

The λ_{ch}-calculus is a computational λ-calculus with additional constructors dealing with channels. This section introduces and explains the calculus.

The situation is nicely expressed by the following intuitive equation:

$$\frac{\lambda_{ch}}{\lambda_c} \approx \frac{(\text{compact closed Freyd category} + \mathbf{I} + \mathbf{D})}{(\text{closed Freyd category})}.$$

The base calculus λ_c is the *computational λ-calculus*, which corresponds to closed Freyd category [33,37]. It is a call-by-value higher-order programming language, given in Fig. 4(a). Our calculus λ_{ch} is obtained by adding type and term constructors originating from the compact closed structure, which λ_c does not have.

Syntax. As for types, λ_{ch} has a new constructor coming from the dual object A^*. Normalising occurrences of the dual A^* using the axioms **(I)** $A^{**} = A$ and **(D)** $(A \otimes B)^* = A^* \otimes B^*$, we obtain the following grammar of types:

$$\sigma ::= \tau \to \tau' \qquad \xi ::= \sigma \mid \sigma^* \qquad \tau ::= (\xi_1, \ldots, \xi_n)$$

where $n \geq 0$ and $(\xi_1, \ldots, \xi_n)$ is an alternative notation for $\xi_1 \otimes \cdots \otimes \xi_n$. Compared with λ_c, the only new type is the dual type σ^* of a function type σ.

As for terms, λ_{ch} has constructors corresponding to the unit and counit

$$\eta_A : I \longrightarrow A \otimes A^* \qquad \epsilon_A : A^* \otimes A \longrightarrow I \qquad \text{(for each object } A\text{)}$$

of the compact closed structure. We simply add these morphisms as constants:

$$\frac{}{\Gamma \vdash \mathbf{channel}_\sigma : () \to (\sigma, \sigma^*)} \qquad \text{and} \qquad \frac{}{\Gamma \vdash \mathbf{send}_\sigma : (\sigma^*, \sigma) \to ()}.$$

We shall often omit the subscript σ.

In summary, we obtain the syntax of λ_{ch} shown in Fig. 4. Interestingly, λ_{ch} can be seen as a very core of Concurrent ML [38], a practical higher-order concurrent language, although λ_{ch} is developed from purely semantic considerations.

Semantics. Let us first discuss the intuitive meanings of the new constructors. The type σ^* is for *output channels*; $\mathbf{channel}\,\langle\rangle$ creates and returns a pair of an input channel and an output channel that are connected; and $\mathbf{send}\,\langle\alpha, V\rangle$ sends the value V via the output channel α. The following points are worth noting.

- λ_{ch} has no type constructor for *input channels*. The type system does not distinguish between input channels for type σ and values of type σ.
- λ_{ch} has no *receive* constructor. Receiving operation is implicit and on demand, delayed as much as possible.
- The send operator broadcasts a value via a channel. Several receivers may receive the same value from the same channel.

The first two points reflect the asynchrony of π_F, and the last point reflects the absence of non-replicated input (cf. Sect. 4.2).

Based on this intuition, we develop the operational, axiomatic and categorical semantics of λ_{ch}. We shall use the following abbreviations:

$$(\boldsymbol{\nu} xy)M \stackrel{\text{def}}{=} \mathbf{let}\,\langle x, y\rangle = \mathbf{channel}\,\langle\rangle\,\mathbf{in}\,M \qquad M \parallel N \stackrel{\text{def}}{=} \mathbf{let}\,\langle\rangle = M\,\mathbf{in}\,N.$$

Operational Semantics. Assume an infinite set $\mathcal{X}$ of *channels*, ranged over by α and β. For each channel α, we write α for the input name and $\bar{\alpha}$ for the output name, both of which are values. A *configuration* is a tuple $(M, \vec{\alpha}, \mu)$ of a term M, a sequence $\vec{\alpha}$ of generated channels and a sequence μ of performed send operations, i.e. $\mu = (\mathbf{send}\,\langle\bar{\beta}_1, V_1\rangle, \ldots, \mathbf{send}\,\langle\bar{\beta}_k, V_k\rangle)$. The *reduction relation* is defined by the following rules for channels

$$\begin{aligned}
(E[\mathbf{channel}\,\langle\rangle],\ \vec{\alpha},\ \mu) &\longrightarrow (E[\langle\beta, \bar{\beta}\rangle],\ \vec{\alpha}\cdot\beta,\ \mu) && (\beta \notin \vec{\alpha})\\
(E[\mathbf{send}\,\langle\bar{\beta}, V\rangle],\ \vec{\alpha},\ \mu) &\longrightarrow (E[\langle\rangle],\ \vec{\alpha},\ \mu\cdot\mathbf{send}\,\langle\bar{\beta}, V\rangle)\\
(E[\beta\, V],\ \vec{\alpha},\ \mu) &\longrightarrow (E[W\, V],\ \vec{\alpha},\ \mu) && (\mathbf{send}\,\langle\bar{\beta}, W\rangle \in \mu).
\end{aligned}$$

in addition to the standard rules for λ-abstractions and let-expressions, which change only M. Here the set of *evaluation contexts* is given by the grammar:

$$E ::= [] \mid \mathbf{let}\,\langle\vec{x}\rangle = E\,\mathbf{in}\,M \mid \mathbf{let}\,\langle\vec{x}\rangle = M\,\mathbf{in}\,E.$$

Note that M and N in $\mathbf{let}\,\langle\vec{x}\rangle = M\,\mathbf{in}\,N$ are evaluated in parallel (cf. Remark 3). This justifies the notation $M \parallel N$, an abbreviation for $\mathbf{let}\,\langle\rangle = M\,\mathbf{in}\,N$.

Axiomatic Semantics. The inference rules of the equational logic for λ_{ch} are those for λ_c with the rule of concurrent evaluation

$$\mathbf{let}\,\langle\vec{x}\rangle = M\,\mathbf{in}\,\mathbf{let}\,\langle\vec{y}\rangle = N\,\mathbf{in}\,L \quad = \quad \mathbf{let}\,\langle\vec{y}\rangle = N\,\mathbf{in}\,\mathbf{let}\,\langle\vec{x}\rangle = M\,\mathbf{in}\,L;$$

the β- and η-rules for channels

$$\begin{array}{rcl}(\boldsymbol{\nu} x\bar{x})(\mathbf{send}\,\langle\bar{x}, V\rangle \parallel M) & = & (\boldsymbol{\nu} x\bar{x})(\mathbf{send}\,\langle\bar{x}, V\rangle \parallel M\{V/x\}) \\ (\boldsymbol{\nu} y\bar{y})(\mathbf{send}\,\langle\bar{z}, y\rangle \parallel N) & = & N\{\bar{z}/\bar{y}\}\end{array}$$

where $\bar{x} \notin \mathbf{Fv}(V) \cup \mathbf{Fv}(M)$, $y \notin \mathbf{Fv}(N)$ and $\bar{z} \neq \bar{y}$; and a GC rule.

Categorical Semantics. One can interpret λ_{ch}-terms in a compact closed Freyd category with **(I)** and **(D)**. The interpretation of the λ_c-calculus part is standard [24,37]; the constant $\mathbf{channel}_\sigma$ (resp. $\mathbf{send}_\sigma$) is interpreted as the "closure" whose body is η_σ (resp. ϵ_σ) as expected.

$$\begin{array}{rcl}[\![\Gamma \vdash \mathbf{channel}_\sigma : () \to (\sigma, \sigma^*)]\!] & \stackrel{\text{def}}{=} & J(!_\Gamma; \Lambda_{I,I,\sigma\otimes\sigma^*}(\eta_\sigma)) \\ [\![\Gamma \vdash \mathbf{send}_\sigma : (\sigma^*, \sigma) \to ()]\!] & \stackrel{\text{def}}{=} & J(!_\Gamma; \Lambda_{I,\sigma\otimes\sigma^*,I}(\epsilon_\sigma)).\end{array}$$

The categorical semantics is sound and complete with respect to the equational theory of the λ_{ch}-calculus. The proofs are basically straightforward but there is a subtle issue in the definition of the term model: we have different definitions of the right adjoint $I \Rightarrow (-)$, which are of course equivalent but do not coincide on the nose. Our choice here is $I \Rightarrow \langle\vec{\xi}\rangle \stackrel{\text{def}}{=} (\vec{\xi}^{\perp}) \to ()$.

4.2 Translations Between λ_{ch} and π_F

The higher-order calculus λ_{ch} is equivalent to π_F. This is because both calculi correspond to the same class of categories, namely, the class of compact closed Freyd categories with **(I)** and **(D)**, i.e.,

$$(\lambda_{ch}) \;\approx\; (\text{compact closed Freyd category} + \mathbf{I} + \mathbf{D}) \;\approx\; (\pi_F).$$

This subsection studies translations derived from this semantic correspondence.

The translations are defined by the interpretations in the term models. For example, the translation $(\!|-|\!)$ from λ_{ch} to π_F is induced by the interpretation of λ_{ch}-terms in the term model $Cl(\emptyset)$. The interpretation $[\![M]\!]_{Cl(\emptyset)}$ of a λ_{ch}-term M is an equivalence class of π_F-processes, since a morphism in $Cl(\emptyset)$ is an equivalence class of π_F-processes. The translation $(\!|M|\!)$ is defined by choosing a representative of the equivalence class. The other direction $[\![-]\!]$ is obtained by the interpretation of π_F in the term model of λ_{ch}.

Figures 5 and 6 are concrete definitions of the translations for a natural choice of representatives. Let us discuss the translations in more details.

The translation from π_F to λ_{ch} (Fig. 5) is easy to understand. It directly expresses the higher-order view of the first-order π-calculus. For example, an

$$[\![\mathbf{ch}^o[\vec{T}]]\!] \stackrel{\text{def}}{=} [\![\vec{T}]\!] \to () \qquad [\![\mathbf{ch}^i[\vec{T}]]\!] \stackrel{\text{def}}{=} ([\![\vec{T}]\!] \to ())^* \qquad [\![(T_1, \ldots, T_n)]\!] \stackrel{\text{def}}{=} ([\![T_1]\!], \ldots, [\![T_n]\!])$$

$$[\![\mathbf{0}]\!] \stackrel{\text{def}}{=} \langle\rangle \qquad [\![P \mid Q]\!] \stackrel{\text{def}}{=} [\![P]\!] \parallel [\![Q]\!] \qquad [\![(\boldsymbol{\nu} xy)P]\!] \stackrel{\text{def}}{=} (\boldsymbol{\nu} xy)[\![P]\!]$$

$$[\![\bar{a}\langle\vec{x}\rangle]\!] \stackrel{\text{def}}{=} \bar{a}\,\langle\vec{x}\rangle \qquad [\![!a(\vec{x}).P]\!] \stackrel{\text{def}}{=} \mathbf{send}\,\langle a, \lambda(\vec{x}).[\![P]\!]\rangle$$

Fig. 5. Translation from π_F to λ_{ch}

$$(\!|\tau_1 \to \tau_2|\!) \stackrel{\text{def}}{=} \mathbf{ch}^o[(\!|\tau_1|\!), (\!|\tau_2|\!)^\perp] \qquad (\!|\sigma^*|\!) \stackrel{\text{def}}{=} (\!|\sigma|\!)^\perp \qquad (\!|(\tau_1, \ldots, \tau_n)|\!) \stackrel{\text{def}}{=} ((\!|\tau_1|\!), \ldots, (\!|\tau_n|\!))$$

$$(\!|x|\!)_p \stackrel{\text{def}}{=} (p \leftharpoonup x) \qquad (\!|\lambda\vec{x}.M|\!)_p \stackrel{\text{def}}{=} !p(\vec{x}, \vec{q}).(\!|M|\!)_{\vec{q}} \qquad (\!|\langle\vec{V}\rangle|\!)_{\vec{p}} \stackrel{\text{def}}{=} (\!|V_1|\!)_{p_1} \mid \cdots \mid (\!|V_n|\!)_{p_n}$$

$$(\!|V\,\langle\vec{W}\rangle|\!)_{\vec{p}} \stackrel{\text{def}}{=} (\boldsymbol{\nu} a\bar{a})(\boldsymbol{\nu}\vec{r}\vec{s})((\!|V|\!)_a \mid (\!|\langle\vec{W}\rangle|\!)_{\vec{s}} \mid \bar{a}\langle\vec{r}, \vec{p}\rangle)$$

$$(\!|\mathbf{let}\,\langle\vec{x}\rangle = M\,\mathbf{in}\,N|\!)_{\vec{p}} \stackrel{\text{def}}{=} (\boldsymbol{\nu}\vec{x}\vec{q})((\!|M|\!)_{\vec{q}} \mid (\!|N|\!)_{\vec{p}})$$

$$(\!|\mathbf{channel}|\!)_p \stackrel{\text{def}}{=} !p(x, y).x \hookrightarrow y \qquad (\!|\mathbf{send}|\!)_p \stackrel{\text{def}}{=} !p(x, y).x \hookrightarrow y$$

Fig. 6. Translation from λ_{ch} to π_F

output action is mapped to an application and an input-prefixing $!a(\vec{x}).P$ to a send operation of the value $\lambda\langle\vec{x}\rangle.P$ via the channel a.

An interesting (and perhaps confusing) phenomenon is that an input channel in π_F is mapped to an output channel in λ_{ch}. This can be explained as follows. In the name-passing viewpoint, the reduction

$$(\boldsymbol{\nu} xy)(!y(\vec{z}).P \mid x\langle\vec{u}\rangle) \quad \longrightarrow \quad (\boldsymbol{\nu} xy)(!y(\vec{z}).P \mid P\{\vec{u}/\vec{z}\})$$

sends $\vec{u}$ to the process $!y(\vec{z}).P$, and thus x is output and y is input. In the process-passing viewpoint, the abstraction $(\vec{z}).P$ is sent to the location of x, and thus y is the output and x is the input.

Next, we explain the translation from λ_{ch} to π_F (Fig. 6).

Let us first examine the translation of types. The most non-trivial part is the translation of a function type $\tau_1 \to \tau_2$. A key to understand the translation is the isomorphism $\tau_1 \to \tau_2 \cong \tau_1 \otimes \tau_2^\perp \to ()$. The latter form of function type corresponds to an output channel type in π_F. Hence a function is understood as a process additionally taking channels to which the return values are passed.

The translation $(\!|M|\!)_{\vec{p}}$ of a λ_{ch}-term $\Gamma \vdash M : (\xi_1, \ldots, \xi_n)$ takes extra parameters $\vec{p} = p_1, \ldots, p_n$ to which the values should be placed. This is a consequence of the definition in the π_F-term model that a morphism $\vec{T} \longrightarrow \vec{S}$ is a process $\vec{x}\colon \vec{T}, \vec{y}\colon \vec{S}^\perp \vdash P : \diamond$. Here $\vec{p}$ corresponds to $\vec{y}$, Γ to $\vec{x}\colon \vec{T}$ and $\vec{\xi}$ to $\vec{S}$.

Now it is not so difficult to understand the interpretations of constructs in the λ_c-calculus. For example, the abstraction $(\!|\lambda\langle\vec{x}\rangle.M|\!)_p$ is mapped to an abstraction $(\vec{x}, \vec{q}).(\!|M|\!)_{\vec{q}}$ placed at p, which takes additional channels $\vec{q}$ to which the results of the evaluation of M should be sent.

It might be surprising that the interpretations of **channel** and **send** coincide. This is because of the one-sided formulation of π_F. In the two-sided formulation, the unit η and counit ϵ of the compact closed structure, corresponding to **channel** and **send**, can be written as logical inference rules

$$(\!|\mathbf{0}|\!) \stackrel{\text{def}}{=} \mathbf{0} \qquad (\!|P \mid Q|\!) \stackrel{\text{def}}{=} (\!|P|\!) \mid (\!|Q|\!) \qquad (\!|(\boldsymbol{\nu} xy)P|\!) \stackrel{\text{def}}{=} (\boldsymbol{\nu} xy)(\!|P|\!) \qquad (\!|\,!x\,v|\!) \stackrel{\text{def}}{=} (\!|v|\!)_x$$

$$(\!|v\langle w_1, \ldots, w_n\rangle|\!) \stackrel{\text{def}}{=} (\boldsymbol{\nu}\bar{a}a)(\boldsymbol{\nu}\bar{b}_1 b_1)\ldots(\boldsymbol{\nu}\bar{b}_n b_n)((\!|v|\!)_a \mid (\!|w_1|\!)_{b_1} \mid \cdots \mid (\!|w_n|\!)_{b_n} \mid \bar{a}\langle \bar{b}_1, \ldots, \bar{b}_n\rangle)$$

$$(\!|x|\!)_a \stackrel{\text{def}}{=} (a \hookrightarrow x) \qquad (\!|(\vec{x}).P|\!)_a \stackrel{\text{def}}{=} !a(\vec{x}).(\!|P|\!)$$

Fig. 7. Translation from AHOπ to π_F

$$\frac{\Gamma, A, A^\perp \vdash \Delta}{\Gamma \vdash \Delta} \qquad \text{and} \qquad \frac{\Gamma \vdash A^\perp, A, \Delta}{\Gamma \vdash \Delta},$$

which are different. In the one-sided formulation, however, they become

$$\frac{\Gamma, A, A^\perp, \Delta^\perp \vdash}{\Gamma, \Delta^\perp \vdash}.$$

Hence η and ϵ (or **channel** and **send**) cannot be distinguished in π_F.

The translation $(\!|-|\!)$ must be the inverse of $[\![-]\!]$ because both the term models are the initial compact closed Freyd category with **(I)** and **(D)**. That means, $\emptyset \rhd \Gamma \vdash P = (\!|[\![P]\!]|\!)$ and $\emptyset \rhd \Gamma \vdash M = [\![(\!|M|\!)]\!]$ are provable for every P and M. This result is independent of the choice of representatives.

4.3 Relation to Other Calculi and Translations

A number of higher-order concurrent calculi, as well as their translations to the first-order π-calculus, have been proposed and studied (e.g. [29,39,40,42,45,47]). The calculus λ_{ch} and the translations have a lot of ideas in common with those calculi and translations; see Sect. 6.

This subsection mainly discusses the relationship to the translations by Sangiorgi [42] (see also [43]) between *asynchronous higher-order π-calculus* (*AHOπ* for short) and *asynchronous local π-calculus* (*Lπ* for short). Here we focus on this work because it is closest to ours. We shall see that our semantic or categorical development provides us with a semantic reconstruction of Sangiorgi's translations, as well as an extension.

A variant of AHOπ can be seen as a fragment of λ_{ch}. The syntax of processes of AHOπ and representation by λ_{ch}-terms are given as follow:

$$\begin{array}{lllllll} v, w & ::= x \mid & (\vec{x}).P & P, Q & ::= \mathbf{0} \mid (P \mid Q) \mid (\boldsymbol{\nu} xy)P \mid & !x\,v & \mid v\langle \vec{w}\rangle \\ & x & \lambda\langle \vec{x}\rangle.P & & \langle\rangle \quad P \parallel Q & (\boldsymbol{\nu} xy)P & \mathbf{send}\,\langle x, v\rangle \quad v\,\langle \vec{w}\rangle. \end{array}$$

(It slightly differs from the original syntax, as $\boldsymbol{\nu}$ binds a pair of names.)

This fragment is nicely described as the limitation on types:

$$\sigma ::= (\vec{\sigma}) \to () \qquad \xi ::= \sigma \mid \sigma^* \qquad \tau ::= ().$$

Recall that σ is a type for abstractions, ξ is a type for variables, and τ is a type for terms. This limitation means that (1) an abstraction cannot take a channel as an argument, and (2) a term M must be of the unit type, i.e. a process.

Once regarding AHOπ as a fragment of λ_{ch}, the translation from AHOπ to π_F is obtained by restricting $(\!|-|\!)$ to AHOπ. The resulting translation is in Fig. 7. As mentioned, the translation is the same as that of Sangiorgi [42] except for minor differences due to the slight change of the syntax.

Sangiorgi also gave a translation in the opposite direction, from Lπ to AHOπ in the same paper. The calculus Lπ is a fragment of the π-calculus in which only output channels can be passed. The **i**/**o**-separation of π_F allows us to characterise the local version of π_F by a limitation on types. In the local variant, the output channel type is restricted to $T ::= \mathbf{ch}^o[\vec{T}]$, expressing that only output channels can be passed via an output channel. Then the definition of type environment should be changed accordingly: $\Gamma ::= \cdot \mid x : T \mid x : T^{\perp}$ (since the syntactic class represented by T is not closed under the dual $(-)^{\perp}$ in the local setting).

Interestingly the limitation on types in AHOπ coincides with that in Lπ, when one identify $\mathbf{ch}^o[\vec{T}]$ with $(\vec{T}) \to ()$ (as we have done in many places). In other words, the syntactic restrictions of AHOπ and Lπ are the same semantic conditions described in different syntax. As a consequence, the image of Lπ by $[\![-]\!]$ is indeed in AHOπ.

Remark 4. There is, however, a notable difference from Sangiorgi's work [42]. Sangiorgi proved that the translation is fully-abstract with respect to barbed congruence; in contrast, we only show that $\vdash M = N$ iff $\vdash (\!|M|\!) = (\!|N|\!)$. In particular, the η-rule is inevitable for our argument. The presence of the η-rules significantly simplifies the argument, at the cost of operational justification (recall that the η-rule is not sound with respect to barbed congruence).

It is natural to ask how one can reconstruct the full-abstraction result with respect to barbed congruence. An interesting observation is that, if M and N are AHOπ processes, then $\vdash^{\ominus} M = N$ iff $\vdash^{\ominus} (\!|M|\!) = (\!|N|\!)$, where $\vdash^{\ominus}$ means provability without using η-rules. We expect that this semantic observation explains why locality is essential as noted in [42]; we leave the details for future work. □

5 Discussions

Connection to Logics. We have so far studied a connection between compact closed Freyd category and π-calculus. Here we briefly discuss the missing piece of the Curry-Howard-Lambek correspondence, namely logic.

The model of this paper is closely related to linear logic. Actually, every compact closed Freyd category is a model of linear logic (more precisely, MELL), as an instance of linear-non-linear model [6] (see, e.g., [27] for categorical models of linear logic). The interpretation of formulas is shown in Table 1. It differs from the translations by Abramsky [1] and Bellin and Scott [5] and from the Curry-Howard correspondence for session types by Caires and Pfenning [8], but resembles the connection between a variant of local π-calculus and a polarised linear logic by Honda and Laurent [19]; a detailed analysis of the translation is left for future work.

The logic corresponding to compact closed Freyd category should be a proper extension of linear logic, since compact closed Freyd categories form a proper

Table 1. The categorical and π_F-calculus interpretations of MELL formulas

linear logic (formula)	compact closed Freyd category (object)	π_F-calculus (type environment)
$A \otimes B$ $A \parr B$	$A \otimes B$	$x : A, y : B$
$!A$	$I \Rightarrow A$	$x : \mathbf{ch}^o[A^\perp]$
$?A$	$(A \Rightarrow I)^*$	$x : \mathbf{ch}^i[A]$

subclass of linear-non-linear models. For example, the following rules are invalid in linear logic but admissible in compact closed Freyd categories:

$$\frac{\vdash \Gamma \qquad \vdash \Delta}{\vdash \Gamma, \Delta} \qquad \frac{\vdash \Gamma, A, B \qquad \vdash \Delta, A^\perp, B^\perp}{\vdash \Gamma, \Delta} \qquad \frac{\vdash \Gamma, A, A^\perp}{\vdash \Gamma}.$$

These rules, especially the second rule called *multicut*, were often studied in concurrency theory; see Abramsky et al. [2] for their relevance to concurrency.

Do the above rules fill the gap between linear logic and compact closed Freyd category? Recent work by Hasegawa [15] suggests that MELL with above rules is still weaker than compact closed Freyd category. First observe that the above rules can be interpreted in any linear-non-linear model of which the monoidal category is compact closed. Hasegawa showed that a linear-non-linear model whose monoidal category is compact closed induces a closed Freyd category of which the monoidal category is *traced* (and vice versa) but the induced Freyd category is not necessarily compact closed. Hence the logic corresponding to compact closed Freyd category has further axioms or rules in addition to the above ones. A reasonable candidate for the additional axiom is $! \cong ?$; interestingly, Atkey et al. [3] reached a similar rule from a different perspective. Further investigation is left for future work.

Non-empty Signature. The categorical type theory for the λ-calculus considers a family parameterised by *signatures*, consisting of atomic types and constants. It covers, for example, the λ-calculus with natural number type and arithmetic constants (such as addition and multiplication), as well as a calculus with integer reference type and read and update functions.

Although this paper only considers the calculus with the empty signature, which has no additional type nor constant, extending our theory to handle non-empty signatures is, in a sense, not difficult. The easiest way is to apply the established theory of the computational λ-calculus [33,37]. As we have seen in Sect. 4, the π_F-calculus can be seen as a computational λ-calculus λ_{ch} having constants for manipulating channels; hence the π_F-calculus with additional constants is λ_{ch} with the additional constants, which is still in the family of computational λ-calculus.

The π_F-calculus with non-empty signature has several applications. We shall briefly discuss some of them.

An important example of π_F with non-empty signature is the calculus with non-replicated input, which we regard as a calculus with additional "process constants" but without any additional type. A key observation is that every non-replicated input process $a(\vec{x}).P$ can be expressed as

$$a(\vec{x}).P \;\cong^c\; (\boldsymbol{\nu}\bar{b}b)(a(\vec{x}).\bar{b}\langle\vec{x}\rangle \mid !b(\vec{x}).P) \qquad (\cong^c \text{ is weak barbed congruence})$$

and thus it suffices to deal with non-replicated input processes in special form, namely $a : \mathbf{ch}^i[\vec{T}],\ \bar{b} : \mathbf{ch}^o[\vec{T}] \vdash a(\vec{x}).\bar{b}\langle\vec{x}\rangle : \diamond$. Adding these processes as constants and the computational rules of $a(\vec{x}).\bar{b}\langle\vec{x}\rangle$ as equational axioms results in a calculus with non-replicated inputs. The categorical model is a compact closed Freyd category with distinguished morphisms $(A \Rightarrow I) \longrightarrow (A \Rightarrow I)$ for each object A which satisfy certain axioms.

This technique is applicable to synchronous output as well. Because

$$\bar{a}\langle\vec{x}\rangle.P \;\cong^c\; (\boldsymbol{\nu}\bar{b}b)(\bar{a}\langle\vec{x}\rangle.\bar{b}\langle\rangle \mid !b().P),$$

it suffices to consider constants representing $\bar{a} : \mathbf{ch}^o[\vec{T}], \vec{x} : \vec{T}, \bar{b} : \mathbf{ch}^o[] \vdash \bar{a}\langle\vec{x}\rangle.\bar{b}\langle\rangle : \diamond$.

6 Related Work

Logical Studies of π-calculi. There is a considerable amount of studies on connections between process calculi and linear logic. Here we divide these studies into two classes. These classes are substantially different; for example, one regards the formula $A \otimes B$ as a type for processes with two "ports" of type A and B, whereas the other as the session-type $!A.B$. Our work is more closely related to the former than the latter, but some interesting coincidence to the latter kind of studies can also be found.

The former class of research dates back to the work by Abramsky [1] and Bellin and Scott [5], where they discovered that π-calculus processes can encode proof-nets of classical linear logic. Later, Abramsky et al. [2] introduced the *interaction categories* to give a semantic description of a CCS-like process calculus. In their work, they observed that the compact closed structure is important to capture the strong expressive power of process calculi.

A tighter connection between π-calculus and proof-nets was recently presented by Honda and Laurent [19]. They showed that an **i**/**o**-typed π-calculus corresponds to *polarised proof-nets*, and introduced the notion of *extended reduction* for the π-calculus to simulate cut-elimination. The π-calculus used in this work is very similar to π_F in terms of syntax and reduction. Their calculus is asynchronous, does not allow non-replicated inputs, and requires **i**/**o**-separation. Furthermore, the extended reduction is almost the same as the rules (E-Beta) and (E-GC) except for the side conditions. A significant difference compared to our work is that their calculus is *local* [28,49], reflecting the fact that the corresponding logic is polarised.

Our work is inspired by these studies. The idea of **i**/**o**-separation can already be found in the work by Bellin and Scott and the use of compact closed category

is motivated by the study of interaction category. It is worth mentioning here that the design of π_F is also influenced by the calculus introduced by Laird [22], although it is not a logical study but categorical (see below).

The latter approach started with the Curry-Howard correspondences between session-typed π-calculi and linear logic established by Caires, Pfenning and Toninho [8,9] and subsequently by Wadler [48]. These correspondences are exact in the sense that every process has a corresponding proof, and vice versa. As a consequence, processes of the calculi inherit good properties of linear logic proofs such as termination and confluence of cut-elimination. In terms of process calculi, process of these calculi do not fall into deadlock or race condition. This can be seen as a serious restriction of expressive power [3,26,48].

Several extensions to increase the expressiveness of these calculi have been proposed and studied. Interestingly, ideas behind some of these extensions are related to our work, in particular to Sect. 5 discussing the multicut rule [2] and the axiom $! \cong ?$. Atkey et al. [3] studied *CP* [48] with the multicut rule and $! \cong ?$ and discussed how these extensions increase the expressiveness of the calculus, at the cost of losing some good properties of CP. Dardha and Gay [10] studied another extension of CP with multicut, keeping the calculus deadlock-free by an elaborated type system.

Balzer and Pfenning [4] proposed a session-typed calculus with shared (mutable) resources, inspired by linear-non-linear adjunction [6].

Categorical Semantics of π-calculi. The idea of using a closed Freyd category to model the π-calculus is strongly inspired by Laird [22]. He introduced the *distributive-closed Freyd category* to describe abstract properties of a game-semantic model of the asynchronous π-calculus and showed that distributive-closed Freyd categories with some additional structures suffice to interpret the asynchronous π-calculus. The additional structures are specific to his game model and not completely axiomatised.[7] Our notion of compact closed Freyd category might be seen as a reformulation of his idea, obtained by filtering out some structures difficult to axiomatise and by strengthening some others to make axioms simpler. A significant difference is that our categorical model does not deal with non-replicated inputs, which we think is essential for a simple axiomatisation.

Another approach for categorical semantics of the π-calculus has been the presheaf based approach [12,44]. These studies gave particular categories that nicely handles the nominal aspects of the π-calculus; these studies, however, do not aim for a correspondence between a categorical structure and the π-calculus.

Higher-Order Calculi with Channels. Besides the λ_{ch}-calculus, there are numbers of functional languages augmented by communication channels, from theoretical ones [13,25,46,48] to practical languages [34,38].

On the practical side, Concurrent ML (CML) [38], among others, is a well-developed higher-order concurrent language. CML has primitives to create channels and threads, and primitives to send and accept values through channels.

[7] A list of properties in [22] does not seem to be complete. We could not prove some claims in the paper only from these properties, but with ones specific to his model.

Since our λ_{ch}-calculus can create (non-linear) channels and send values via channels, the λ_{ch}-calculus can be seen as a core calculus of CML despite its origin in categorical semantics. The major difference between CML and the λ_{ch}-calculus is that communications in CML are synchronous whereas communications in the λ_{ch}-calculus are asynchronous.

On the theoretical side, session-typed functional languages have been actively studied [13,25,46,48]. Notably, some of these languages [25,46,48] are built upon the Curry-Howard foundation between linear logic and session-typed processes. It might be interesting to investigate whether we can relate these languages and the λ_{ch}-calculus through the lens of Curry-Howard-Lambek correspondence.

Higher-Order vs. First-Order π-calculus. A number of translations from higher-order languages to the π-calculus have been developed [39,40,42,45,47] since Milner [29] presented the encodings of the λ-calculus into the π-calculus. The basic idea shared by these studies is to transform $\lambda x.M$ to a process $!a(x,p).P$ that receives the argument x together with a name p where the rest of the computation will be transmitted. In our framework, this idea is described as the isomorphism $A \Rightarrow B \cong A \otimes B^* \Rightarrow I$.

Among others, the translation from AHOπ to Lπ [42] is the closest to our translation from the λ_{ch}-calculus to the π_F-calculus. Sangiorgi [41] observed that Milner's translation can be established via the translation of AHOπ by applying the CPS transformation to the λ-calculus. This observation also applies to our translation. That is, we can obtain Milner's translation by combining CPS transformation and the compilation of the λ_{ch}-calculus.

7 Conclusion and Future Work

We have introduced an **i**/**o**-typed π-calculus (π_F-calculus) as well as the categorical counterpart of π_F-calculus (compact closed Freyd category) and showed the categorical type theory correspondence between them. The correspondence was established by regarding the π-calculus as a higher-order programming language, introducing the **i**/**o**-separation, and introducing the η-rule, a rule that explains the mismatch between behavioural equivalences and categorical models.

As an application of our semantic framework we introduced a higher-order calculus λ_{ch}-calculus "equivalent" to the π_F-calculus. We have demonstrated that translations between λ_{ch}-calculus and π_F-calculus can be derived by a simple semantic argument, and showed that the translation from λ_{ch} to π_F is a generalisation of the translation from AHOπ to Lπ given by Sangiorgi [42].

There are three main directions for future work. First, further investigation on the η-rule is indispensable. We plan to construct a categorical model of the π_F-calculus with an additional constant that captures barbed congruence. Revealing the relationship between locality and the η-rule is another important problem. Second, the operational properties of the λ_{ch}-calculus and its relation to the equational theory needs a further investigation. Third, finding the logical counterpart of compact closed Freyd category to establish a proper Curry-Howard-Lambek correspondence is an interesting future work.

Acknowledgement. We would like to thank Naoki Kobayashi, Masahito Hasegawa and James Laird for discussions, and anonymous referees for valuable comments. This work was supported by JSPS KAKENHI Grant Number 15H05706 and 16K16004.

References

1. Abramsky, S.: Proofs as processes. Theor. Comput. Sci. **135**(1), 5–9 (1994)
2. Abramsky, S., Gay, S.J., Nagarajan, R.: Interaction categories and the foundations of typed concurrent programming. In: Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany, pp. 35–113 (1996)
3. Atkey, R., Lindley, S., Morris, J.G.: Conflation confers concurrency. In: A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, pp. 32–55 (2016)
4. Balzer, S., Pfenning, F.: Manifest sharing with session types. PACMPL **1**(ICFP), 37:1–37:29 (2017)
5. Bellin, G., Scott, P.J.: On the π-calculus and linear logic. Theor. Comput. Sci. **135**(1), 11–65 (1994)
6. Benton, P.N.: A mixed linear and non-linear logic: proofs, terms and models. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933, pp. 121–135. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0022251
7. Boreale, M.: On the expressiveness of internal mobility in name-passing calculi. Theor. Comput. Sci. **195**(2), 205–226 (1998)
8. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_16
9. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Math. Struct. Comput. Sci. **26**(3), 367–423 (2016)
10. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Baier, C., Dal Lago, U. (eds.) FoSSaCS 2018. LNCS, vol. 10803, pp. 91–109. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89366-2_5
11. de Nicola, R., Hennessy, M.C.B.: Testing equivalences for processes. In: Diaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 548–560. Springer, Heidelberg (1983). https://doi.org/10.1007/BFb0036936
12. Fiore, M.P., Moggi, E., Sangiorgi, D.: A fully abstract model for the π-calculus. Inf. Comput. **179**(1), 76–117 (2002)
13. Gay, S.J., Vasconcelos, V.T.: Linear type theory for asynchronous session types. J. Funct. Program. **20**(1), 19–50 (2010)
14. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987)
15. Hasegawa, M.: From linear logic to cyclic sharing. Lecture slides, Linearity (2018)
16. Hayashi, S.: Adjunction of semifunctors: categorical structures in nonextensional lambda calculus. Theor. Comput. Sci. **41**, 95–104 (1985)
17. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
18. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_35
19. Honda, K., Laurent, O.: An exact correspondence between a typed pi-calculus and polarised proof-nets. Theor. Comput. Sci. **411**(22–24), 2223–2238 (2010)

20. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053567
21. Kelly, G.M., Laplaza, M.L.: Coherence for compact closed categories. J. Pure Appl. Algebra **19**, 193–213 (1980)
22. Laird, J.: A game semantics of the asynchronous π-calculus. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 51–65. Springer, Heidelberg (2005). https://doi.org/10.1007/11539452_8
23. Lambek, J., Scott, P.J.: Introduction to Higher-Order Categorical Logic, vol. 7. Cambridge University Press, New York (1988)
24. Levy, P.B., Power, J., Thielecke, H.: Modelling environments in call-by-value programming languages. Inf. Comput. **185**(2), 182–210 (2003)
25. Lindley, S., Morris, J.G.: A semantics for propositions as sessions. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 560–584. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_23
26. Mazza, D.: The true concurrency of differential interaction nets. Math. Struct. Comput. Sci. **28**(7), 1097–1125 (2018)
27. Melliès, P.A.: Categorical semantics of linear logic. Panoramas et syntheses **27**, 15–215 (2009)
28. Merro, M.: Locality in the π-calculus and applications to distributed objects. Ph.D. thesis, École Nationale Supérieure des Mines de Paris (2000)
29. Milner, R.: Functions as processes. Math. Struct. Comput. Sci. **2**(2), 119–141 (1992)
30. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Inf. Comput. **100**(1), 1–40 (1992)
31. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, II. Inf. Comput. **100**(1), 41–77 (1992)
32. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55719-9_114
33. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS 1989), Pacific Grove, California, USA, 5–8 June 1989, pp. 14–23 (1989)
34. Peyton Jones, S.L., Gordon, A.D., Finne, S.: Concurrent Haskell. In: Conference Record of POPL 1996: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, 21–24 January 1996, pp. 295–308 (1996)
35. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Math. Struct. Comput. Sci. **6**(5), 409–453 (1996)
36. Power, J., Robinson, E.: Premonoidal categories and notions of computation. Math. Struct. Comput. Sci. **7**(5), 453–468 (1997)
37. Power, J., Thielecke, H.: Closed Freyd- and κ-categories. In: Wiedermann, J., van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 625–634. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48523-6_59
38. Reppy, J.H.: CML: a higher-order concurrent language. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, 26–28 June 1991, pp. 293–305 (1991)
39. Sangiorgi, D.: Expressing mobility in process algebras: first-order and higher-order paradigms. Ph.D. thesis, University of Edinburgh, UK (1993)

40. Sangiorgi, D.: π-Calculus, internal mobility, and agent-passing calculi. Theor. Comput. Sci. **167**(1&2), 235–274 (1996)
41. Sangiorgi, D.: From λ to π; or, rediscovering continuations. Math. Struct. Comput. Sci. **9**(4), 367–401 (1999)
42. Sangiorgi, D.: Asynchronous process calculi: the first- and higher-order paradigms. Theor. Comput. Sci. **253**(2), 311–350 (2001)
43. Sangiorgi, D., Walker, D.: The π-calculus—A Theory of Mobile Processes. Cambridge University Press, New York (2001)
44. Stark, I.: A fully abstract domain model for the π-calculus. In: Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, 27–30 July 1996, pp. 36–42 (1996)
45. Toninho, B., Caires, L., Pfenning, F.: Functions as session-typed processes. In: Birkedal, L. (ed.) FoSSaCS 2012. LNCS, vol. 7213, pp. 346–360. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28729-9_23
46. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: a monadic integration. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 350–369. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_20
47. Turner, D.N.: The polymorphic Pi-calculus: theory and implementation. Ph.D. thesis, University of Edinburgh, UK (1996)
48. Wadler, P.: Propositions as sessions. J. Funct. Program. **24**(2–3), 384–418 (2014)
49. Yoshida, N.: Minimality and separation results on asynchronous mobile processes - representability theorems by concurrent combinators. Theor. Comput. Sci. **274**(1–2), 231–276 (2002)

A Process Algebra for Link Layer Protocols

Rob van Glabbeek[1,2(✉)], Peter Höfner[1,2], and Michael Markl[1,3]

[1] Data61, CSIRO, Sydney, Australia
rvg@cs.stanford.edu
[2] Computer Science and Engineering, University of New South Wales, Sydney, Australia
[3] Institut für Informatik, Universität Augsburg, Augsburg, Germany

Abstract. We propose a process algebra for link layer protocols, featuring a unique mechanism for modelling frame collisions. We also formalise suitable liveness properties for link layer protocols specified in this framework. To show applicability we model and analyse two versions of the Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol. Our analysis confirms the hidden station problem for the version without virtual carrier sensing. However, we show that the version with virtual carrier sensing not only overcomes this problem, but also the exposed station problem with probability 1. Yet the protocol cannot guarantee packet delivery, not even with probability 1.

1 Introduction

The (data) link layer is the 2nd layer of the ISO/OSI model of computer networking [18]. Amongst others, it is responsible for the transfer of data between adjacent nodes in Wide Area Networks (WANs) and Local Area Networks (LANs).

Examples of link layer protocols are Ethernet for LANs [16], the Point-to-Point Protocol [24] and the High-Level Data Link Control protocol (e.g. [14]). Part of this layer are also multiple access protocols such as the Carrier-Sense Multiple Access with Collision Detection (CSMA/CD) protocol for re-transmission in Ethernet bus networks and hub networks, or the Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol [17,19] in wireless networks.

One of the unique characteristics of the link layer is that when devices attempt to use a medium simultaneously, *collisions of messages* occur. So, any modelling language and formal analysis of layer-2 protocols has to support such collisions. Moreover, some protocols are of probabilistic nature: CSMA/CA for example chooses time slots probabilistically with discrete uniform distribution.

As we are not aware of any formal framework with primitives for modelling data collisions, this paper introduces a process algebra for modelling and analysing link layer protocols. In Sect. 2 we present an algebra featuring a unique mechanism for modelling collisions, 'hard-wired' in the semantics. It is the non-probabilistic fragment of the Algebra for Link Layer protocols (ALL), which we

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 668–693, 2019.
https://doi.org/10.1007/978-3-030-17184-1_24

introduce in Sect. 3. In Sect. 4 we formulate *packet delivery*, a liveness property that ideally ought to hold for link layer protocols, either outright, or with a high probability. In Sect. 5 we use this framework to formally model and analyse the CSMA/CA protocol.

Our analysis confirms the hidden station problem for the version of CSMA/CA without virtual carrier sensing (Sect. 5.2). However, we also show that the version with virtual carrier sensing overcomes not only this problem, but also the exposed station problem with probability 1. Yet the protocol cannot guarantee packet delivery, not even with probability 1.

2 A Non-probabilistic Subalgebra

In this section we propose a timed process algebra that can model the collision of link layer messages, called *frames*.[1] It can be used for link layer protocols that do not feature probabilistic choice, and is inspired by the (Timed) Algebra for Wireless Networks ((T-)AWN) [2,12,13], a process algebra suitable for modelling and analysing protocols on layers 3 (network) and 4 (transport) of the OSI model.

The process algebra models a (wired or wireless) network as an encapsulated parallel composition of network nodes. Due to the nature of the protocols under consideration, on each node exactly one sequential process is running. The algebra features a discrete model of time, where each sequential process maintains a local variable `now` holding its local clock value—an integer. We employ only one clock for each sequential process. All sequential processes in a network synchronise in taking time steps, and at each time step all local clocks advance by one unit. Since this means that all clocks are in sync and do not run at different speeds it is clear that we do not consider the problem of clock shift. For the rest, the variable `now` behaves like any other variable maintained by a process: its value can be read when evaluating guards, thereby making progress time-dependant, and any value can be assigned to it, thereby resetting the local clock. Network nodes communicate with their direct neighbours—those nodes that are in transmission range. The algebra provides a mobility option that allows nodes to move in or out of transmission range. The encapsulation of the entire network inhibits communications between network nodes and the outside world, with the exception of the receipt and delivery of data packets from or to clients (the higher OSI layers).

2.1 A Language for Sequential Processes

The internal state of a process is determined, in part, by the values of certain data variables that are maintained by that process. To this end, we assume a data structure with several types, variables ranging over these types, operators and predicates. Predicate logic yields terms (or *data expressions*) and formulas

[1] As it is the nonprobabilistic fragment of a forthcoming algebra we do not name it.

to denote data values and statements about them. Our data structure always contains the types $\mathtt{TIME}$, $\mathtt{DATA}$, $\mathtt{MSG}$, $\mathtt{CHUNK}$, $\mathtt{ID}$ and $\mathscr{P}(\mathtt{ID})$ of discrete *time values*, which we take to be integers, *network layer data*, *messages*, *chunks* of messages that take one time unit to transmit, *node identifiers* and *sets of node identifiers*. We further assume that there are variables $\mathtt{now}$ of type $\mathtt{TIME}$ and $\mathtt{rfr}$ of type $\mathtt{CHUNK}$. In addition, we assume a set of *process names*. Each process name X comes with a *defining equation*

$$X(\mathtt{var}_1, \ldots, \mathtt{var}_n) \stackrel{def}{=} P\,,$$

in which $n \in \mathbb{N}$, $\mathtt{var}_i$ are variables and P is a *sequential process expression* defined by the grammar below. It may contain the variables $\mathtt{var}_i$ as well as X. However, all occurrences of data variables in P have to be *bound*.[2] The choice of the underlying data structure and the process names with their defining equations can be tailored to any particular application of our language.

The *sequential process expressions* are given by the following grammar:

$$P ::= X(exp_1, \ldots, exp_n) \mid [\varphi]P \mid [\![\mathtt{var} := exp]\!]P \mid \alpha.P \mid P + P$$
$$\alpha ::= \mathbf{transmit}(ms) \mid \mathbf{newpkt}(\mathtt{data}, \mathtt{dest}) \mid \mathbf{deliver}(\mathtt{data})$$

Here X is a process name, exp_i a data expression of the same type as $\mathtt{var}_i$, φ a data formula, $\mathtt{var} := exp$ an assignment of a data expression exp to a variable $\mathtt{var}$ of the same type, ms a data expression of type $\mathtt{MSG}$, and $\mathtt{data}$, $\mathtt{dest}$ data variables of types $\mathtt{DATA}$, $\mathtt{ID}$ respectively.

Given a valuation of the data variables by concrete data values, the sequential process $[\varphi]P$ acts as P if φ evaluates to $\mathtt{true}$, and deadlocks if φ evaluates to $\mathtt{false}$. In case φ contains free variables that are not yet interpreted as data values, values are assigned to these variables in any way that satisfies φ, if possible. The process $[\![\mathtt{var} := exp]\!]P$ acts as P, but under an updated valuation of the data variable $\mathtt{var}$. The process $P + Q$ may act either as P or as Q, depending on which of the two processes is able to act at all. In a context where both are able to act, it is not specified how the choice is made. The process $\alpha.P$ first performs the action α and subsequently acts as P. The above behaviour is identical to AWN, and many other standard process algebras. The action $\mathbf{transmit}(ms)$ transmits (the data value bound to the expression) ms to all other network nodes within transmission range. The action $\mathbf{newpkt}(\mathtt{data}, \mathtt{dest})$ models the injection by the network layer of a data packet $\mathtt{data}$ to be transmitted to a destination $\mathtt{dest}$. Technically, $\mathtt{data}$ and $\mathtt{dest}$ are variables that will be bound to the obtained values upon receipt of a **newpkt**. Data is delivered to the network layer by $\mathbf{deliver}(\mathtt{data})$. In contrast to AWN, we do not have a primitive for

[2] An occurrence of a data variable in P is *bound* if it is one of the variables $\mathtt{var}_i$, one of the two special variables $\mathtt{now}$ or $\mathtt{rfr}$, a variable $\mathtt{var}$ occurring in a subexpression $[\![\mathtt{var} := exp]\!]Q$, an occurrence in a subexpression $[\varphi]Q$ of a variable occurring free in φ, or a variable $\mathtt{data}$ or $\mathtt{dest}$ occurring in a subexpression $\mathbf{newpkt}(\mathtt{data}, \mathtt{dest}).Q$. Here Q is an arbitrary sequential process expression.

receiving messages from neighbouring nodes, because our processes are *always* listening to neighbouring nodes, in parallel with anything else they do.

As in AWN, the internal state of a sequential process described by an expression P is determined by P, together with a *valuation* ξ associating values $\xi(\mathtt{var})$ to variables $\mathtt{var}$ maintained by this process. Valuations naturally extend to ξ-*closed* expressions—those in which all variables are either bound or in the domain of ξ. We denote the valuation that assigns the value v to the variable $\mathtt{var}$, and agrees with ξ on all other variables, by $\xi[\mathtt{var} := v]$. The valuation $\xi_{|S}$ agrees with ξ on all variables $\mathtt{var} \in S$ and is undefined otherwise. Moreover we use $\xi[\mathtt{var}\,{+\!\!+}]$ as an abbreviation for $\xi[\mathtt{var} := \xi(\mathtt{var}) + 1]$, for suitable types.

To capture the durational nature of transmitting a message between network nodes, we model a message as a sequence of *chunks*, each of which takes one time unit to transmit. The function $\mathtt{dur} : \mathtt{MSG} \to \mathtt{TIME}_{>0}$ calculates the amount of time steps needed for a sending a message, i.e. it calculates the number of chunks. We employ the internal data type $\mathtt{CHUNK} := \{m{:}c \mid m \in \mathtt{MSG}, 1 \leq c \leq \mathtt{dur}(m)\} \cup \{\underline{\mathtt{conflict}}, \underline{\mathtt{idle}}\}$. The chunk $m{:}c$ indicates the cth fragment of a message m. Data conflicts—junk transmitted via the medium—is modelled by the special chunk $\underline{\mathtt{conflict}}$, and the absence of an incoming chunk is modelled by $\underline{\mathtt{idle}}$.

Our process algebra maintains a variable $\mathtt{rfr}$ of type $\mathtt{CHUNK}$, storing the fragment of the current message received so far. As a value of this variable, $m{:}c$ indicates that the first c chunks of message m have been received in order; $\underline{\mathtt{conflict}}$ indicates that the last incoming chunk was not the expected (next) part of a message in progress, and $\underline{\mathtt{idle}}$ indicates that the channel was idle during the last time step. The table on the right, with $*$ a wild card, shows how the value of $\mathtt{rfr}$ evolves upon receiving a new chunk ch.

$\mathtt{rfr}$	ch	$\mathtt{rfr} \star ch$
$*$	$\underline{\mathtt{conflict}}$	$\underline{\mathtt{conflict}}$
$*$	$\underline{\mathtt{idle}}$	$\underline{\mathtt{idle}}$
$*$	$m{:}1$	$m{:}1$
$m{:}c$	$m{:}c{+}1$	$m{:}c{+}1$
rfr	$m{:}c{+}1$	$\underline{\mathtt{conflict}}$
		if $rfr \neq m{:}c$

Specifications may refer to the data type $\mathtt{CHUNK}$ only through the Boolean functions NEW—having a single argument msg of type $\mathtt{MSG}$—and IDLE, defined by $\text{NEW}(msg) := (\mathtt{rfr} = (msg : \mathtt{dur}(msg)))$ and $\text{IDLE} := (\mathtt{rfr} = \underline{\mathtt{idle}})$. A guard $[\text{NEW}(msg)]$ evaluates to true iff a new message msg has just been received; [IDLE] evaluates to true iff in the last time slice the medium was idle.

The structural operational semantics of Table 1 describes how one internal state can evolve into another by performing an *action*. The set Act of actions consists of $\mathbf{transmit}(m{:}c, ch)$, $\mathbf{wait}(ch)$, $\mathbf{newpkt}(d, dest)$, $\mathbf{deliver}(d)$, and internal actions τ, for each choice of $m \in \mathtt{MSG}$, $c \in \{1, \ldots, \mathtt{dur}(m)\}$, $ch \in \mathtt{CHUNK}$, $d \in \mathtt{DATA}$ and $dest \in \mathtt{ID}$, where the first two actions are time consuming. On every time-consuming action, each process receives a chunk ch and updates the variable $\mathtt{rfr}$ accordingly; moreover, the variable $\mathtt{now}$ is incremented on all process expressions in a (complete) network synchronously.

Besides the special variables $\mathtt{now}$ and $\mathtt{rfr}$, the formal semantics employs an internal variable $\mathtt{cntr} \in \mathbb{N}$ that enumerates the chunks of split messages and is

Table 1. Structural operational semantics for sequential process expressions

(1) $\xi, \mathbf{transmit}(ms).P \xrightarrow{\mathbf{transmit}(\xi(ms):\mathtt{c+},ch)} \xi\begin{bmatrix}\mathtt{cntr}\,{+\!\!+}\\ \mathtt{rfr} := \mathtt{rfr} \star ch\\ \mathtt{now}\,{+\!\!+}\end{bmatrix}, \mathbf{transmit}(\xi(ms)).P$ $(\text{if } \mathtt{c+} < \mathtt{dur}(\xi(ms)))$ $(\forall ch \in \mathtt{CHUNK})$

(2) $\xi, \mathbf{transmit}(ms).P \xrightarrow{\mathbf{transmit}(\xi(ms):\mathtt{c+},ch)} \xi\begin{bmatrix}\mathtt{cntr} := 0\\ \mathtt{rfr} := \mathtt{rfr} \star ch\\ \mathtt{now}\,{+\!\!+}\end{bmatrix}, P$ $(\text{if } \mathtt{c+} = \mathtt{dur}(\xi(ms)))$ $(\forall ch \in \mathtt{CHUNK})$

(3) $\xi, \mathbf{newpkt}(\mathtt{data}, \mathtt{dest}).P \xrightarrow{\mathbf{newpkt}(d,dest)} \xi\begin{bmatrix}\mathtt{data} := d\\ \mathtt{dest} := dest\end{bmatrix}, P$ $(\forall d \in \mathtt{DATA},\ dest \in \mathtt{ID})$

(4) $\xi, \mathbf{newpkt}(\mathtt{data}, \mathtt{dest}).P \xrightarrow{\mathbf{wait}(ch)} \xi\begin{bmatrix}\mathtt{rfr} := \mathtt{rfr} \star ch\\ \mathtt{now}\,{+\!\!+}\end{bmatrix}, \mathbf{newpkt}(\mathtt{data}, \mathtt{dest}).P$ $(\forall ch \in \mathtt{CHUNK})$

(5) $\xi, \mathbf{deliver}(\mathtt{data}).P \xrightarrow{\mathbf{deliver}(\xi(\mathtt{data}))} \xi, P$

(6) $\xi, [\![\mathtt{var} := exp]\!]P \xrightarrow{\tau} \xi\lceil \mathtt{var} := \xi(exp) \rceil, P$

(7) $$\frac{\xi_{|\mathtt{RO}}\lceil \mathtt{var}_i := \xi(exp_i)\rceil\ {}_{i=1}^{n}, P \xrightarrow{a} \zeta, P'}{\xi, X(exp_1, \ldots, exp_n) \xrightarrow{a} \zeta, P'}\ (X(\mathtt{var}_1, \ldots, \mathtt{var}_n) \stackrel{def}{=} P)$$ $(\forall a \in \mathrm{Act} - \{\mathbf{wait}(ch) \mid ch \in \mathtt{CHUNK}\})$

(8) $$\frac{\xi_{|\mathtt{RO}}\lfloor \mathtt{var}_i := \xi(exp_i)\rfloor\ {}_{i=1}^{n}, P \xrightarrow{\mathbf{wait}(ch)} \zeta, P'}{\xi, X(exp_1, \ldots, exp_n) \xrightarrow{\mathbf{wait}(ch)} \xi\begin{bmatrix}\mathtt{rfr} := \mathtt{rfr} \star ch\\ \mathtt{now}\,{+\!\!+}\end{bmatrix}, X(exp_1, \ldots, exp_n)}\ (X(\mathtt{var}_1, \ldots, \mathtt{var}_n) \stackrel{def}{=} P)$$ $(\forall ch \in \mathtt{CHUNK})$

(9) $\xi, P \xrightarrow{\mathbf{wait}(ch)} \xi\begin{bmatrix}\mathtt{rfr} := \mathtt{rfr} \star ch\\ \mathtt{now}\,{+\!\!+}\end{bmatrix}, P$ $(\text{if } \xi(P)\!\uparrow)$ $(\forall ch \in \mathtt{CHUNK})$

(10) $$\frac{\xi, P \xrightarrow{a} \zeta, P'}{\xi, P + Q \xrightarrow{a} \zeta, P'} \quad \frac{\xi, Q \xrightarrow{a} \zeta, Q'}{\xi, P + Q \xrightarrow{a} \zeta, Q'}$$ $(\forall a \in \mathrm{Act} - \{\mathbf{wait}(ch) \mid ch \in \mathtt{CHUNK}\})$

(11) $$\frac{\xi, P \xrightarrow{\mathbf{wait}(ch)} \zeta, P' \quad \xi, Q \xrightarrow{\mathbf{wait}(ch)} \zeta', Q'}{\xi, P + Q \xrightarrow{\mathbf{wait}(ch)} \zeta, P' + Q'}$$ $(\forall ch \in \mathtt{CHUNK})$

(12) $$\frac{\xi \xrightarrow{\varphi} \zeta}{\xi, [\varphi]P \xrightarrow{\tau} \zeta, P} \qquad \frac{\xi \not\xrightarrow{\varphi}}{\xi, [\varphi]P \xrightarrow{\mathbf{wait}(ch)} \xi\begin{bmatrix}\mathtt{rfr} := \mathtt{rfr} \star ch\\ \mathtt{now}\,{+\!\!+}\end{bmatrix}, [\varphi]P}$$ $(\forall ch \in \mathtt{CHUNK})$

used to identify which chunk needs to be sent next. The variables `now`, `rfr` and `cntr` are not meant to be changed by ALL specifications, e.g. by using assignments. We call them read-only and collect them in the set $\texttt{RO} = \{\texttt{now}, \texttt{rfr}, \texttt{cntr}\}$.

Let us have a closer look at the rules of Table 1.

The first two rules describe the sending of a message *ms*. Remember that $\texttt{dur}(ms)$ calculates the time needed to send *ms*. The counter `cntr` keeps track of the time passed already. The action $\mathbf{transmit}(m{:}c, ch)$ occurs when the node transmits the fragment $m{:}c$; simultaneously, it receives the fragment *ch*.[3] The counter `cntr` is 0 before a message is sent, and is incremented before the transmission of each chunk. So, each chunk sent has the form $\xi(ms){:}\xi(\texttt{cntr})+1$. To ease readability we abbreviate $\xi(\texttt{cntr})+1$ by `c+`. In case the (already incremented) counter `c+` is strictly smaller than the number of chunks needed to send $\xi(ms)$, another **transmit**-action is needed (Rule 1); if the last fragment has been sent ($\texttt{c+} = \texttt{dur}(\xi(ms))$) the process can continue to act as P (Rule 2).

The actions $\mathbf{newpkt}(d, dest)$ and $\mathbf{deliver}(d)$ are instantaneous and model the submission of data d from the network layer, destined for *dest*, and the delivery of data d to the network layer, respectively. The process $\mathbf{newpkt}(d, dest).P$ has also the possibility to wait, namely if no network layer instruction arrives.

Rule 6 defines a rule for assignment in a straightforward fashion; only the valuation of the variable `var` is updated.

In Rules 7 and 8, which define recursion, $\xi_{|\texttt{RO}}[\texttt{var}_i := \xi(exp_i)]_{i=1}^{n}$ is the valuation that *only* assigns the values $\xi(exp_i)$ to the variables $\texttt{var}_i$, for $i = 1, \ldots, n$, and maintains the values of the variables `now`, `rfr` and `cntr`. These rules state that a defined process X has the same transitions as the body p of its defining equation. In case of a **wait**-transition, the sequential process does not progress, and accordingly the recursion is not yet unfolded.

Most transition rules so far feature statements of the form $\xi(exp)$ where *exp* is a data expression. The application of the rule depends on $\xi(exp)$ being defined. Rule 9 covers all cases where the above rules cannot be applied since at least one data expression in an action α is not defined. A state ξ, P is *unvalued*, denoted by $\xi(p)\uparrow$, if P has the form $\mathbf{transmit}(ms).P$, $\mathbf{deliver}(\texttt{data}).P$, $[\![\texttt{var} := exp]\!]P$ or $X(exp_1, \ldots, exp_n)$ with either $\xi(ms)$ or $\xi(\texttt{data})$ or $\xi(exp)$ or some $\xi(exp_i)$ undefined. From such a state the process can merely wait.

A process $P + Q$ can wait *only* if both P and Q can do the same; if either P or Q can achieve 'proper' progress, the choice process $P + Q$ always chooses progress over waiting. A simple induction shows that if $\xi, P \xrightarrow{\mathbf{wait}(ch)} \zeta, P'$ and $\xi, Q \xrightarrow{\mathbf{wait}(ch)} \zeta', Q'$ then $P = P'$, $Q = Q'$ and $\zeta = \zeta'$.

The first rule of (12), describing the semantics of guards $[\varphi]$, is taken from AWN. Here $\xi \xrightarrow{\varphi} \zeta$ says that ζ is an extension of ξ, i.e. a valuation that agrees with ξ on all variables on which ξ is defined, and evaluates other variables occurring free in φ, such that the formula φ holds under ζ. All variables not free in φ and not evaluated by ξ are also not evaluated by ζ. Its negation $\xi \not\xrightarrow{\varphi}$ says

[3] Normally, a node is in its own transmission range. In that case the received chunk *ch* will be either the chunk $m{:}c$ it is transmitting itself, or `conflict` in case some other node within transmission range is transmitting as well.

that no such extension exists, and thus, that φ is false in the current state, no matter how we interpret the variables whose values are still undefined. If that is the case, the process $[\varphi]p$ will idle by performing the action $\mathbf{wait}(ch)$.

2.2 A Language for Node Expressions

We model network nodes in the context of a (wireless) network by *node expressions* of the form

$$id{:}(\xi, P){:}R\,.$$

Here $id \in \mathtt{ID}$ is the *address* of the node, P is a sequential process expression with a valuation ξ, and $R \in \mathscr{P}(\mathtt{ID})$ is the *range* of the node, defined as the set of nodes within transmission range of id. Unlike AWN, the process algebra does not offer a parallel operator for combining sequential processes; such an operator is not needed due to the nature of link layer protocols.

In the semantics of this layer it is crucial to handle frame collisions. The idea is that all chunks sent are recorded, together with the respective recipient. In case a node receives more than one chunk at a time, a conflict is raised, as it is impossible to send two or more messages via the same medium at the same time.

The formal semantics for node expressions, presented in Table 2, uses transition labels $\mathbf{traffic}(\mathcal{T},\mathcal{R})$, $id{:}\,\mathbf{deliver}(d)$, $id{:}\,\mathbf{newpkt}(d,id')$, $\mathbf{connect}(id,id')$, $\mathbf{disconnect}(id,id')$ and τ, with partial functions $\mathcal{T},\mathcal{R}:\mathtt{ID} \rightharpoonup \mathtt{CHUNK}$, $id, id' \in \mathtt{ID}$, and $d \in \mathtt{DATA}$.

Table 2. Structural operational semantics for node expressions

$$\frac{P \xrightarrow{\mathbf{wait}(\mathtt{idle})} P'}{id{:}P{:}R \xrightarrow{\mathbf{traffic}(\emptyset,\emptyset)} id{:}P'{:}R} \qquad \frac{P \xrightarrow{\mathbf{transmit}(m:c,\mathtt{idle})} P'}{id{:}P{:}R \xrightarrow{\mathbf{traffic}(\{(r,m:c)\,|\,r\in R\},\emptyset)} id{:}P'{:}R}$$

$$\frac{P \xrightarrow{\mathbf{wait}(ch)} \quad (ch\neq\mathtt{idle})}{id{:}P{:}R \xrightarrow{\mathbf{traffic}(\emptyset,\{(id,\,ch)\})} id{:}P'{:}R} \qquad \frac{P \xrightarrow{\mathbf{transmit}(m:c,ch)} P' \quad (ch\neq\mathtt{idle})}{id{:}P{:}R \xrightarrow{\mathbf{traffic}(\{(r,m:c)\,|\,r\in R\},\{(id,\,ch)\})} id{:}P'{:}R}$$

$$\frac{P \xrightarrow{\mathbf{deliver}(d)} P'}{id{:}P{:}R \xrightarrow{id:\mathbf{deliver}(d)} id{:}P'{:}R} \qquad \frac{P \xrightarrow{\mathbf{newpkt}(d,dest)} P'}{id{:}P{:}R \xrightarrow{id:\mathbf{newpkt}(d,dest)} id{:}P'{:}R} \qquad \frac{P \xrightarrow{\tau} P'}{id{:}P{:}R \xrightarrow{\tau} id{:}P'{:}R}$$

$$id{:}P{:}R \xrightarrow{\mathbf{connect}(id,id')} id{:}P{:}R \cup \{id'\} \qquad id{:}P{:}R \xrightarrow{\mathbf{disconnect}(id,id')} id{:}P{:}R - \{id'\}$$

$$id{:}P{:}R \xrightarrow{\mathbf{connect}(id',id)} id{:}P{:}R \cup \{id'\} \qquad id{:}P{:}R \xrightarrow{\mathbf{disconnect}(id',id)} id{:}P{:}R - \{id'\}$$

$$\frac{id \notin \{id', id''\}}{id{:}P{:}R \xrightarrow{\mathbf{connect}(id',id'')} id{:}P{:}R} \qquad \frac{id \notin \{id', id''\}}{id{:}P{:}R \xrightarrow{\mathbf{disconnect}(id',id'')} id{:}P{:}R}$$

All time-consuming actions on process level ($\mathbf{transmit}(m{:}c,ch)$ and $\mathbf{wait}(ch)$) are transformed into an action $\mathbf{traffic}(\mathcal{T},\mathcal{R})$ on node level: the first argument

Table 3. Structural operational semantics for network expressions

$$\frac{M \xrightarrow{a} M'}{M\|N \xrightarrow{a} M'\|N} \qquad \frac{N \xrightarrow{a} N'}{M\|N \xrightarrow{a} M\|N'} \qquad \frac{M \xrightarrow{a} M'}{[M] \xrightarrow{a} [M']} \qquad \left(\forall a \in \left\{\begin{array}{l}\tau,\, id:\mathbf{deliver}(d),\\ id:\mathbf{newpkt}(d,\, id),\end{array}\right\}\right)$$

$$\frac{M \xrightarrow{a} M' \quad N \xrightarrow{a} N'}{M\|N \xrightarrow{a} M'\|N'} \qquad \frac{M \xrightarrow{a} M'}{[M] \xrightarrow{a} [M']} \qquad \left(\forall a \in \left\{\begin{array}{l}\mathbf{connect}(id,\, id'),\\ \mathbf{disconnect}(id,\, id')\end{array}\right\}\right)$$

$$\frac{M \xrightarrow{\mathbf{traffic}(\mathcal{T}_1,\mathcal{R}_1)} M' \quad N \xrightarrow{\mathbf{traffic}(\mathcal{T}_2,\mathcal{R}_2)} N'}{M\|N \xrightarrow{\mathbf{traffic}(\mathcal{T}_1 \uplus \mathcal{T}_2,\mathcal{R}_1 \uplus \mathcal{R}_2)} M'\|N'} \qquad \frac{M \xrightarrow{\mathbf{traffic}(\mathcal{R},\mathcal{R})} M'}{[M] \xrightarrow{\mathbf{tick}} [M']}$$

$\mathcal{T}$ maps *dest* to *m*:*c* if and only if the chunk *m*:*c* is transmitted to *dest*. The second argument $\mathcal{R}$ maps *id* to *m*:*c* if and only if the chunk *m*:*c* is received on process level at node *id*. For the sos-rules of Table 2 we use the set-theoretic presentation of partial functions. The two rules for **wait** set $\mathcal{T} := \emptyset$, as no chunks are transmitted; the rules for **transmit** allow a transmitted chunk *m*:*c* to travel to all nodes within transmission range: $\mathcal{T} := \{(r, m{:}c) \,|\, r \in R\}$. In case that during the transmission or waiting no chunk is received ($ch = \underline{\texttt{idle}}$) we set $\mathcal{R} = \emptyset$; otherwise $\mathcal{R} = \{(id, ch)\}$, indicating that chunk *ch* is received by node *id*.

The actions $id{:}\,\mathbf{newpkt}(d, dest)$ and $id{:}\,\mathbf{deliver}(d)$ as well as the internal actions τ are simply inherited by node expressions from the processes that run on these nodes.

The remaining rules of Table 2 model the mobility aspect of wireless networks; the rules are taken straight from AWN [12,13]. We allow actions $\mathbf{connect}(id, id')$ and $\mathbf{disconnect}(id, id')$ for $id, id' \in \texttt{ID}$ modelling a change in network topology. These actions can be thought of as occurring nondeterministically, or as actions instigated by the environment of the modelled network protocol. In this formalisation node id' is in the range of node *id*, meaning that id' can receive messages sent by *id*, if and only if *id* is in the range of id'. To break this symmetry, one just skips the last four rules of Table 2 and replaces the synchronisation rules for **connect** and **disconnect** in Table 3 by interleaving rules (like the ones for **deliver**, **newpkt** and τ) [12]. For some applications a wired or non-mobile network need to be considered. In such cases the last six rules of Table 2 are dropped.

Whether a node $id{:}\,P{:}\,R$ receives its own transmissions depends on whether $id \in R$. Only if $id \in R$ our process algebra will disallow the transmission from and to a single node *id* at the same time, yielding a $\underline{\texttt{conflict}}$.

2.3 A Language for Networks

A *partial network* is modelled by a *parallel composition* $\|$ of node expressions, one for every node in the network. A *complete network* is a partial network within an *encapsulation operator* $[\_]$, which limits the communication between network nodes and the outside world to the receipt and delivery of data packets to and from the network layer.

The syntax of networks is described by the following grammar:

$$N ::= [M_T^T] \qquad M_{S_1 \dot\cup S_2}^T ::= M_{S_1}^T \| M_{S_2}^T \qquad M_{\{id\}}^T ::= id{:}(\xi, P){:}R \ ,$$

with $\{id\} \cup R \subseteq T \subseteq \mathtt{ID}$. Here M_S^T models a partial network describing the behaviour of all nodes $id \in S$. The set T contains the identifiers of all nodes that are part of the complete network. This grammar guarantees that node identifiers of node expressions—the first component of $id{:}P{:}R$—are unique.

The operational semantics of network expressions is given in Table 3. Internal actions τ as well as the actions $id{:}\,\mathbf{deliver}(d)$ and $id{:}\,\mathbf{newpkt}(d,id)$ are interleaved in the parallel composition of nodes that makes up a network, and then lifted to encapsulated networks (Line 1 of Table 3).

Actions **traffic** and (**dis**)**connect** are synchronised. The rule for synchronising the action **traffic** (Line 3), the only action that consumes time on the network layer, uses the union $\uplus$ of partial functions. It is formally defined as

$$(\mathcal{R}_1 \uplus \mathcal{R}_2)(id) := \begin{cases} \underline{\mathtt{conflict}} & \text{if } id \in \mathtt{dom}(\mathcal{R}_1) \cap \mathtt{dom}(\mathcal{R}_2) \\ \mathcal{R}_1(id) & \text{if } id \in \mathtt{dom}(\mathcal{R}_1) - \mathtt{dom}(\mathcal{R}_2) \\ \mathcal{R}_2(id) & \text{if } id \in \mathtt{dom}(\mathcal{R}_2) - \mathtt{dom}(\mathcal{R}_1) \ . \end{cases}$$

The synchronisation of the sets $\mathcal{R}_i$ and $\mathcal{T}_i$ has the following intuition: if a node identifier $id \in \mathtt{ID}$ is in both $\mathtt{dom}(\mathcal{T}_1)$ and $\mathtt{dom}(\mathcal{T}_2)$ then there exist two nodes that transmit to node id at the same time, and therefore a frame collision occurs. In our algebra this is modelled by the special chunk $\underline{\mathtt{conflict}}$. The sos rules of Tables 2 and 3 guarantee that there cannot be collisions within the set of received chunks $\mathcal{R}$. The reason is that each node merely contributes to $\mathcal{R}$ a chunk for itself; it can be the chunk $\underline{\mathtt{conflict}}$ though. Therefore we could have written $\mathcal{R}_1 \cup \mathcal{R}_2$ instead of $\mathcal{R}_1 \uplus \mathcal{R}_2$ in the sixth rule of Table 3.

The last rule propagates a $\mathbf{traffic}(\mathcal{T}, \mathcal{R})$-action of a partial network M to a complete network $[M]$. By then $\mathcal{T}$ consists of all chunks (after collision detection) that are being transmitted by any member in the network, and $\mathcal{R}$ consists of all chunks that are received. The condition $\mathcal{R} = \mathcal{T}$ determines the content of the messages in $\mathcal{R}$. The $\mathbf{traffic}(\mathcal{T}, \mathcal{R})$-actions become internal at this level, as they cannot be steered by the outside world; all that is left is a time-step **tick**.

2.4 Results on the Process Algebra

As for the process algebra T-AWN [2], but with a slightly simplified proof, one can show that our processes have no *time deadlocks*:

Theorem 2.1. *A complete network N in our process algebra always admits a transition, independently of the outside environment, i.e. $\forall N, \exists a$ such that $N \xrightarrow{a}$ and $a \notin \{\boldsymbol{connect}(id, id'), \boldsymbol{disconnect}(id, id'), id{:}\mathtt{newpkt}(d, dest)\}$.*
More precisely, either $N \xrightarrow{\mathbf{tick}}$, or $N \xrightarrow{id:\,\boldsymbol{deliver}(d)}$ or $N \xrightarrow{\tau}$.

The following results (statements and proofs) are very similar to the results about the process algebra AWN, as presented in [13]. A rich body of foundational

meta theory of process algebra allows the transfer of the results to our setting, without too much overhead work.

Identical to AWN and its timed version T-AWN, our process algebra admits a translation into one without data structures (although we cannot describe the target algebra without using data structures). The idea is to replace any variable by all possible values it can take. The target algebra differs from the original only on the level of sequential processes; the subsequent layers are unchanged. The construction closely follows the one given in the appendix of [2]. The inductive definition contains the rules

$\mathscr{T}_\xi(\mathbf{deliver}(data).P) = \mathbf{deliver}(\xi(data)).\mathscr{T}_\xi(P)$ and

$\mathscr{T}_\xi([\![\mathtt{var} := exp]\!]P) = \tau.\mathscr{T}_{\xi[\mathtt{var} := \xi(exp)]}(P)$.

Most other rules require extra operators that keep track of the passage of time and the evolution of other internal variables. The resulting process algebra has a structural operational semantics in the (infinitary) *de Simone* format, generating the same transition system—up to strong bisimilarity, $\leftrightarrows$—as the original. It follows that $\leftrightarrows$, and many other semantic equivalences, are congruences on our language [23].

Theorem 2.2. *Strong bisimilarity is a congruence for all operators of our language.*

This is a deep result that usually takes many pages to establish (e.g. [25]). Here we get it directly from the existing theory on structural operational semantics, as a result of carefully designing our language within the disciplined framework described by de Simone [23]. □

Theorem 2.3. *The operator* $\|$ *is associative and commutative, up to* $\leftrightarrows$.

Proof. The operational rules for this operator fits a format presented in [6], guaranteeing associativity up to $\leftrightarrows$. The *A*SSOC-de Simone format of [6] applies to all transition system specifications (TSSs) in de Simone format, and allows 7 different types of rules (named 1–7) for the operators in question. Our TSS is in de Simone format; the four rules for $\|$ of Table 3 are of types 1, 2 and 7, respectively. To be precise, it has rules 1_a and 2_a for $a \in \{\tau, id\colon \mathbf{deliver}(d), id\colon \mathbf{newpkt}(d, dest)\}$, rules $7_{(a,b)}$ for

$$(a,b) \in \{(\mathbf{traffic}(\mathcal{T}_1, \mathcal{R}_1), \mathbf{traffic}(\mathcal{T}_2, \mathcal{R}_2)) \mid \mathcal{R}_1, \mathcal{R}_2, \mathcal{T}_1, \mathcal{T}_2 \in \mathtt{ID} \rightharpoonup \mathtt{CHUNK}\}$$

and rules $7_{(c,c)}$ for $c \in \{\mathbf{connect}(id, id'), \mathbf{disconnect}(id, id') \mid id, id' \in \mathtt{ID}\}$. Moreover, the partial *communication function* $\gamma : \mathrm{Act} \times \mathrm{Act} \rightharpoonup \mathrm{Act}$ is given by $\gamma(\mathbf{traffic}(\mathcal{T}_1, \mathcal{R}_1), \mathbf{traffic}(\mathcal{T}_2, \mathcal{R}_2)) = \mathbf{traffic}(\mathcal{T}_1 \uplus \mathcal{T}_2, \mathcal{R}_1 \uplus \mathcal{R}_2)$ and $\gamma(c, c) = c$. The main result of [6] is that an operator is guaranteed to be associative, provided that γ is associative and six conditions are fulfilled. In the absence of rules of types 3, 4, 5 and 6, five of these conditions are trivially fulfilled, and the remaining one reduces to

$$7_{(a,b)} \Rightarrow (1_a \Leftrightarrow 2_b) \wedge (2_a \Leftrightarrow 2_{\gamma(a,b)}) \wedge (1_b \Leftrightarrow 1_{\gamma(a,b)})\,.$$

Here 1_a says that rule 1_a is present, etc. This condition is trivially met for $\|$ as there neither exists a rule of the form $1_{\mathbf{traffic}(\mathcal{T},\mathcal{R})}$ nor of the form $2_{\mathbf{traffic}(\mathcal{T},\mathcal{R})}$, or 1_c, 2_c with c as above. As on **traffic** actions γ is basically the union of partial functions ($\uplus$), where a collision in domains is indicated by an error $\underline{\texttt{conflict}}$, it is straightforward to prove associativity of γ.

Commutativity of $\|$ follows by symmetry of the sos rules. □

3 An Algebra for Link Layer Protocols

We now introduce ALL, the *Algebra for Link Layer protocols*. It is obtained from the process algebra presented in the previous section by the addition of a probabilistic choice operator $\bigoplus_0^n$. As a consequence, the semantics of the algebra is no longer a labelled transition system, but a *probabilistic labelled transition system* (pLTS) [8]. This is a triple $(S, \mathrm{Act}, \rightarrow)$, where

(i) S is a set of states
(ii) Act is a set of actions
(iii) $\rightarrow \subseteq S \times \mathrm{Act} \times \mathcal{D}(S)$, where $\mathcal{D}(S)$ is the set of all (discrete) probability distributions over S: functions $\Delta : S \rightarrow [0,1]$ with $\sum_{s \in S} \Delta(s) = 1$.

As with LTSs, we usually write $s \xrightarrow{\alpha} \Delta$ instead of $(s, \alpha, \Delta) \in \rightarrow$. The *point distribution* δ_s, for $s \in S$, is the distribution with $\delta_s(s) = 1$. We simply write $s \xrightarrow{\alpha} t$ for $s \xrightarrow{\alpha} \delta_t$. An LTS may be viewed as a degenerate pLTS, in which only point distributions occur. For a uniform distribution over $s_0, \ldots, s_n \in S$ we write $\mathcal{U}_{i=0}^n s_i$. The pLTS associated to ALL takes S to be the disjoint union of the pairs ξ, P, with P a sequential process expression, and the network expressions. Act is the collection of transition labels, and $\rightarrow$ consists of the transitions derivable from the structural operational semantics of the language.

Rules (1)–(6), (9), (11) and (12) of Table 1 are adopted to ALL unchanged, whereas in Rules (7), (8) and (10) the state ζ, P' (or ζ, Q') is replaced by an arbitrary distribution Δ. Add to those the following rule for the probabilistic choice operator:

$$\xi, \bigoplus_{\mathtt{i}=0}^{n} P \xrightarrow{\tau} \mathcal{U}_{i=0}^{\xi(n)} \xi[\mathtt{i} := i], P$$

Here the data variable i may occur in P. The rules of Tables 2 and 3 are adapted to ALL unchanged, except that P', M' and N' are now replaced by arbitrary distributions over sequential processes and network expressions, respectively. Here we adapt the convention that a unary or binary operation on states lifts to distributions in the standard manner. For example, if Δ is a distribution over sequential processes, $id \in \mathtt{ID}$ and $R \subseteq \mathtt{ID}$, then $id{:}\Delta{:}R$ describes the distribution over node expressions that only has probability mass on nodes with address id and range R, and for which the probability of $id{:}P{:}R$ is $\Delta(P)$. Likewise, if Δ and Θ are distributions over network expressions, then $\Delta \| \Theta$ is the distribution over network expressions of the form $M \| N$, where $(\Delta \| \Theta)(M \| N) = \Delta(M) \cdot \Theta(N)$.

4 Formalising Liveness Properties of Link Layer Protocols

Link layer protocols communicate with the network layer through the actions $id\colon\mathbf{newpkt}(d, dest)$ and $id\colon\mathbf{deliver}(d)$. The typical liveness property expected of a link layer protocol is that if the network layer at node id injects a data packet d for delivery at destination $dest$ then this packet is delivered eventually. In terms of our process algebra, this says that every execution of the action $id\colon\mathbf{newpkt}(d, dest)$ ought to be followed by the action $dest\colon\mathbf{deliver}(d)$. This property can be formalised in Linear-time Temporal Logic [22] as

$$\mathbf{G}\big(id\colon\mathbf{newpkt}(d, dest) \Rightarrow \mathbf{F}(dest\colon\mathbf{deliver}(d))\big) \tag{1}$$

for any $id, dest \in \mathtt{ID}$ and $d \in \mathtt{DATA}$. This formula has the shape $\mathbf{G}\big(\phi^{pre} \Rightarrow \mathbf{F}\phi^{post}\big)$, and is called an *eventuality property* in [22]. It says that whenever we reach a state in which the precondition ϕ^{pre} is satisfied, this state will surely be followed by a state were the postcondition ϕ^{post} holds. In [7,13] it is explained how action occurrences can be seen or encoded as state-based conditions. Here we will not define how to interpret general LTL-formula in pLTSs, but below we do this for eventuality properties with specific choices of ϕ^{pre} and ϕ^{post}.

Formula (1) is too strong and does not hold in general: in case the nodes id and $dest$ are not within transmission range of each other, the delivery of messages from id to $dest$ is doomed to fail. We need to postulate two side conditions to make this liveness property plausible. Firstly, when the request to deliver the message comes in, id needs to be connected to $dest$. We introduce the predicate $\mathbf{cntd}(id, dest)$ to express this, and hence take ϕ^{pre} to be $\mathbf{cntd}(id, dest) \wedge id\colon\mathbf{newpkt}(d, dest)$. Secondly, we assume that the link between id and $dest$ does not break until the message is delivered. As remarked in [13], such a side condition can be formalised by taking ϕ^{post} to be $dest\colon\mathbf{deliver}(d) \vee \mathbf{disconnect}(id, dest)$. Thus the liveness property we are after is

$$\begin{array}{c}\mathbf{G}\big(\mathbf{cntd}(id, dest) \wedge id\colon\mathbf{newpkt}(d, dest) \Rightarrow \\ \mathbf{F}(dest\colon\mathbf{deliver}(d) \vee \mathbf{disconnect}(id, dest) \vee \mathbf{disconnect}(dest, id))\big)\end{array} \tag{2}$$

We now define the validity of eventuality properties $\mathbf{G}\big(\phi^{pre} \Rightarrow \mathbf{F}\phi^{post}\big)$. Here ϕ^{pre} and ϕ^{post} denote sets of transitions and actions, respectively, and hold if one of the transitions or actions in the set occurs. In (2), ϕ^{pre} denotes the transitions with label $id\colon\mathbf{newpkt}(d, dest)$ that occur when the side condition $\mathbf{cntd}(id, dest)$ is met, whereas $\phi^{post} = \{dest\colon\mathbf{deliver}(d), \mathbf{disconnect}(id, dest), \mathbf{disconnect}(dest, id)\}$ is a set of actions.

A *path* in a pLTS $(S, \mathrm{Act}, \rightarrow)$ is an alternating sequence $s_0, \alpha_1, s_1, \alpha_2, \ldots$ of states and actions, starting with a state and either being infinite or ending with a state, such that there is a transition $s_i \xrightarrow{\alpha_{i+1}} \Delta_{i+1}$ with $\Delta_{i+1}(s_{i+1}) > 0$ for each i. The path is *rooted* if it starts with a state marked as 'initial', and *complete* if either it is infinite, or there is no transition starting from its last state. A state or transition is *reachable* if it occurs in a rooted path.

In a pLTS with an initial state, an eventually formula $\mathbf{G}\big(\phi^{pre} \Rightarrow \mathbf{F}\phi^{post}\big)$, with ϕ^{pre} and ϕ^{post} denoting sets of transitions and actions, *holds outright* if all complete paths starting with a reachable transition from ϕ^{pre} contain a transition with a label from ϕ^{post}.

Definitions 3 and 5 in [9] define the set of probabilities that a pLTS with an initial state will ever execute the action ω. One obtains a set of probabilities rather than a single probability due to the possibility of nondeterministic choice. This definition generalises to *sets* of actions ϕ^{post} (seen as disjunctions) by first renaming all actions in such a set into ω. It also generalises trivially to pLTSs with an *initial transition*. For t a transition in a pLTS, let $Prob(t, \phi^{post})$ be the infimum of the set of probabilities that the pLTS in which t is taken to be the initial transition will ever execute ϕ^{post}. Now in a pLTS with an initial state, an eventually formula $\mathbf{G}\big(\phi^{pre} \Rightarrow \mathbf{F}\phi^{post}\big)$ *holds with probability at least* p if for all reachable transitions t in ϕ^{pre} we have $Prob(t, \phi^{post}) \geq p$.

Possible correctness criteria for link layer protocols are that the liveness property (2) either holds outright, holds with probability 1, or at least holds with probability p for a sufficiently high value of p.

Sometimes we are content to establish that (2) holds under the additional assumptions that the network is stable until our packet is delivered, meaning that no links between any nodes are broken or established, and/or that the network layer refrains from injecting more packets. This is modelled by taking

$$\phi^{post} = \{dest\colon \mathbf{deliver}(d), \mathbf{disconnect}(*,*), \mathbf{connect}(*,*), \mathbf{newpkt}(*,*)\}. \tag{3}$$

We will refer to this version of (2) as the *weak packet delivery* property. *Packet delivery* is the strengthening without $\mathbf{newpkt}(*,*)$ in (3), i.e. not assuming that the network layer refrains from injecting more packets.

5 Modelling and Analysing the CSMA/CA Protocol

In this section we model two versions of the CSMA/CA protocol, using the process algebra ALL. Moreover, we briefly discuss some results we obtained while analysing these protocols.

The *Carrier-Sense Multiple Access* (CSMA) protocol is a media access control (MAC) protocol in which a node verifies the absence of other traffic before transmitting on a shared transmission medium. If a carrier is sensed, the node waits for the transmission in progress to end before initiating its own transmission. Using CSMA, multiple nodes may, in turn, send and receive on the same medium. Transmissions by one node are generally received by all other nodes connected to the medium.

The CSMA protocol with Collision Avoidance (CSMA/CA) [17,19][4] improves the performance of CSMA. If the transmission medium is sensed busy

[4] The primary medium access control (MAC) technique of IEEE 802.11 [19] is called *distributed coordination function* (DCF), which is a CSMA/CA protocol.

before transmission then the transmission is deferred for a *random* time interval. This interval reduces the likelihood that two or more nodes waiting to transmit will simultaneously begin transmission upon termination of the detected transmission. CSMA/CA is used, for example, in Wi-Fi.

It is well known that CSMA/CA suffers from the *hidden station problem* (see Sect. 5.2). To overcome this problem, CSMA/CA is often supplemented by the request-to-send/clear-to-send (RTS/CTS) handshaking [19]. This mechanism is known as the IEEE 802.11 RTS/CTS exchange, or *virtual carrier sensing*. While this extension reduces the amount of collisions, wireless 802.11 implementations do not typically implement RTS/CTS for all transmissions because the transmission overhead is too great for small data transfers.

We use the process algebra ALL to model both the CSMA/CA without and with virtual carrier sensing.

5.1 A Formal Model for CSMA/CA

Our formal specification of CSMA/CA consists of four short processes written in ALL. It is precise and free of ambiguities—one of the many advantages formal methods provide, in contrast to specifications written in English prose.

The syntax of ALL is intended to look like pseudo code, and it is our belief that the specification can easily be read and understood by software engineers, who may or may not have experience with process algebra.

As the underlying data structure of our model is straightforward, we do not present it explicitly, but introduce it while describing the different processes.

The basic process `CSMA`, depicted in Process 1, is the protocol's entry point.

Process 1. The Basic Routine

```
CSMA(id) =def
1. newpkt(data,dest). INIT(id,0,dataframe(data,id,dest))
2. + [NEW(dataframe(data,src,id))] deliver(data) .
3.    (
4.       ⟦timeout := now + sifs⟧ [now ≥ timeout]
5.       transmit(ackframe(src)) . CSMA(id)
6.    )
```

This process maintains a single data variable `id` in which it stores its own identity. It waits until either it receives a request from the network layer to transmit a packet `data` to destination `dest`, or it receives from another node in the network a CSMA message (data frame) destined for itself.

In case of a newly injected data packet (Line 1), the process `INIT` is called; this process (described below) initiates the sending of the message via the medium. When passing the message on to `INIT` we use a function $\texttt{dataframe} : \texttt{DATA} \times \texttt{ID} \times \texttt{ID} \to \texttt{MSG}$ that generates a message in a format used by the protocol: next to the header fields (from which we abstract) it contains the injected `data` as well as the designated receiver `dest` and the sender `id`—the current node.

In case of an incoming `dataframe` destined for this node (the third argument carrying the destination is `id`) (Line 2)—any other incoming message is ignored by this process—the `data` is handed over to the network layer (**deliver**(`data`)) followed by the transmission of an acknowledgement back to the sender of the message (`src`). CSMA/CA requires a short period of idling medium before sending the acknowledgement: in [19] this interval is called *short interframe space* (`sifs`). The process waits until the time of the interframe spacing has passed, and then transmits the acknowledgement. The acknowledgement sent is not always received by `src`, e.g. due to data collision; therefore `src` could send the same message again (see Process 4) and `id` could deliver the same data to the network layer again.

Process 2. Protocol Initialisation

$\texttt{INIT}(\texttt{id},\texttt{tries},\texttt{dframe}) \stackrel{def}{=}$

1. $[\texttt{tries} \leq \texttt{max\_retransmit}]$
2. $[\![\texttt{cw} := \texttt{cwmin} \times 2^{\texttt{tries}}]\!]$
3. $\bigoplus_{\texttt{b}=0}^{\texttt{cw}-1} \texttt{CCA}(\texttt{id},\texttt{b},\texttt{tries},\texttt{dframe})$ /* choose a backoff from $\{0,\ldots,\texttt{cw}-1\}$ */
4. $+\ [\texttt{tries} > \texttt{max\_retransmit}]$
5. **deliver**(`channel_access_failure`) . $\texttt{CSMA}(\texttt{id})$

The process `INIT` (Process 2) initiates the sending of a message via the medium. Next to the variable `id`, which is maintained by all processes, it maintains the variable `tries` and `dframe`: `tries` stores the number of attempts already made to send message `dframe`. When the process is called the first time for a message `dframe` (Line 1 of Process 1) the value of `tries` is 0.

The constant `max_retransmit` specifies the maximum number of attempts the protocol is allowed to retransmit the same message. If the limit is not yet reached (Line 1) the message `dframe` is sent. As mentioned above, CSMA/CA defers messages for a *random* time interval to avoid collision. The node must start transmission within the contention window `cw`, a.k.a. backoff time. `cw` is calculated in Line 2; it increases exponentially.[5] After `cw` is determined, the process `CCA` is called, which performs the actual **transmit**-action. In case the maximum number of retransmits is reached (Line 4), the process notifies the network layer and restarts the protocol, awaiting new instructions from the application layer, or a new incoming message.

Process 3 takes care of the actual transmission of `dframe`. However, the protocol has a complicated procedure when to send this message.

First, the process senses the medium and awaits the point in time when it is idle (Line 6). In case, before this happens, it receives from another node in the network a CSMA message destined for itself (Line 1), this message is handled just as in Process 1, except that after acknowledging this message the protocol returns to Process 3.

[5] A typical value for `cwmin` is 16; it must satisfy $\texttt{cwmin} > 0$.

Process 3. Clear Channel Assessment With Physical Carrier Sense

```
CCA(id,b,tries,dframe) =def
 1. [NEW(dataframe(data,src,id))] deliver(data) .
 2.   (
 3.     ⟦timeout := now + sifs⟧ [now ≥ timeout]
 4.     transmit(ackframe(src)) . CCA(id,b,tries,dframe)
 5.   )
 6. + [IDLE]
 7.   ⟦timeout:=now+difs⟧      /* start wait for duration difs */
 8.   (
 9.     [¬IDLE] CCA(id,b,tries,dframe)
10.     + [IDLE ∧ now ≥ timeout]
11.       ⟦timeout := now + b⟧
12.       (
13.         [¬IDLE]      /* busy during backoff time */
14.           ⟦b := timeout − now⟧ CCA(id,b,tries,dframe)
15.         + [IDLE ∧ now ≥ timeout]      /* idle for backoff time */
16.           transmit(dframe) .
17.           ACKRECV(id,tries,now+max_ack_wait,dframe)
18.       )
19.   )
```

To guarantee a gap between messages sent via the medium, CSMA/CA (as well as other protocols) specifies the *distributed (coordination function) interframe space* (difs ∈ TIME), which is usually small,[6] but larger than sifs, so that acknowledgements get priority over new data frames. When the medium becomes busy during the interframe space, another node started transmitting and the process goes back to listening to the medium (Line 9). In case nothing happens on the medium and the end of the interframe space is reached (Line 10), the process determines the actual time to start transmitting the message, taking the backoff time b into account (Line 11). If the medium is idle for the entire backoff period (Line 15), the message is transmitted (Line 16), and the process calls the process ACKRECV that will await an acknowledgement from the recipient of dframe (Line 17); the third argument specifies the maximum time the process should wait for such an acknowledgement. (As mentioned before an acknowledgement may never arrive.) If another node transmits on the medium during the backoff period, the protocol restarts the routine (Lines 13 and 14), with an adjusted backoff value b—the process already started waiting and should not be punished when the waiting is restarted; this update guarantees fairness of the protocol.

The process awaiting an acknowledgement (Process 4) is straightforward. It waits until either it receives a CSMA message destined for itself (Line 1), or it receives an acknowledgement (Line 6), or it has waited for this acknowledgement as long as it is going to (Line 8).

[6] Recommended values for the constant difs are given in [19].

In the first case, the message is handled just as in Process 1, except that after acknowledging this message the protocol returns to Process 4. In the second case the network layer is informed that the sending of `dframe` was successful and the process loops back to Process 1 (Line 7). Line 8 describes the situation where no acknowledgement message arrives and the process times out. Here CSMA/CA retries to send the message; the counter `tries` is incremented.

Process 4. Receiving an ACK

```
ACKRECV(id,tries,acktimeout,dframe) =def
1. [NEW(dataframe(data,src,id))] deliver(data) .
2.   (
3.     ⟦timeout := now + sifs⟧ [now ≥ timeout]
4.     transmit(ackframe(src)) . ACKRECV(id,tries,acktimeout,dframe)
5.   )
6. + [NEW(ackframe(id))]      /* acknowledgement received */
7.   deliver(success) . CSMA(id)
8. + [now ≥ acktimeout] INIT(id,tries+1,dframe)
```

5.2 The Hidden Station Problem

As mentioned in the introduction to this section, CSMA/CA suffers from the hidden station problem. This refers to the situation where two nodes A and C are not within transmission range of each other, while a node B is in range of both. In this situation C may be transmitting to B, but A is not able to sense this, and thus may start a transmission to B at roughly the same time, leading to data collisions at B.

While CSMA/CA is not able to avoid such collisions as a whole—it is always possible that two (or more) nodes hidden from each other happen to (randomly) choose the same backoff time to send messages—it is the exponential growth of the backoff slots that makes the problem less pressing in the long run, as the following theorem shows.

Theorem 5.1. If $\underline{\texttt{max\_retransmit}}=\infty$ then weak packet delivery holds with probability 1.

Proof sketch. Since the number of messages that nodes transmit is bounded, and all nodes select random times to start transmitting out of an increasing longer time span, with probability 1 each message will eventually go through. □

In practice, $\underline{\texttt{max\_retransmit}}$ is set to a value that is not high enough to approximate the idea behind the above proof. In fact, the transmission time of a single message may be larger than the maximal backoff period allowed. For this reason the hidden station problem does occur when running the CSMA/CA protocol, as studies have shown [5]. Nevertheless, the above analysis still shows that link layer protocols can be formally analysed by process algebra in general, and ALL in particular.

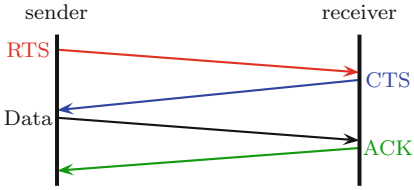


Fig. 1. RTS/CTS exchange

5.3 A Formal Model for CSMA/CA with Virtual Carrier Sensing

To overcome the hidden station problem the usage of a request-to-send/clear-to-send (RTS/CTS) handshaking [19] mechanism is available. This mechanism is also known as *virtual carrier sensing*. The exchange of RTS/CTS messages happens just before the actual data is sent, see Fig. 1. The mechanism serves two purposes: (a) As the RTS and CTS messages are very short—they only contain two node identifiers as well as a natural number indicating the time it will take to send the actual `data` (plus overhead)—the likelihood of a collision is reduced. (b) While the handshaking does not help with solving the hidden station problem for the RTS message itself, it avoids the problem for the sending of `data`. The reason is that a hidden node, which could interfere with the sending of `data` will receive the CTS message from the designated recipient of `data`, and the hidden node will remain silent until the `data` has been sent.

As for the CSMA/CA protocol we have modelled this extension in ALL, based on the model of CSMA/CA we presented earlier.

Our extended model uses two functions to generate `rts` and `cts` messages, respectively. The signature of both is $\texttt{ID} \times \texttt{ID} \times \texttt{TIME} \to \texttt{MSG}$. The first argument carries the sender (source) of the message, the second the indented destination, and the third argument a duration (time period) of silence that is requested/granted. For example, before the message `rts(src,dest,d)` is transmitted, the time period `d` is calculated by
The calculation is straightforward as it follows the protocol logic and determines the amount of time needed until the acknowledgement would be received (see Fig. 2). After the `rts` message has been received the medium should be idle for the interframe space `sifs`; then a `cts` message is sent back, which takes time `dur_cts`; then another interframe space is needed, followed by the actual transmission of the message—the sending will take `dur(dataframe(data,id,dest))` time units; after the message is received (hopefully) another interframe space is required before the acknowledgement is sent back.

$$[\![\texttt{d} := \texttt{sifs} + \texttt{dur\_cts} + \texttt{sifs} + \texttt{dur}(\texttt{dataframe}(\texttt{data},\texttt{id},\texttt{dest})) + \texttt{sifs} + \texttt{dur\_ack}]\!] .$$

Process 2 remains essentially unchanged; it is merely equipped with the destination `dest` of the message that needs to be transmitted, and an additional timed variable $\texttt{nav} \in \texttt{TIME}$. These variables are not used in this process, but required later on. Variable `nav` holds the point in time until the process should

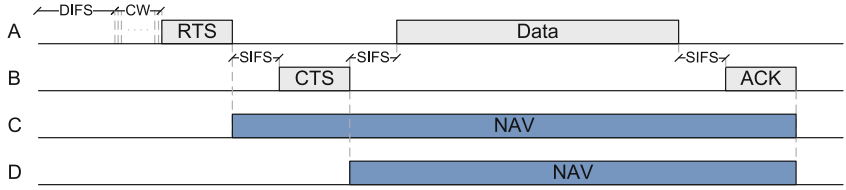


Fig. 2. The use of virtual channel sensing using CSMA/CA [3]

not transmit any `rts` or `cts` message. This period of silence is necessary as the node figures out that until time `nav` another node will transmit message(s).[7]

Process 5 is the modified version of Process 1. Identical to Process 1 it awaits an instruction from the network layer, or an incoming CSMA message destined for itself. Lines 1–3 are identical to Process 1. Lines 4–11 handle the two new message types. In case an `rts` message `rts(src,dest,d)` is received that is intended for another recipient (`dest` $\neq$ `id`) the node concludes that another node wants to use the medium for the amount of `d` time units; the process updates the variable `nav` if needed, indicating the period the node should remain silent, by taking the maximum of the current value of `nav`, and `now+d`, the point in time until the sender `src` of the `rts` message requires the medium. The same behaviour occurs if a `cts` message is received that is not intended for the node itself (Line 4). If the incoming message is an `rts` message intended for the node itself (Line 6) by default the node answers with a clear-to-send message back to the sender (Line 9). However, when the receiver of the `rts` has knowledge about other nodes requiring the medium (`now` $\leq$ `nav`), a clear-to-send cannot be granted, and the request is dropped (Line 6). Similar to the sending of an acknowledgement (Line 2), the process waits for the short interframe space (`sifs`) before sending the CTS (Line 6). Line 8 handles the case where the medium becomes busy ($\neg$IDLE) during this period; also here a clear-to-send cannot be granted, and the request is dropped.[8] Only when the medium stays idle during the entire interframe space the node `id` can inform the source of the `rts` message that the medium is clear to send; the `cts` is transmitted in Line 9. The time a receiver of this message has to be silent is adjusted by deducting the time elapsed before this happens. In Line 10 the process resets `nav` to remind itself not to issue any `rts` message until the present exchange has been completed.[9]

[7] After a successful RTS/CTS exchange, communicating nodes proceed with transmitting the data and an acknowledgement regardless of the value of `nav`.

[8] The condition `now` $>$ `timeout`$-$`sifs` prevents the process from dropping the request in the very first time slice that `CSMA` is running. Here the medium counts as busy, but only because we have just received an `rts` message.

[9] A case NEW(`cts(src,dest,d)`) $\wedge$ `dest` $=$ `id` is not required as a `cts` message is only expected in case an `rts` was sent, and hence handled in process `RTSREACT`.

Process 5. The Basic Routine (RTS/CTS)

```
CSMA(id,nav) =def
 1. newpkt(data,dest). INIT(id,dest,0,dataframe(data,id,dest),nav)
 2. + [NEW(dataframe(data,src,id))] deliver(data) . ⟦timeout := now + sifs⟧
 3.    [now ≥ timeout] transmit(ackframe(src)) . CSMA(id,nav)
 4. + [(NEW(rts(src,dest,d)) ∨ NEW(cts(src,dest,d))) ∧ dest ≠ id ∧ nav < now+d]
 5.    ⟦nav := now+d⟧ CSMA(id,nav)
 6. + [NEW(rts(src,id,d)) ∧ now > nav] ⟦timeout := now + sifs⟧
 7.    (
 8.       [¬IDLE ∧ now > timeout−sifs] CSMA(id,nav)
 9.       + [IDLE ∧ now ≥ timeout] transmit(cts(id,src,d−dur_cts−sifs)) .
10.          ⟦nav := now+d−dur_cts−sifs⟧ CSMA(id,nav)
11.    )
```

Process 6. Clear Channel Assessment With Virtual Carrier Sense

```
CCA(id,dest,b,tries,dframe,nav) =def
 1. [NEW(dataframe(data,src,id))] deliver(data) . ⟦timeout := now + sifs⟧
 2.    [now ≥ timeout] transmit(ackframe(src)) . CCA(id,dest,b,tries,dframe,nav)
 3. + [(NEW(rts(src,dest,d)) ∨ NEW(cts(src,dest,d))) ∧ dest ≠ id ∧ nav < now+d]
 4.    ⟦nav := now+d⟧ CCA(id,dest,b,tries,dframe,nav)
 5. + [NEW(rts(src,id,d)) ∧ now > nav] ⟦timeout := now + sifs⟧
 6.    (
 7.       [¬IDLE ∧ now > timeout−sifs] CCA(id,dest,b,tries,dframe,nav)
 8.       + [IDLE ∧ now ≥ timeout] transmit(cts(id,src,d−dur_cts−sifs)) .
 9.          ⟦nav := now+d−dur_cts−sifs⟧ CCA(id,dest,b,tries,dframe,nav)
10.    )
11. + [IDLE ∧ now > nav]
12.    ⟦timeout:=now+difs⟧
13.    (
14.       [¬IDLE] CCA(id,dest,b,tries,dframe,nav)
15.       + [IDLE ∧ now ≥ timeout]
16.          ⟦timeout := now + b⟧
17.          (
18.             [¬IDLE]      /* busy during backoff time */
19.                ⟦b := timeout − now⟧ CCA(id,dest,b,tries,dframe,nav)
20.             + [IDLE ∧ now ≥ timeout]      /* idle for backoff time */
21.                ⟦d := sifs + dur_cts + sifs + dur(dframe) + sifs + dur_ack⟧
22.                transmit(rts(id,dest,d)) .
23.                CTSRECV(id,dest,tries,now + max_cts_wait,dframe,nav)
24.          )
25.    )
```

Process 6 is the modified version of Process 3. The goal of this process is to send an `rts` message (Line 22). Before it can start its work, it waits until the medium is idle, and any time it is required to be silent has elapsed (Line 11).

Until this happens incoming data frames, `rts` or `cts` messages are treated just as in Process 5: Lines 1–10 copy Lines 2–11 of Process 5, except that afterwards the process returns to itself. Then Lines 12–20 are copied from Lines 7–15 from Process 3. Line 21 calculates the time other nodes ought to keep silent when receiving the `rts` message, and Line 23 passes control to the process `CTSRECV`, which awaits a `cts` response to the `rts` message transmitted in Line 22. The fourth argument of `CTSRECV` specifies the maximum time that process should wait for such a response; a good value for `max_cts_wait` is `sifs` + `dur_cts`.

Process `CTSRECV` listens for this time to a `cts` message with source `dest` and destination `id`. In case the expected `cts` message arrives in time (Line 1), the node waits for a time `sifs` (Line 2) and then transmits the data frame and proceeds to await an acknowledgement (Line 3). The fourth argument of `ACKRECV` specifies the maximum time the process should wait for such an acknowledgement; a good value for `max_ack_wait` is `sifs` + `dur_ack`. If the `cts` message does not arrive in time (Line 6), the process returns to `INIT` to send another `rts` message, while incrementing the counter `tries` (Line 7). While waiting for the `cts` message, any incoming `rts` or `cts` message destined for another node is treated exactly as in Process 5 (Lines 4–5). Incoming data frames cannot arrive when this process is running, and incoming `rts` messages to `id` are ignored.

Process 7. Receiving a CTS

$\texttt{CTSRECV(id,dest,tries,ctstimeout,dframe,nav)} \stackrel{def}{=}$

1. $[\textsc{new}(\texttt{cts(dest,id,d)})]$
2. $[\![\texttt{timeout} := \texttt{now} + \texttt{sifs}]\!]\ [\texttt{now} \geq \texttt{timeout}]$
3. $\textbf{transmit}(\texttt{dframe})\ .\ \texttt{ACKRECV(id,dest,tries,now} + \texttt{max\_ack\_wait,dframe,nav)}$
4. $+\ [(\textsc{new}(\texttt{rts(src,dest,d)}) \vee \textsc{new}(\texttt{cts(src,dest,d)})) \wedge \texttt{dest} \neq \texttt{id} \wedge \texttt{nav} < \texttt{now+d}]$
5. $[\![\texttt{nav} := \texttt{now+d}]\!]\ \texttt{CTSRECV(id,dest,tries,ctstimeout,dframe,nav)}$
6. $+\ [\texttt{now} \geq \texttt{ctstimeout}]$
7. $\texttt{INIT(id,dest,tries+1,dframe,nav)}$

Process 8. Receiving an ACK

$\texttt{ACKRECV(id,dest,tries,acktimeout,dframe,nav)} \stackrel{def}{=}$

1. $[\textsc{new}(\texttt{ackframe(id)})]$
2. $\textbf{deliver}(\texttt{success})\ .\ \texttt{CSMA(id,nav)}$
3. $+\ [(\textsc{new}(\texttt{rts(src,dest,d)}) \vee \textsc{new}(\texttt{cts(src,dest,d)})) \wedge \texttt{dest} \neq \texttt{id} \wedge \texttt{nav} < \texttt{now} + \texttt{d}]$
4. $[\![\texttt{nav} := \texttt{now+d}]\!]\ \texttt{ACKRECV(id,dest,tries,acktimeout,dframe,nav)}$
5. $+\ [\texttt{now} \geq \texttt{timeout}]$ /* nothing received */
6. $\texttt{INIT(id,dest,tries+1,dframe,nav)}$

Process 8 handles the receipt of an acknowledgement in response to a successful data transmission. If an acknowledgement arrives, it must be from the node to which `id` has transmitted a data frame. In that case (Line 1), the network layer is informed that the sending of `dframe` was successful and the process loops back to Process 5 (Line 2). Line 5 describes the situation where no acknowledgement message arrives and the process times out. Also here CSMA/CA retries

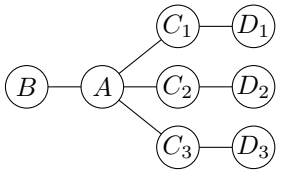
to send the message; the counter `tries` is incremented. Lines 3–4 describe the usual handling of incoming `rts` or `cts` messages destined for another node.

5.4 The Exposed Station Problem

Another source of collisions in CSMA/CA is the well-known *exposed station problem*. This refers to a linear topology $A - B - C - D$, where an unending stream of messages between C and D interferes with attempts by A to get a message across to B. In the default CSMA/CA protocol as formalised in Sect. 5.1, transmissions from A to B may perpetually collide at B with transmissions from C destined for D. CSMA/CA with virtual carrier sensing mitigates this problem, for a `cts` sent by B in response to an `rts` sent by A will tell C to keep silent for the required duration. In fact, we can show that in the above topology, if `max_retransmit`$=\infty$ then packet delivery holds with probability 1. A non-probabilistic guarantee cannot be given since nodes A and C could behave in the same way, meaning if one node is sending out a message the other does the same at the very same moment, and if one is silent the other remains silent as well. In this scenario all messages to be sent are doomed.

Based on our formalisation, we can prove that once the RTS/CTS handshake has been successfully concluded, meaning that all nodes within range of the intended recipient have received the `cts`, then packet delivery holds outright. So the only problem left is to achieve a successful RTS/CTS handshake. Since `rts` and `cts` messages are rather short, even by modest values of `max_retransmit` it becomes likely that such messages do not collide.

In spite of this, CSMA/CA with (or without) virtual channel sensing cannot achieve packet delivery with probability 1 for general topologies. Assume the following network topology



Here it may happen that one of the C_is is always busy transmitting a large message to D_i; any given C_i is occasionally silent (not sending any message), but then one of the others is transmitting. As C_i is disconnected from C_j, for $j \neq i$, coordination between the nodes is impossible. As a consequence, the medium at A will always be busy, so that A cannot send an `rts` message from B.

6 Related Work

The CSMA protocol in its different variants has been analysed with different formalisms in the past.

Multiple analyses were performed for the CSMA/CD protocol (CSMA with collision detection), a predecessor of CSMA/CA that has a constant backoff, i.e.

the backoff time is not increased exponentially, see [10,11,20,21,26]. In all these approaches frame collisions have to be modelled explicitly, as part of the protocol description. In contrast, our approach handles collisions in the semantics; thereby achieving a clear separation between protocol specifications and link layer behaviour.

Duflot et al. [10,11] use probabilistic timed automata (PTAs) to model the protocol, and use probabilistic model checking (PRISM) and approximate model checking (APMC) for their analysis. The model explained in [26] is based on PTAs as well, but uses the model checker Uppaal as verification tool. These approaches, although formal, have very little in common with our approach. On the one hand it is not easy to change the model from CSMA/CD to CSMA/CA, as the latter requires unbounded data structures (or alike) to model the exponential backoff. On the other hand, as usual, model checking suffers from state space explosion and only small networks (usually fewer than ten nodes) can be analysed. This is sufficient and convenient when it comes to finding counter examples, but these approaches cannot provide guarantees for arbitrary network topologies, as ours does.

Jensen et al. [20] use models of CSMA/CD to compare the tools SPIN and Uppaal. Their models are much more abstract than ours. It is proven that no collisions will ever occur, without stating the exact conditions under which this statement holds.

To the best of our knowledge, Parrow [21] is the only one who used process algebra (CCS) to model and analyse CSMA. His untimed model of CSMA/CD is extremely abstract and the analysis performed is limited to two nodes only, avoiding scenarios such as the hidden station problem.

There are far fewer formal analyses techniques available when it comes to CSMA/CA (with and without virtual medium sensing). Traditional approaches to the analysis of network protocols are simulation and test-bed experiments. This is also the case for CSMA/CA (e.g. [4]). While these are important and valid methods for protocol evaluation, in particular for quantitative performance evaluation, they have limitations in regards to the evaluation of basic protocol correctness properties.

Following the spirit of the above-mentioned research of model checking CSMA, Fruth [15] analyses CSMA/CA using PTAs and PRISM. He considers properties such as the minimum probability of two nodes successfully completing their transmissions, and maximum expected number of collisions until two nodes have successfully completed their transmissions. As before, this analysis technique does not scale; in [15] the experiments are limited to two contending nodes only.

Beyond model checking, simulation and test-bed experiments, we are only aware of two other formal approaches. In [1] Markov chains are used to derive an accurate, analytical model to compute the throughput of CSMA/CA. Calculating throughput is an orthogonal task to our vision of proving (functional) correctness.

An approach aiming at proving the correctness of CSMA/CA with virtual carrier sensing (RTS/CTS), and hence related to ours, is presented in [3]. Based

on stochastic bigraphs with sharing it uses rewrite rules to analyse quantitative properties. Although it is an approach that is capable to analyse arbitrary topologies, to apply the rewrite rules a particular topology needs to be modelled by a directed acyclic graph structure, which is part of the bigraph.

7 Conclusion

In this paper we have proposed a novel process algebra, called ALL, that can be used to model, verify and analyse link layer protocols. Since we aimed at a process algebra featuring aspects of the link layer such as frame collisions, as well as arbitrary data structures (to model a rich class of protocols), we could not use any of the existing algebras. The design of ALL is layered. The first layer allows modelling protocols in some sort of pseudo code, which hopefully makes our approach accessible for network and software researchers/engineers. The other layers are mainly for giving a formal semantics to the language. The layer of partial network expressions, the third layer, provides a unique and sophisticated mechanism for modelling the collision of frames. As it is hard-wired in the semantics there is no need to model collisions manually when modelling a protocol, as it was done before [21]. Next to primitives needed for modelling link layer protocols (e.g. **transmit**) and standard operators of process algebra (e.g. nondeterministic choice), ALL provides an operator for probabilistic choice.

This operator is needed to model aspects of link layer protocols such as the exponential backoff for the Carrier-Sense Multiple Access with Collision Avoidance protocol, the case study we have chosen to demonstrate the applicability of ALL. We have modelled and analysed two versions of CSMA/CA, without and with virtual carrier sensing. Our analysis has confirmed the hidden station problem for the version without virtual carrier sensing. However, we have also shown that the version with virtual carrier sensing overcomes not only this problem, but also the exposed station problem with probability 1. Yet the protocol cannot guarantee packet delivery, not even with probability 1.

To perform this analysis we had to formalise suitable liveness properties for link layer protocols specified in our framework.

Acknowledgement. We thank Tran Ngoc Ma for her involvement in this project in a very early phase. We also like to thank the German Academic Exchange Service (DAAD) that funded an internship of the third author at Data61, CSIRO.

References

1. Bianchi, G.: Performance analysis of the IEEE 802.11 distributed coordination function. IEEE J. Sel. Areas Commun. **18**(3), 535–547 (2000). https://doi.org/10.1109/49.840210
2. Bres, E., van Glabbeek, R.J., Höfner, P.: A timed process algebra for wireless networks with an application in routing. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 95–122. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_5

3. Calder, M., Sevegnani, M.: Modelling IEEE 802.11 CSMA/CA RTS/CTS with stochastic bigraphs with sharing. Formal Aspects Comput. **26**(3), 537–561 (2014). https://doi.org/10.1007/s00165-012-0270-3
4. Chhaya, H.S., Gupta, S.: Performance modeling of asynchronous data transfer methods of IEEE 802.11 MAC Protocol. Wirel. Netw. **3**, 217–234 (1997). https://doi.org/10.1023/A:1019109301754
5. Comer, D.: Computer Networks and Internets. Pearson Education Inc., UpperSaddle River (2009)
6. Cranen, S., Mousavi, M.R., Reniers, M.A.: A rule format for associativity. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 447–461. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_35
7. De Nicola, R., Vaandrager, F.W.: Three logics for branching bisimulation. J. ACM **42**(2), 458–487 (1995). https://doi.org/10.1145/201019.201032
8. Deng, Y., van Glabbeek, R.J., Hennessy, M., Morgan, C.C., Zhang, C.: Remarks on testing probabilistic processes. In: Cardelli, L., Fiore, M., Winskel, G. (eds.) Computation, Meaning, and Logic: Articles Dedicated to Gordon Plotkin, Electronic Notes in Theoretical Computer Science, vol. 172, pp. 359–397. Elsevier (2007). https://doi.org/10.1016/j.entcs.2007.02.013
9. Deng, Y., van Glabbeek, R.J., Morgan, C.C., Zhang, C.: Scalar outcomes suffice for finitary probabilistic testing. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 363–378. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_25
10. Duflot, M., et al.: Probabilistic model checking of the CSMA/CD, protocol using PRISM and APMC. In: Automated Verification of Critical Systems (AVoCS 2004). Electronic Notes in Theoretical Computer Science Series, vol. 128, pp. 195–214 (2004). https://doi.org/10.1016/j.entcs.2005.04.012
11. Duflot, M., et al.: Practical applications of probabilistic model checking to communication protocols. In: Gnesi, S., Margaria, T. (eds.) Formal Methods for Industrial Critical Systems: A Survey of Applications, pp. 133–150. IEEE (2013). https://doi.org/10.1002/9781118459898.ch7
12. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A.K., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 295–315. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_15
13. Fehnker, A., van Glabbeek, R.J., Höfner, P., McIver, A.K., Portmann, M., Tan, W.L.: A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. Technical report 5513, NICTA (2013). http://arxiv.org/abs/1312.7645
14. Friend, G.E., Fike, J.L., Baker, H.C., Bellamy, J.C.: Understanding Data Communications, 2nd edn. Howard W. Sams & Company, Indianapolis (1988)
15. Fruth, M.: Probabilistic model checking of contention resolution in the IEEE 802.15.4 low-rate wireless personal area network protocol. In: Leveraging Applications of Formal Methods, Second International Symposium (ISoLA 2006), pp. 290–297. IEEE Computer Society (2006). https://doi.org/10.1109/ISoLA.2006.34
16. IEEE: IEEE standard for ethernet (2016). https://doi.org/10.1109/IEEESTD.2016.7428776
17. IEEE: IEEE standard for low-rate wireless networks (2016). https://doi.org/10.1109/IEEESTD.2016.7460875
18. ISO/IEC 7498–1: Information technology—open systems interconnection—basic reference model: The basic model (1994). https://www.iso.org/standard/20269.html

19. ISO/IEC/IEEE 8802–11: Information technology—telecommunications and information exchange between systems—local and metropolitan area networks—specific requirements—part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications (2018). https://www.iso.org/standard/73367.html
20. Jensen, H.E., Larsen, K.G., Skou, A.: Modelling and analysis of a collision avoidance protocol using Spin and Uppaal. In: The Spin Verification System. Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 33–50. DIMACS/AMS (1996). https://doi.org/10.7146/brics.v3i24.20005
21. Parrow, J.: Verifying a CSMA/CD-protocol with CCS. In: Aggarwal, S. (eds.) IFIP Symposium on Protocol Specification, Testing and Verification (PSTV 1988), North-Holland, pp. 373–384 (1988)
22. Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science (FOCS 1977), pp. 46–57. IEEE (1977). https://doi.org/10.1109/SFCS.1977.32
23. de Simone, R.: Higher-level synchronising devices in Meije-SCCS. TCS **37**, 245–267 (1985). https://doi.org/10.1016/0304-3975(85)90093-3
24. Simpson, W.: The point-to-point protocol (PPP). RFC 1661 Internet Standard (1994). http://www.ietf.org/rfc/rfc1661.txt
25. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Sci. Comput. Program. **75**, 440–469 (2010). https://doi.org/10.1016/j.scico.2009.07.008
26. Zhao, J., Li, X., Zheng, T., Zheng, G.: Removing irrelevant atomic formulas for checking timed automata efficiently. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 34–45. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-40903-8_4

Program Analysis and Automated Verification

Data Races and Static Analysis for Interrupt-Driven Kernels

Nikita Chopra, Rekha Pai(✉), and Deepak D'Souza

Indian Institute of Science, Bangalore, India
{nikita,rekhapai,deepakd}@iisc.ac.in

Abstract. We consider a class of interrupt-driven programs that model the kernel API libraries of some popular real-time embedded operating systems and the synchronization mechanisms they use. We define a natural notion of data races and a happens-before ordering for such programs. The key insight is the notion of *disjoint blocks* to define the synchronizes-with relation. This notion also suggests an efficient and effective lockset based analysis for race detection. It also enables us to define efficient "sync-CFG" based static analyses for such programs, which exploit data race freedom. We use this theory to carry out static analysis on the FreeRTOS kernel library to detect races and to infer simple relational invariants on key kernel variables and data-structures.

Keywords: Static analysis · Interrupt-driven programs · Data races

1 Introduction

Embedded software is widespread and increasingly employed in safety-critical applications in medical, automobile, and aerospace domains. These programs are typically multi-threaded applications, running on uni-processor systems, that are compiled along with a kernel library that provides priority-based scheduling, and other task management and communication functionality. The applications themselves are similar to classical multi-threaded programs (using lock, semaphore, or queue based synchronization) although they are distinguished by their priority-based execution semantics. The kernel on the other hand typically makes use of non-standard low-level synchronization mechanisms (like disabling-enabling interrupts, suspending the scheduler, and flag-based synchronization) to ensure thread-safe access to its data-structures. In the literature such software (both applications and kernels) are referred to as *interrupt-driven* programs. Our interest in this paper is in the subclass of interrupt-driven programs corresponding to kernel libraries.

Efficient static analysis of concurrent programs is a challenging problem. One could carry out a precise analysis by considering the *product* of the control flow graphs (CFGs) of the threads, however this is prohibitively expensive due to the exponential number of program points in the product graph. A promising direction is to focus on the subclass of *race-free* programs. This is an important class

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 697–723, 2019.
https://doi.org/10.1007/978-3-030-17184-1_25

of programs, as most developers aim to write race-free code, and one could try to exploit this property to give an efficient way of analyzing programs that fall in this class. In recent years there have been many techniques [7,11,12,18,21] that exploit the race-freedom property to perform sound and efficient static analysis. In particular [11,21] create an appealing structure called a "sync-CFG" which is the *union* of the control flow graphs of the threads augmented with possible "synchronization" edges, and essentially perform sequential analysis on this graph to obtain sound facts about the concurrent program. However these techniques are all for classical lock-based concurrent programs. A natural question asks if we can analyze interrupt-driven programs in a similar way.

There are several challenges in doing this. Firstly one needs to define *what* constitutes a data race in a generalized setting that includes these programs. Secondly, how does one define the happens-before order, and in particular the *synchronizes-with* relation that many of the race-free analysis techniques rely on, given the ad-hoc synchronization mechanisms used in these programs.

A natural route that suggests itself is to translate a given interrupt-driven program into one that uses classical locks, and faithfully captures the interleaved executions of the original program. One could then use existing techniques for lock-based concurrency to analyze these programs. However, this route is fraught with many challenges. To begin with, it is not clear how one would handle flag-based synchronization which is one of the main synchronization mechanisms used in these programs. Even if one could handle this, such a translation *may not* preserve data races, in that the original program might have had a race but the translated program does not. Finally, some of the synchronizes-with edges in the translated program are clearly unnecessary, leading to imprecise data-flow facts in the analyses.

In this paper, we show that it is possible to take a more organic route and address these challenges in a principled way that could apply to other non-standard classes of concurrent systems as well. Firstly, we propose a general definition of a data race that is not based on a happens-before order, but on the operational semantics of the class of programs under consideration. The definition essentially says that two statements s and t can race, if two notional "blocks" around them can *overlap* in time during an execution. We believe that this definition accurately captures what it is that a programmer tries to avoid while dealing with shared variables whose values matter. Secondly we propose a way of defining the *synchronizes-with* relation, based on the notion of *disjoint blocks*. These are statically identifiable pairs of path segments in the CFGs of different threads that are guaranteed to never overlap (in time) during an execution of the program, much like blocks of code that lie between an acquire and release of the same lock. This relation now suggests a natural sync-CFG structure on which we can perform analyses like value-set (including interval, null-deference, and points-to analysis), and region-based relational invariant analysis, in a sound and efficient manner. We also use the notion of disjoint blocks to define an efficient and precise lock-set-based analysis for detecting races in interrupt-driven programs.

We implement some of these analyses on the FreeRTOS kernel library [3] which is one of the most widely used open-source real-time kernels for embedded systems, comprising about 3,500 lines of C code. Our race-detection analysis reports a total of 64 races in kernel methods, of which 18 turn out to be true positives. We also carry out a region-based relational analysis using an implementation based on CIL [22]/Apron [15], to prove several relational invariants on the kernel variables and abstracted data-structures.

2 Overview

We give an overview of our contributions via an illustrative example modelled on a portion of the FreeRTOS kernel library. Figure 1 shows an interrupt-driven program that contains a main thread that first initializes the kernel variables. The variables represent components of a message queue, like `msgw` (the number of messages waiting in the queue), `len` (max length of the queue), `wtosend` (the number of tasks waiting to send to the queue), `wtorec` (the number of tasks waiting to receive from the queue), and `RxLock` (a counter which also acts as a synchronization flag that mediates access to the waiting queues). The main thread then creates (or spawns) two threads: *qsend* which models the kernel API method for sending a message to the queue, and *qrec_ISR* which models a method for receiving a message, and which is meant to be called from an interrupt-service routine. The basic semantics of this program is that the ISR thread can interrupt *qsend* at any time (provided interrupts are not disabled), but always runs to completion itself. The threads use `disableint`/`enableint` to disable and enable interrupts, `suspendsch`/`resumesch` to suspend/resume the scheduler (thereby preventing preemption by another non-ISR thread), and finally flag-based synchronization (using the `RxLock` variable), as different means to ensure mutual exclusion.

Our first contribution is a general notion of data races which is applicable to such programs. We say that two conflicting statements s and t in two different threads are involved in a data race if assuming s and t were enclosed in a notional "block" of skip statements, there is an execution in which the two blocks "overlap" in time. The given program can be seen to be free of races. However if we were to remove the `disableint` statement of line 10, then the statements accessing `msgw` in lines 12 and 42 would be racy, since soon after the access of `msgw` in *qsend* at line 12, there could be preemption by *qrec_ISR* which goes on to execute line 42.

Next we illustrate the notion of "disjoint blocks" which is the key to defining synchronizes-with edges, which we need in our sync-CFG analysis as well as to define an appropriate happens-before relation. Disjoint blocks are also used in our race-detection algorithm. A pair of blocks of code (for example any of the like-shaded blocks of code in the figure) are *disjoint* if they can never overlap during an execution. For example, the block comprising lines 11–14 in *qsend* and the whole of *qrec_ISR*, form a pair of disjoint blocks.

Next we give an analysis for checking race-freedom, by adapting the standard lockset analysis [24] for classical concurrent programs. We associate a unique

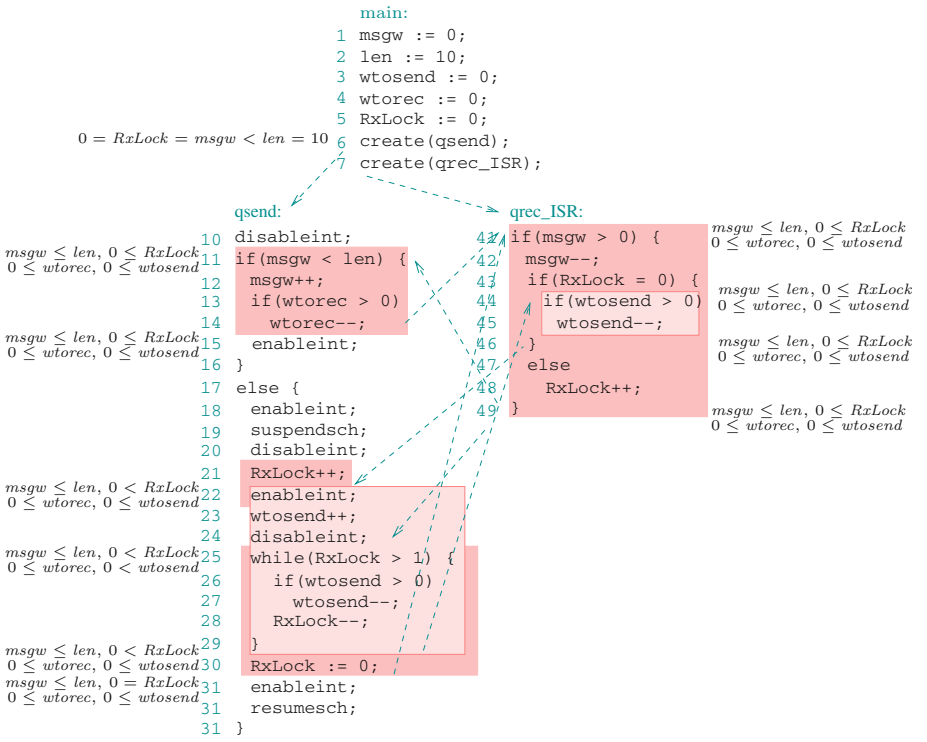


Fig. 1. An interrupt-driven program modelled on the FreeRTOS kernel library. Similarly shaded blocks denote disjoint blocks. Some of the sync-with edges are shown in dashed lines. Some edges like 22 → 41 and 49 → 20 have been omitted for clarity.

lock with each pair of disjoint blocks, and add notional acquires and releases of this lock at the beginning and end (respectively) of these blocks. We now do the standard lockset analysis on this version of the program, and declare two accesses to be non-racy if they hold sets of locks with a non-empty intersection.

Finally, we show how to do data-flow analysis for such programs in a sound and efficient way. The basic idea is to construct a "sync-CFG" for the program by unioning the control-flow graphs of the threads, and adding *sync* edges that capture the synchronizes-with edges (going from the end of a block to the beginning of its paired block), for example line 14 to line 41 and line 49 to line 11. The sync-edges are shown by dashed arrows in the figure. We now do a standard "value-set" analysis (for example interval analysis) on this graph, keeping track of a set of values each variable can take. The resulting facts about a variable are guaranteed to be sound at points where the variable is accessed (or even "owned" in the sense that a notional read of the variable at that point is non-racy). For example an interval analysis on this program would give us that $0 <$ `msgw` at line 14. Finally, we could do a region-based value-set analysis, by identifying regions of variables that are accessed as a unit – for example `msgw` and `len` could

be in one region, while `wtosend` and `wtorec` could be in another. The figure shows some facts inferred by a polyhedral analysis based on these regions, for the given program.

3 Interrupt-Driven Programs

The programs we consider have a finite number of (static) threads, with a designated "main" thread in which execution begins. The threads access a set of shared global variables, some of which are used as "synchronization flags", using a standard set of commands like assignment statements of the form `x := e`, conditional statements (`if-then-else`), loop statements (`while`), etc. In addition, the threads can use commands like `disableint`, `enableint` (to disable and enable interrupts, respectively), `suspendsch`, `resumesch` (to suspend and resume the scheduler, respectively), while the main thread can also `create` a thread (enable it for execution). Table 1 shows the set of basic statements $cmd_{V,T}$ over a set of variables V and a set of threads T.

We allow standard integer and Boolean expressions over a set of variables V. For an integer expression e over V, and an environment ϕ for V, we denote by $[\![e]\!]_\phi$ the integer value that e evaluates to in ϕ. Similarly for a Boolean expression b, we denote the Boolean value (*true* or *false*) that b evaluates to in ϕ by $[\![b]\!]_\phi$. For a set of environments Φ for a set of variables V, we define the set of integer values that e can evaluate to in an environment in Φ, by $[\![e]\!]_\Phi = \{[\![e]\!]_\phi \mid \phi \in \Phi\}$. Similarly, for a boolean expression b, we define the set of environments in Φ that satisfy b to be $[\![b]\!]_\Phi = \{\phi \in \Phi \mid [\![b]\!]_\phi = \mathit{true}\}$.

Each thread is of one of two *types*: "task" threads that are like standard threads, and "ISR" threads that represent threads that run as interrupt service routines. The *main* thread is a task thread, which is the only task thread enabled initially. The *main* thread can enable other threads (both task and ISR) for execution using the `create` command. Task threads can be preempted by other task threads (whenever interrupts are not disabled, and the scheduler is not suspended) or by ISR threads (whenever interrupts are not disabled). On the other hand ISR threads cannot be preempted and are assumed to run to completion.

Only task threads are allowed to use `disableint`, `enableint`, `suspendsch` and `resumesch` commands. Similarly, if flag-based synchronization is used, only task threads can modify the flag variable, while an ISR can only check whether the flag is set or not, and perform some actions accordingly.

Formally we represent an interrupt-driven program P as a tuple (V, T) where V is a finite set of integer variables, and T is a finite set of named threads. Each thread $t \in T$ has a *type* which is one of *task* or *ISR*, and an associated control-flow graph of the form $G_t = (L_t, s_t, inst_t)$ where L_t is a finite set of *locations* of thread t, $s_t \in L_t$ is the *start* location of thread t, $inst_t \subseteq L_t \times cmd_{V,T} \times L_t$ is a finite set of *instructions* of thread t.

Some definitions related to threads will be useful going forward. We denote by $L_P = \bigcup_{t \in T} L_t$ the disjoint union of the thread locations. Whenever P is clear

Table 1. Basic statements $cmd_{V,T}$ over variables V and threads T

Command	Description
`skip`	Do nothing
`x := e`	Assign the value of expression e to variable $x \in V$
`assume`(b)	Enabled only if expression b evaluates to *true*, acts like `skip`
`create(t)`	Enable thread $t \in T$ for execution
`disableint`	Disable interrupts and context switches
`enableint`	Enable interrupts and context switches
`suspendsch`	Suspend the scheduler (other task threads cannot preempt the current thread); Also sets `ssflag` variable
`resumesch`	Resume the scheduler (other task threads can now preempt the current thread); Also unsets `ssflag` variable

from the context we will drop the subscript of P from L_P and its decorations. For a location $l \in L$ we denote by $tid(l)$ the thread t which contains location l. We denote the set of instructions of P by $inst_P = \bigcup_{t \in T} inst_t$. For an instruction $\iota \in inst_t$, we will also write $tid(\iota)$ to mean the thread t. For an instruction $\iota = \langle l, c, l' \rangle$, we call l the *source* location, and l' the *target* location of ι.

We denote the set of commands appearing in program P by $cmd(P)$. We will consider an assignment `x := e` as a *write-access* to x, and as a *read-access* to every variable that appears in the expression e. Similarly, `assume`(b) is considered to be a read-access of every variable that occurs in expression b. We say two accesses are *conflicting* accesses if they are read/write accesses to the same variable, and at least one of them is a write. We assume that the control-flow graph of each thread comes from a well-structured program. Finally, we assume that the *main* thread begins by initializing the variables to constant values. Figure 2 shows an example program and the control-flow-graphs of its threads.

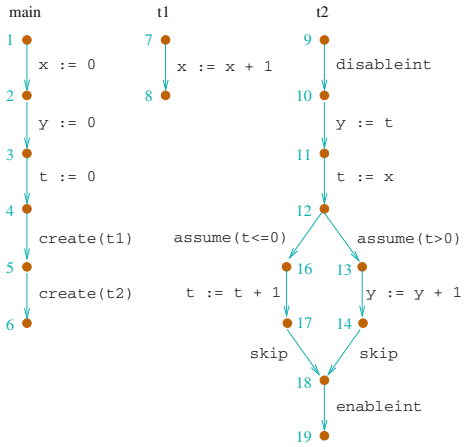
We define the operational semantics of an interrupt-driven program using a labeled transition system (LTS). Let $P = (V, T)$ be a program. We define an LTS $\mathcal{T}_P = (Q, \Sigma, s, \Rightarrow)$ corresponding to P, where:

- Q is a set of states of the form $(pc, \phi, enab, rt, it, id, ss)$, where $pc \in T \to L$ is the program counter giving the current location of each thread, $\phi \in V \to \mathbb{Z}$ is a valuation for the variables, $enab \subseteq T$ is the set of enabled threads, $rt \in T$ is the currently running thread; $it \in T$ is the task thread which is interrupted when the scheduler is suspended; and id and ss are Boolean values telling us whether interrupts are disabled ($id = true$) or not ($id = false$) and whether the scheduler is suspended ($ss = true$) or not ($ss = false$).
- The set of labels Σ is the set of instructions $inst_P$ of P.
- The initial state s is $(\lambda t.s_t, \lambda x.0, \{main\}, main, main, false, false)$. Thus all threads are at their entry locations, the initial environment sets all variables to 0, only the main thread is enabled and running, the interrupted task is

```
main:
1. x := 0;
2. y := 0;
3. t := 0;
4. create(t1);
5. create(t2);
6.

t1:                t2:
7. x := x + 1;     9. disableint;
8.                 10. y := t;
                   11. t := x;
                   12. if(t > 0) {
                   13.   y := y + 1;
                   14. }
                   15. else {
                   16.   t := t + 1;
                   17. }
                   18. enableint;
                   19.
```

(a) Example program



(b) Control-flow-graph representation

Fig. 2. An example program and its CFG representation.

set to *main* (this is a dummy value as it is used only when the scheduler is suspended), interrupts are enabled, and the scheduler is not suspended.

– For an instruction $\iota = \langle l, c, l' \rangle$ in $inst_P$, with $tid(\iota) = t$, we define

$$(pc, \phi, enab, rt, it, id, ss) \Rightarrow_\iota (pc', \phi', enab', rt', it', id', ss')$$

iff the following conditions are satisfied:

- $t \in enab$; $pc(t) = l$; $pc' = pc[t \mapsto l']$;
- if id is true or rt is an ISR then $t = rt$;
- if ss is true, then either $t = rt$ or t is an ISR thread;
- Based on the command c, the following conditions must be satisfied:
  - If c is the `skip` command then $\phi' = \phi$, $enab' = enab$, $id' = id$, and $ss' = ss$.
  - If c is an assignment statement of the form $x := e$ then $\phi' = \phi[x \mapsto [\![e]\!]_\phi]$, $enab' = enab$, $id' = id$, and $ss' = ss$.
  - If c is a command of the form `assume`(b) then $[\![b]\!]_\phi = true$, $\phi' = \phi$, $enab' = enab$, $id' = id$, and $ss' = ss$.
  - If c is a `create`(u) command then $t = main$, $\phi' = \phi$, $enab' = enab \cup \{u\}$, $id' = id$, and $ss' = ss$.
  - If c is the `disableint` command then $\phi' = \phi$, $enab' = enab$, $id' = true$, and $ss' = ss$.
  - If c is the `enableint` command then $\phi' = \phi$, $enab' = enab$, $id' = false$, and $ss' = ss$.
  - If c is the `suspendsch` command then $\phi' = \phi[ssflag \mapsto 1]$, $enab' = enab$, $id' = id$, and $ss' = true$.
  - If c is the `resumesch` command then $\phi' = \phi[ssflag \mapsto 0]$, $enab' = enab$, $id' = id$, and $ss' = false$.

- In addition, the transitions set the new running thread rt' and interrupted task it' as follows. If t is an ISR thread, ss is true, and ι is the first statement of t then $it' = rt$, $rt' = t$. If t is an ISR thread, ss is true, and ι is the last statement of t then $it' = it$, $rt' = it$. In all other cases, $rt' = t$ and $it' = it$.

An execution σ of P is a finite sequence of transitions in $\mathcal{T}_P$ from the initial state s: $\sigma = \tau_0, \tau_1, \ldots, \tau_n$ $(n \geq 0)$ from $\Rightarrow$, such that there exists a sequence of states $q_0, q_1, \ldots, q_{n+1}$ from Q, with $q_0 = s$ and $\tau_i = (q_i, \iota_i, q_{i+1})$ for each $0 \leq i \leq n$. Wherever convenient we will also represent an execution like σ above as a sequence of the form $q_0 \Rightarrow_{\iota_0} q_1 \Rightarrow_{\iota_1} \cdots \Rightarrow_{\iota_n} q_{n+1}$. We say that a state $q \in Q$ is *reachable* in program P if there is an execution of P leading to state q.

4 Data Races and Happens-Before Ordering

In this section we propose a definition of a data race which has general applicability, and also define a natural happens-before order for interrupt-driven programs.

4.1 Data Races

Data races have typically been defined in the literature in terms of a *happens-before* order on program executions. In the classical setting of lock-based synchronization, the happens-before relation is a partial order on the instructions in an execution, that is reflexive-transitive closure of the union of the *program-order* relation between two instructions in the same thread, and the *synchronizes-with* relation which relates a release of a lock in a thread to the next acquire of the same lock in another thread. Two instructions in an execution are then defined to be involved in a data race if they are conflicting accesses to a shared variable and are *not* ordered by the happens-before relation.

We feel it is important to have a definition of a data race that is based on the operational semantics of the class of programs we are interested in, and not on a happens-before relation. Such a definition would more tangibly capture what it is that a programmer typically tries to avoid when dealing with shared variables whose consistency she is worried about. Moreover, when coming up with a definition of the happens-before order (the synchronizes-with relation in particular) for non-standard concurrent programs like interrupt-driven programs, it is useful to have a reference notion to relate to. For instance, one could show that a proposed happens-before order is strong enough to ensure the absence of races.

We propose to define a race between two conflicting statements in a program in terms of whether two imaginary blocks enclosing each of these statements can *overlap* in an execution. Let us consider a multi-threaded program P in a class of concurrent programs with a certain operational execution semantics. Consider a block of contiguous instructions in a thread t of a program P and another block in thread t' of P. We say that these two blocks are involved in a *high-level race* in an execution of P if they *overlap* with each other during the execution, in that

one block begins *in between* the beginning and ending of the other. We say two conflicting statements s and t in P are involved in a *data race* (or are *racy*), if the following condition is true: Consider the program P' which is obtained from P by replacing the statement s by the block "`skip`; s; `skip`", and similarly for statement t. Then there is an execution of P' in which the two blocks containing s and t are involved in a high-level race. The definition is illustrated in Fig. 3. We say a program P is *race-free* if no pair of instructions in it are racy.

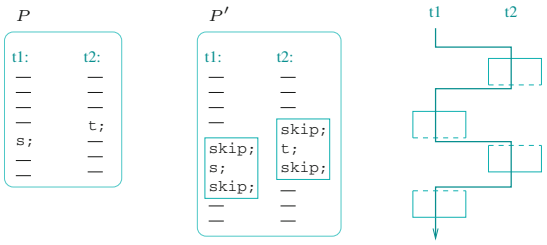


Fig. 3. Illustrating the definition of a data race on statements s and t. A program P, its transformation P', and an execution of P' in which the blocks overlap.

The rationale for this definition is that the concerned statements s and t may be compiled down to a sequence of instructions (represented by the blocks with `skip`'s around s and t) depending on the underlying processor and compiler, and if these instructions interleave in an execution, it may lead to undesirable results.

To illustrate the definition, consider the program in Fig. 2a. The accesses to x in line 7 and line 11 can be seen to be racy, since there is an execution of the augmented program P' in which *t1* performs the `skip` followed by the increment to x at line 7, followed by a context switch to thread *t2* which goes on to execute lines 9 and 10 and then the read of x in line 11. On the other hand, the version of the program in which line 7 is enclosed in a `disableint-enableint` block, does *not* contain a race.

We note that for classical concurrent programs, it might suffice to define a race as *consecutive* occurrences of conflicting accesses in an execution, as done in [4,17]. However, this definition is not general enough to apply to interrupt-driven programs. By this definition, the statements in lines 7 and 11 of the program in Fig. 2a are *not* racy, as there is *no* execution in which they happen consecutively. This is because the `disableint-enableint` block containing the access in line 11 is "atomic" in that the statements in the block must happen contiguously in any execution, and hence the instructions corresponding to line 7 and line 11 can never happen immediately one after another.

4.2 Disjoint Blocks and the Happens-Before Relation

Now that we have a proposed definition of races, we can proceed to give a principled way to define the happens-before relation for our class of interrupt-

driven programs. The main question is how does one define the synchronizes-with relation. Our insight here is that the key to defining the synchronizes-with relation lies in identifying what we call *disjoint blocks* for the class of programs. Disjoint blocks are statically identifiable pairs of path segments in the CFGs of different threads, which are guaranteed by the execution semantics of the class of programs never to *overlap* in an execution of the program. Disjoint block structures – for example in the form of blocks enclosed between locks/unlocks of the same lock – are the primary mechanism used by developers to ensure race-freedom. The synchronizes-with relation in an execution can then be defined as relating, for every pair (A, B) of disjoint blocks in the program, the end of block A to the beginning of the succeeding occurrence of block B in the execution. The happens-before order for an execution can now be defined, as before, in terms of the program order and the synchronizes-with order, and is easily seen to be sufficient to ensure non-raciness.

Let us illustrate this hypothesis on classical lock-based programs. The disjoint block pairs for this class of programs are segments of code enclosed between acquires and releases of the *same* lock; or the portion of a thread's code before it spawns a thread t, and the whole of thread t's code; and similarly for joins. The synchronizes-with relation between instructions in an execution essentially goes from a release to the succeeding acquire of the same lock. If two accesses are related by the resulting happens-before order, they clearly cannot be involved in a race.

We now focus on defining a happens-before relation based on disjoint blocks for our class of interrupt-driven programs. We have identified eight pairs of disjoint block patterns for this class of programs, which are depicted in Fig. 4. We use the following types of blocks to define the pairs. A block of type D is a path segment in a task thread that begins with a `disableint` and ends with an `enableint` with no intervening `enableint` in between. A block of type S is a path segment in a task thread that begins with a `suspendsch` and ends with a `resumesch` with no intervening `resumesch`. An I block is an initial and terminating path segment in an ISR thread (i.e. begins with the first instruction and ends with a terminating instruction). Similarly, for a task thread t, T_t is an initial and terminating path in t, while M_t is an initial segment of the main thread that ends with a `create`(t) command. A block of type C_{ssflag} is a path segment in an ISR thread corresponding to the `then` block of a conditional that checks if $ssflag = 0$. For a synchronization flag f, C_f is the path segment in an ISR thread corresponding to the `then` block of a conditional that checks if $f = 0$. Finally F_f is a segment between statements that set f to 1 and back to 0, in a task thread. We also require that an F_f segment be within the scope of a `suspendsch` command.

We can now describe the pairs of disjoint blocks depicted in Fig. 4. Case (a) says that two D blocks in different task threads are disjoint. Clearly two such blocks can never overlap in an execution, since once one of the blocks begins execution no context-switch can occur until interrupts are enabled again. Case (b) says that D and I blocks are disjoint. Once again this is because once the D block

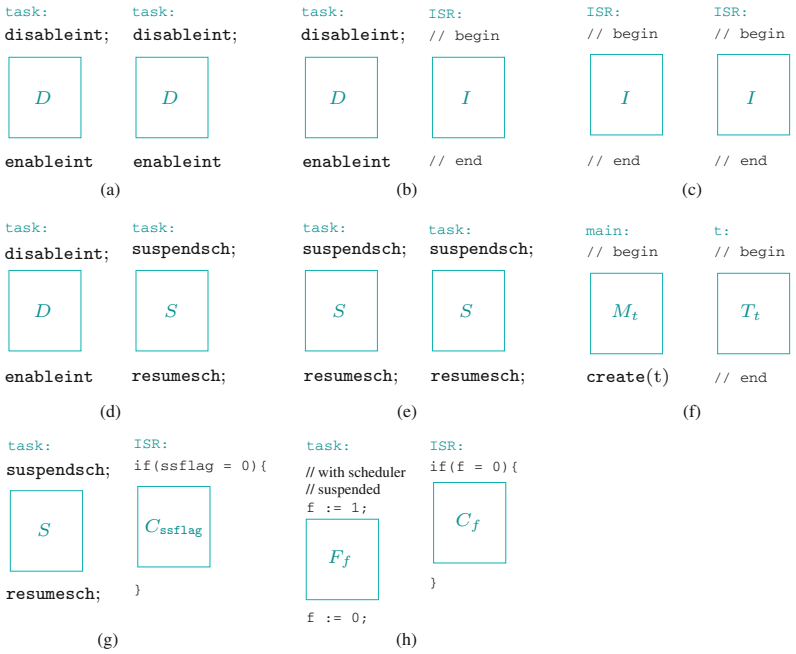


Fig. 4. Disjoint blocks in an interrupt-driven program.

begins execution no ISR can run until interrupts are enabled again, and once an ISR begins execution it runs to completion without any context-switches. Case (e) says that S blocks in different task threads are disjoint, because once the scheduler is suspended no context-switch to another task thread can occur. Case (f) says that M_t and T_t blocks are disjoint, since a thread cannot begin execution before it is created in main. Case (g) says that an S block is disjoint from a $C_{\texttt{ssflag}}$ block. This is because once the scheduler is suspended by the `suspendsch` command, and even if a context-switch to an ISR occurs, the `then` block of the `if` statement will not execute. Conversely, if the ISR is running there can be no context-switch to another thread. Finally, case (h) is similar to case (g). We note that the disjoint block pairs are not ordered (the relation is symmetric).

We can now define the synchronizes-with relation as follows. Let $\sigma = q_0 \Rightarrow_{\iota_0} q_1 \Rightarrow_{\iota_1} \cdots \Rightarrow_{\iota_n} q_{n+1}$ be an execution of P. We say instruction ι_i *synchronizes-with* an instruction ι_j of P in σ, if $i < j$, $tid(\iota_i) \neq tid(\iota_j)$, and there exists a pair of disjoint blocks A and B, with ι_i ending block A and ι_j beginning block B. As usual we say ι_i is *program-order* related to ι_j iff $i < j$ and $tid(\iota_i) = tid(\iota_j)$. We define the *happens-before* relation on σ as the reflexive-transitive closure of the union of the program-order and synchronizes-with relations for σ.

We can now define a *HB-race* in an execution σ of P as follows: we say that two instructions ι_i and ι_j in σ are involved in a *HB-race* if they are conflicting

instructions that are *not* ordered by the happens-before relation in σ. We say that two instructions in P are *HB-racy* if there is an execution of P in which they are involved in a HB-race. Finally, we say a program P is *HB-race-free* if no two of its instructions are HB-racy.

Once again, it is fairly immediate to see that if two statements of a program are not involved in a HB-race, they cannot be involved in a race. Further, if two statements belong to disjoint blocks, then they are clearly happens-before ordered in every execution. Hence belonging to disjoint blocks is sufficient to ensure that the statements are happens-before ordered, which in turn ensures that the statements cannot be involved in a race.

5 Sync-CFG Analysis for Interrupt-Driven Programs

In this section we describe a way of lifting a sequential value-set analysis in a sound way for a HB-race free interrupt-driven program, in a similar way to how it is done for lock-based concurrent programs in [11]. A value-set analysis keeps track of the set of values each variable can take at each program point. The basic idea is to create a "sync-CFG" for a given interrupt-driven program P, which is essentially the union of the CFGs of each thread of P, along with "may-synchronize-with" edges between statements that may be synchronizes-with related in an execution of P, and then perform the value-set analysis on the resulting graph. Whenever the given program is *HB-race free*, the result of the analysis is guaranteed to be sound, in a sense made clear in Theorem 1.

5.1 Sync-CFG

We begin by defining the "sync-CFG" for an interrupt-driven program. It is on this structure that we will do the value-set analysis. Let $P = (V, T)$ be an interrupt-driven program, and let G be the disjoint union (over threads $t \in T$) of the CFGs G_t. We define a set of *may-synchronize-with* edges in G, denoted $MSW(G)$, as follows. The edges correspond to the pairs of disjoint blocks depicted in Fig. 4, in that they connect the ending of one block to the beginning of the other block in the pair. Consider two instructions $\iota = \langle l, c, m\rangle \in inst_t$ and $\kappa = \langle l', c', m'\rangle \in inst_{t'}$, with $t \neq t'$. We add the edge (m, l') in $MSW(G)$, iff for some pair of disjoint blocks (A, B), ι ends a block of type A in thread t and κ begins a block of type B in thread t'. For example, corresponding to a (D, D) pair of disjoint blocks, we add the edge (m, l') when c is an `enableint` command, and c' is a `disableint` command.

The sync-CFG induced by P is the control flow graph given by G along with the additional edges in MSW(G). Figure 6 shows a program P_2 and its induced sync-CFG.

5.2 Value Set Analysis

We first spell out the particular form of abstract interpretation we will be using. It is similar to the standard formulation of [9], except that it is a little more general to accommodate non-standard control-flow graphs like the sync-CFG.

An *abstract interpretation* of a program $P = (V, T)$ is a structure of the form $\mathcal{A} = (D, \leq, d_o, F)$ where

- D is the set of *abstract states*.
- $(D, \leq)$ forms a complete lattice. We denote the join (least upper bound) in this lattice by $\sqcup_{\leq}$, or simply $\sqcup$ when the ordering is clear from the context.
- $d_0 \in D$ is the initial abstract state.
- $F : inst_P \to (D \to D)$ associates a *transfer function* $F(\iota)$ (or simply F_ι) with each instruction ι of P. We require each transfer function F_ι to be *monotonic*, in that whenever $d \leq d'$ we have $F_\iota(d) \leq F_\iota(d')$.

An abstract interpretation $\mathcal{A} = (D, \leq, d_0, F)$ of P induces a "global" transfer function $\mathcal{F}_\mathcal{A} : D \to D$, given by $\mathcal{F}_\mathcal{A}(d) = d_0 \sqcup \bigsqcup_{\iota \in inst_P} F_\iota(d)$. This transfer function can also be seen to be monotonic. By the Knaster-Tarski theorem [28], $\mathcal{F}_\mathcal{A}$ has a least fixed point (LFP) in D, which we denote by $LFP(\mathcal{F}_\mathcal{A})$, and refer to as the resulting value of the analysis.

A *value set* for a set of variables V is a map $vs : V \to 2^{\mathbb{Z}}$, associating a set of integer values with each variable in V. A value set vs induces a set of environments Φ_{vs} in a natural way: $\Phi_{vs} = \{\phi \mid \text{for all } x \in V,\ \phi(x) \in vs(x)\}$ (i.e. essentially the Cartesian product of the values sets). Conversely, a set of environments Φ for V, induces a value set $valset(\Phi)$ given by $valset(\Phi)(x) = \{v \in \mathbb{Z} \mid \exists \phi \in \Phi,\ \phi(x) = v\}$, which is the "projection" of the environments to each variable $x \in V$. Finally, we define a point-wise ordering on value sets as follows: $vs \preceq vs'$ iff $vs(x) \subseteq vs'(x)$ for each variable x in V. We denote the least element in this ordering by $vs_\bot = \lambda x.\emptyset$.

We can now define the value-set analysis $\mathcal{A}_{vset}$ for an interrupt-driven program $P = (V, T)$ as follows. Let $\mathcal{A}_{vset} = (D, \leq, d_0, F)$ where

- D is the set $L_P \to (V \to 2^{\mathbb{Z}})$ (thus an element of D associates a value-set with each program location)
- The ordering $d \leq d'$ holds iff $d(l) \preceq d'(l)$ for each $l \in L_P$
- The initial abstract value d_0 is given by:

$$d_0 = \lambda l. \begin{cases} \lambda x.\{0\} & \text{if } l = s_{main} \\ vs_\bot & \text{otherwise.} \end{cases}$$

- The transfer functions are given as follows. Given an abstract value d, and a location $l \in L_P$, we define vs^d_l to be the join of the value-set at l, and the value-set at all may-synchronizes-with edges coming into l. Thus $vs^d_l = d(l) \sqcup_\preceq \bigsqcup_{(n,l) \in MSW(G)} d(n)$. Below we will use Φ as an abbreviation of the set $\Phi_{vs^d_l}$ of environments induced by vs^d_l. Let $\iota = \langle l, c, l' \rangle$ be an instruction in P.
  - If c is the command `x := e` then $F_\iota(d) = d'$ where

$$d'(m) = \begin{cases} vs^d_l[x \mapsto [\![e]\!]_\Phi] & \text{if } m = l' \\ vs_\bot & \text{otherwise.} \end{cases}$$

- If c is the command `assume`(b), then $F_\iota(d) = d'$ where

$$d'(m) = \begin{cases} valset([\![b]\!]_\Phi) & \text{if } m = l' \\ vs_\perp & \text{otherwise.} \end{cases}$$

- If c is any other command (`skip`, `disableint`, `enableint`, `suspendsch`, `resumesch`, or `create`) then $F_\iota(d) = d'$ where

$$d'(m) = \begin{cases} vs_l^d & \text{if } m = l' \\ vs_\perp & \text{otherwise.} \end{cases}$$

Figure 6 shows the results of a value-set analysis on the sync-CFG of program P_2. The data-flow facts are shown just before a statement, at selected points in the program.

Soundness. The value-set analysis is sound in the following sense: if P is a *HB-race free* program, and we have a reachable state of P at a location l in a thread where a variable x is *read*; then the value of x in this state is contained in the value-set for x, obtained by the analysis at point l. More formally:

Theorem 1. *Let $P = (V, T)$ be an HB-race free interrupt-driven program, and let d^* be the result of the analysis $\mathcal{A}_{vset}$ on P. Let l be a location in a thread $t \in T$ where a variable x is read (i.e. P contains an instruction of the form $\langle l, c, l' \rangle$ where c is a read access of x). Let ϕ be an environment at l reachable via some execution of P. Then $\phi(x) \in d^*(l)(x)$.*

The proof of this theorem is similar to the one for classical concurrent programs in [11] (see [10] for a more accurate proof). The soundness claim can be extended to locations where a variable is "owned" (which includes locations where it is read). We say a variable x is *owned* by a thread t at location l, if an inserted read of x at this point is non-HB-racy in the resulting program.

Region-Based Analysis. One problem with the value-set analysis is that it may not be able to prove *relational* invariants (like $x \leq y$) for a program. One way to remedy this is to exploit the fact that concurrent programs often ensure race-free access to a *region* of variables, and to essentially do a region-based value-set analysis, as originally done in [21]. More precisely, let us say we have a partition of the set of variables V of a program P into a set of regions $R_1, \ldots, R_n$. We classify each read (write) access to a variable x in a region R, as an read (write) access to region R. We say that two instructions in an execution of P are involved in a *HB-region-race*, if the two instructions are conflicting accesses to the same region R, and are *not* happens-before ordered in the execution. A program is *HB-region-race free* if none of its executions contain a HB-region-race.

We can now define a region-based version of the value-set analysis for a program P, which we call $\mathcal{A}_{rvset}$. The value-set for a region R is a set of valuations (or sub-environments) for the variables in R. The transfer functions are defined in an analogous way to the value-set analysis. The analogue of Theorem 1 for regions gives us that for a HB-region-race free program, at any location where a region R is accessed, the region-value-set computed by the analysis at that point will contain every sub-environment of R reachable at that point.

6 Translation to Classical Lock-Based Programs

In this section we address the question of why an execution-preserving translation to a classical lock-based program is not a fruitful route to take. In a nutshell, such a translation would not preserve races and would induce a sync-CFG with many unnecessary MSW edges, leading to much more imprecise facts than the analysis on the native sync-CFG described in the previous section. We also describe how our approach can be viewed as a *lightweight* translation of an interrupt-driven program to a classical lock-based one. The translation is "lightweight" in the sense that it does *not* attempt to preserve the execution semantics of the given interrupt-driven program, but instead preserves races and the sync-CFG structure of the original program.

6.1 Execution-Preserving Lock Translation

One could try to translate a given interrupt-driven program P into a classical lock-based program P^L in a way that preserves the interleaved execution semantics of P. By this we mean that every execution of P has a corresponding execution in P^L that follows essentially the same sequence of interleaved instructions from the different threads (modulo of course the synchronization statements which may differ); and vice-versa. For example, to capture the semantics of `disableint-enableint`, one could introduce an "execution" lock E which is acquired in place of disabling interrupts, and released in place of enabling interrupts. Every instruction in a task thread outside a `disableint-enableint` block must also acquire and release E immediately before and after the instruction. Note that the latter step is necessary if we want to capture the fact that once a thread disables interrupts it cannot be preempted by any thread. Figure 5a shows an interrupt-driven program P_1 and its lock translation P_1^L in Fig. 5b. There are still issues with the translation related to re-entrancy of locks and it is not immediately clear how one would handle flag-based synchronization – but let us keep this aside for now.

The first problem with this translation is that it does not preserve race information. Consider the program P_1 in Fig. 5a and its translation P_1^L. The original program clearly has a race on x in statements 4 and 9. However the translation P_1^L does *not* have a race as the accesses are protected by the lock E. Hence checking for races in P^L does not substitute for checking in P. An alternative around this would be to first construct P' (recall that this is the version of P in which we introduce the `skip`-blocks around statements we want to check for races), then construct its lock translation $(P')^L$, and check this program for *high-level* races on the introduced `skip`-blocks. However this is expensive as it involves a 3x blow-up in going from P to P' and another 3x blow-up in going from P' to $(P')^L$. Further, checking for high-level races (for example using a lock-set analysis) is more expensive than just checking for races. In contrast, as we show next, our lock-set analysis on the native program P does not incur any of these expenses.

```
main:
1. x := y := t := 0;
2. create(t1);
3. create(t2);

t1:              t2:
4. x := x + 1;   8. disableint;
5. disableint;   9. t := x;
6. x := y;      10. enableint;
7. enableint;
```

(a) Example program P_1

```
main:
1. x := y := t := 0;
2. spawn(t1);
3. spawn(t2);

t1:              t2:
4. lock(E)       10. lock(E);
5. x := x + 1;   11. t := x;
6. unlock(E)     12. unlock(E);
7. lock(E)
8. x := y;
9. unlock(E)
```

(b) Exec-preserving trans. P_1^L

```
main:
1. x := y := t := 0;
2. spawn(t1);
3. spawn(t2);

t1:              t2:
4. x := x + 1;   8. lock(A);
5. lock(A);      9. t := x;
6. x := y;      10. unlock(A);
7. unlock(A);
```

(c) Lightweight trans. P_1^W

Fig. 5. Example program P_1, and its lock and lightweight translations P_1^L, P_1^W.

The second problem with a precise lock translation is that the sync-CFG of the translated program has many unnecessary MSW-edges, leading to imprecision in the ensuing analysis. Consider the program P_2 in Fig. 6, and its lock translation P_2^L in Fig. 7. P_2 is similar to P_1 except that line 4 is now an increment of y instead of x, and the resulting program is race-free (in fact HB-race-free). Notice that the may-sync-with edges from line 13 to 4, and line 6 to 10 in the sync-CFG of P_2^L in Fig. 7 are *unnecessary* (they are not present in the native sync-CFG) and lead to imprecise facts in an interval analysis on this graph. Some of the final facts in an interval analysis on these graphs are shown alongside the programs in Figs. 6 and 7. In particular the analysis on P_2^L is unable to prove the assertion in line 10 of the original program.

6.2 A Lightweight Lock-Translation

Our disjoint block-based approach of Sect. 5 can be viewed as a *lightweight* lock translation which does not attempt to preserve execution semantics, but preserves disjoint blocks and hence also races and the sync-CFG structure of the original interrupt-driven program.

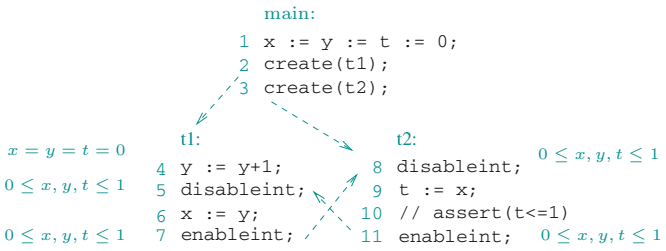


Fig. 6. Program P_2 with its Sync-CFG and facts from an interval analysis

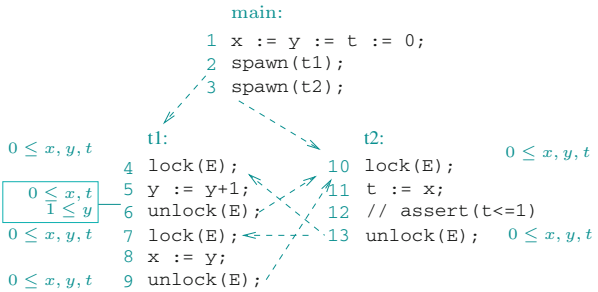


Fig. 7. Lock translation P_2^L of P_2, with its Sync-CFG and interval analysis facts

Let us first spell out the translation. Let us fix an interrupt-driven program $P = (V, T)$. The idea is simply to introduce a lock corresponding to each pattern of disjoint block pairs listed in Fig. 4, and to insert at the entry and exit to these blocks an acquire and release (respectively) of the corresponding lock. For each of the cases (a) through (h) we introduce locks named A through H, with some exceptions. Firstly, for case (f) regarding the `create` of a thread t, we simply translate these as a `spawn`(t) command in a classical lock-based programming language, which has a standard acquire-release semantics. Secondly, for case (h), we need a copy of H for *each* thread t, which we call H_t. This is because the concerned blocks (say between a set and unset of the flag f) are *not* disjoint across *task* threads, but only with the "then" block of an ISR thread statement that checks if $f = 0$. The ISR thread now acquires the set of locks $\{H_t \mid t \in T\}$ at the beginning of the "then" block of the `if` statement, and releases them at the end of that block. We call the resulting classical lock-based program P^W. Figure 5c shows this translation for the program P_1.

Figure 8 shows this translation along with the sync-CFG edges and some of the final facts in an interval analysis for the program P_2.

It is not difficult to see that P^W allows all executions that are possible in P. However it also allows more: for example the execution of P_1^W (Fig. 5c) in which thread $t1$ preempts $t2$ at line 9 to execute the statement at line 4, is *not* allowed in P_1. Thus it only *weakly* captures the execution semantics of P. However, every race in P is also a race in P^W. To see this, suppose we have a race on statements s and t in P. This means there is a high-level race on the two skip blocks around s and t in the augmented program P'. Since an execution exhibiting the high-level race on these blocks would also be present in $(P')^W$ which is identical to $(P^W)'$, it follows that the corresponding statements are racy in P^W as well.

Further, since our translation preserves disjoint blocks by construction, if s and t are in disjoint blocks in P, the corresponding statements will be in disjoint blocks in P^W; and vice-versa. It follows that the sync-CFGs induced by P and P^W are essentially isomorphic (modulo the synchronization statements). As a result, any value-set-based analysis will produce identical results on the two graphs.

Finally, if statements s and t are HB-racy in P, they must also be HB-racy in P^W. This is because disjoint blocks are preserved and the synchronizes-with relation is inherited from the disjoint blocks. Hence the execution witnessing the HB-race in P would also be present in P^W, and would also witness a HB-race on the corresponding statements.

We summarize these observations below:

Proposition 1. *Let P be an interrupt-driven program and P^W the classical lock program obtained using our lightweight lock translation. Then:*

1. *If statements s and t are racy in P, the corresponding statements are racy in P^W as well.*
2. *If statements s and t are HB-racy in P, the corresponding statements are HB-racy in P^W as well.*
3. *The sync-CFGs induced by P and P^W are essentially isomorphic. As a result the final facts in a value-set-based analysis on these graphs will be identical.*

□

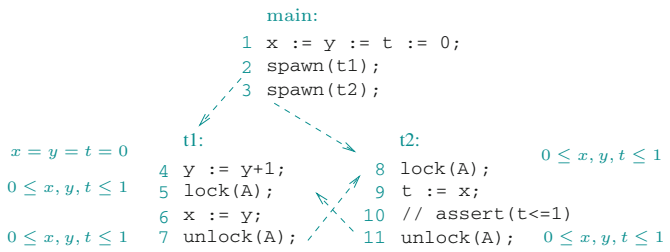


Fig. 8. Our lightweight translation P_2^W of P_2, with its Sync-CFG and interval analysis facts

6.3 Lockset Analysis for Race Detection

For classical lock-based programs, the lockset analysis [24] essentially tracks whether two statements are in disjoint blocks. Here two blocks are disjoint if they hold the same lock for the duration of the block. When two statements are in disjoint blocks, they are necessarily happens-before ordered, and hence this gives us a way to declare pairs of statements to be non-HB-racy.

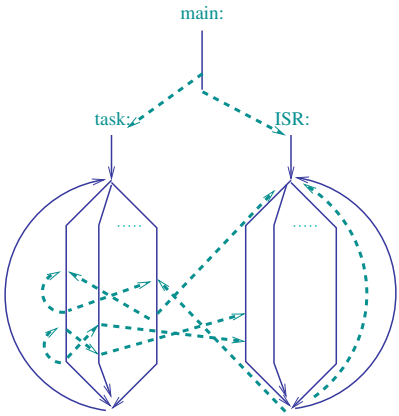
A lockset analysis computes the set of locks held at each program point as follows: at program entry it is assumed that no locks are held. When a call to acquire(l) is encountered, the analysis adds the lock l at the *out* point of the call. When a call to release(l) is encountered the lockset at the *out* point of the call is the lockset computed at the *in* point with the lock l removed. For any other statement, the lockset from the *in* point of the statement is copied to its *out* point. The *join* operation is the simple intersection of the input locksets. Once locksets are computed at each point, a pair of conflicting statements s and

t in different threads are declared to *may* HB-race if the locksets held at these points have no lock in common.

Using our lock translation above, we can detect races as follows. Given an interrupt-driven program P, we first translate it to the lock-based program P^W, and do a lockset analysis on P^W. If any pair of conflicting statements s and t are found to be may-HB-racy in P^W, we declare them to be may-HB-racy in P. By Proposition 1(2), it follows that this is a sound analysis for interrupt-driven programs.

7 Analyzing the FreeRTOS Kernel Library

We now perform an experimental evaluation of the proposed race detection algorithm and sync-CFG-based relational analysis for interrupt-driven programs. We use the FreeRTOS kernel library [3], on which our interrupt-driven program semantics are based, to perform our evaluation. FreeRTOS is a collection of functions mostly written in C, that an application developer compiles with and invokes in the application code. We view the FreeRTOS kernel library as an interrupt-driven program as follows: we build an interrupt-driven program out of the FreeRTOS kernel as shown in the figure alongside. The main thread is responsible for initializing the kernel data structures and then creating two threads: a *task* thread which branches out calling each task kernel API function, and loops on this; and an *ISR* thread which similarly branches and loops on the ISR kernel API functions. FreeRTOS provides versions of API functions that can be called from interrupt service routines. These functions have "FromISR" appended to their name. While it is sufficient to have one ISR thread, we assume (in the analysis) that there could be any number of task threads running. To achieve this we simply add sync-edges *within* each task kernel function, in addition to the usual sync-edges between task functions. We used FreeRTOS version 10.0.0 for our experiments. We conducted these experiments on an Intel Core i7 machine with 32 GB RAM running Ubuntu 16.04.



7.1 Race Detection

We consider 49 task and queue API functions that can be called from an application (termed top-level functions) for race detection. The functions operating on semaphores and mutexes were not considered.

We prepared the API functions for analysis, in two steps: (1) inlining and (2) lock insertion, as follows: The function `vTaskStartScheduler` and the queue initialization code in the function `xQueueGenericCreate` were treated as part of the main thread, which initializes kernel data structures. All the helper function calls made inside the top-level functions were inlined. After inlining, the functions are modified to acquire and release locks using the strategy explained in Sect. 6.2. We consider each pair of disjoint blocks as taking the same distinct lock. For example, the pair of disjoint blocks protected by `disableint-enableint` take lock *A*. That is `disableint` is replaced with `acquire`(A) and `enableint` is replaced with `release`(A). A total of 9 locks corresponding to disjoint blocks were employed in the modification of the FreeRTOS code. The two steps outlined above are automated. Inlining is achieved using the `inline` pass in the CIL framework [22]. Lock insertion is accomplished using a script.

The modified code, which has over 3.5K lines of code, is used for race detection. We tracked 24 variables and check whether the statements accessing them are racy. These variables include fields in the queue data-structure, task control block, and queue registry, as well as variables related to tasks. FreeRTOS maintains lists for the states of the tasks like "ready", "suspended", "waiting to send", etc. The pointers to these lists are also analysed. Access to any portion of a list (like the delayed list) is treated as an access of a corresponding variable of the same name.

Races are detected in this modified FreeRTOS code in three steps - (1) compute locks held, (2) identify whether access of a variable is a read or write, and (3) report potential races. First a lockset analysis, as explained in Sect. 6.3, to compute locks held at each access to variables, is implemented as a pass in CIL. The modified FreeRTOS code is analyzed using this new pass and the lockset at each access to the 24 variables of interest is computed. Then, a `writes` pass to identify whether accesses to variables are "read" or "write", also implemented in CIL, is run on the modified FreeRTOS code. Finally, a shell script to interpret both the results in the previous steps and report potential races is employed. The script identifies the conflicting access pairs (using the `writes` pass) and the locks held by the conflicting accesses (using lockset pass).

Our analysis reports 64 pairs of conflicting accesses as being potentially racy. On manual inspection we classified 18 of them are real races and the rest as false positives. Table 2 summarizes our findings. The second column in the table lists the variables of interest involved in the race, like various task list pointers, queue registry fields `pcQueueName` and `xHandle`, task variable `uxCurrentNumberOfTasks`, tick count `xTickCount`, etc. The third column lists the functions in which the conflicting accesses are made and the fourth gives the number of racing pairs. The fifth column assesses the potential races based on our manual inspection of the code. The analysis took 3.91 s.

The false positives were typically due to the fact that we had abstracted data-structures (like the delayed list which is a linked-list) by a synonymous variable. Thus even if the accesses were to different parts of the structure (like

the container field of a list item and the next pointer of a different list item) our analysis flagged them as races.

We were in touch with the developers of FreeRTOS regarding the 18 pairs we classified as true positives. The 14 races on the queue registry were deemed to be non-issues as the queue delete function is usually invoked only once the application is about to terminate. The 2 races on `uxCurrentNumberOfTasks` are known (going by comments in the code) but are considered benign as the variable is of "base type". The remaining couple of races on the delayed task lists appear to be real issues as they have been fixed (independent of our work) in v10.1.1.

7.2 Region-Based Relational Analysis

Our aim here is to do a region-based interval and polyhedral analysis of a region-race-free subset of the FreeRTOS kernel APIs, and to prove some simple assertions about the kernel variables in each region.

We first identified six regions for this purpose. One region corresponds to variables protected by disabling interrupts (like `xTickCount`, `xNextTaskUnblockTime`, etc.), while variables protected by suspend and resume scheduler commands (like `uxPendedTicks`, `xPendingReadyList`, etc.) are in another region. Fields of the queue structure like `pcHead`, `pcTail`, etc. are in a third region, while the waiting lists for a queue form another region. The queue registry fields like `pcQueueName` and `xHandle` are in region 5. The pointer variable `pxCurrentTCB`, pointing to the current Task Control Block (TCB), is put in the sixth region.

The FreeRTOS code was modified further to reflect access to regions. For this new variables $R_1, \ldots, R_6$, are declared. Wherever there is a write (or read) access to a variable in region i an assignment statement that defines (or reads from) variable R_i is inserted just before the access. This is done using a script which takes the result of the `writes` pass to find where in the source code an appropriate assignment statement has to be inserted. We selected 15 APIs that did not contain any region races.

Next, we prepared the API functions for the analysis in two steps. They are described below:

Abstraction of FreeRTOS API Functions. We abstracted the FreeRTOS source code to prepare it for the relational analysis. In this abstraction, we basically model the various lists (ready list, delayed list) by their lengths and the value at the head of the list (if required). Using this abstraction, we are able to convert list operations to operations on integers.

Similarly, to model insertion into a list, we abstract it by incrementing the variable which represents the length of the list. We abstracted all the API functions in a similar fashion.

Creation of the Sync-CFG. The next step is to create a sync-CFG out of the abstracted program. For doing this, we used the abstracted version of the FreeRTOS code (along with acquire-release added as explained in Sect. 7.1).

Table 2. Potential races

Variables	Functions	#Race pairs	Remark
`pxDelayedTaskList`	`eTaskGetState` `xTaskIncrementTick`	1	Real race. Read of `pxDelayedTaskList` in `eTaskGetState` while it is written to in `xTaskIncrementTick`
`pxOverflowDelayedTaskList`	`eTaskGetState` `xTaskIncrementTick`	1	Real race. (similar as above)
`uxCurrentNumberOfTasks`	`xTaskCreate` `uxTaskGetNumberOfTasks`	2	Real race. Unprotected read in `uxTaskGetNumberOfTasks` while it is written to in `xTaskCreate`
`pcQueueName` `xHandle`	`vQueueDelete` `pcQueueGetName` `vQueueAddToRegistry`	14	Real race. Unprotected accesses in queue registry functions
`xTasksWaitingToSend` `xTasksWaitingToReceive`	`eTaskGetState` `xQueueGenericReset`	2	False positive. Initialization of vars when queue is created
`pxDelayedTaskList` `pxOverflowDelayedTaskList` `xSuspendedTaskList` `pxCurrentTCB`	9 functions like `xTaskCreate`, `eTaskGetState`, etc.	11	False positive. Initialization of vars when the first task is created
`pxDelayedTaskList` `pxOverflowDelayedTaskList` `xSuspendedTaskList` `xTasksWaitingToSend` `xTasksWaitingToReceive`	13 functions like `vTaskDelay`, `eTaskGetState`, etc.	33	False positive. The accesses are to disjoint portions of the lists

Next, we used a script to insert non-deterministic gotos from the point of release of a lock to the acquire of the same lock. Since we are using gotos for creation of sync-CFG, we keep all the API functions in main itself and evaluate a non-deterministic "if" condition before entering the code for an API function.

Results. For the purpose of analysis we listed out some numerical relations between kernel variables in the same region, which we believed should hold. We identified a total of 15 invariants including 4 invariants which involve relations between kernel variables. We then inserted assertions for these invariants at the key points in our source code like the exit of a block protecting a region.

We have implemented an interval-based value-set analysis and a region-based octagon and polyhedral analysis for C programs using CIL [22] as the front-end and the Apron library (version 0.9.11) [16]. We represent the sync-with edges of the sync-CFG of a program using goto statements from the source (release) to the target (acquire) of the may-synchronizes-with (MSW) edges.

We ran our implementation on the abstracted version of the FreeRTOS kernel library, with the aim of checking how many of the invariants it was able to prove. The abstracted code along with addition of gotos is about 1500 lines of code. We did a preliminary interval analysis on this abstracted sync-CFG and were able to prove 11 out of these 15 invariants. With a widening threshold of 30, the interval analysis takes under 5 min to run. As expected, the interval analysis could not prove the relational invariants.

We then did a region-based polyhedral analysis using the six regions identified above. For the region-based analysis, we used convex polyhedra domain with a widening threshold of 30. It is able to prove all the assertions we believed to be true. The analysis takes about 30 min to complete with the convex polyhedra domain and about 20 min with the octagon domain.

The results obtained by our analysis are shown in Table 3.

Table 3. Relational analysis results

Assertion	Interval Anal	Region Anal (Oct/Polyhedral)
xTickCount $\leq$ xNextTaskUnblockTime	No	Yes
head(pxDelayedTaskList) = xNextTaskUnblockTime	No	Yes
head(pxDelayedTaskList) $\geq$ TickCount	No	Yes
uxMessagesWaiting $\leq$ uxLength	No	Yes
uxMessagesWaiting ≥ 0	Yes	Yes
uxCurrentNumberOfTasks ≥ 0	Yes	Yes
lenpxReadyTasksLists ≥ 0	Yes	Yes
uxTopReadyPriority ≥ 0	Yes	Yes
lenpxDelayedTaskList ≥ 0	Yes	Yes
lenxPendingReadyList ≥ 0	Yes	Yes
lenxSuspendedTaskList ≥ 0	Yes	Yes
cRxLock ≥ -1	Yes	Yes
cTxLock ≥ -1	Yes	Yes
lenxTasksWaitingToSend ≥ 0	Yes	Yes
lenxTasksWaitingToReceive ≥ 0	Yes	Yes

8 Related Work

We classify related work based on the main topics touched upon in this paper.

Data Races. Adve and Hill [1] introduce the notion of a data race using a happens-before relation, and identify instructions that form release-acquire pairs, for low-level concurrent programs. Boehm and Adve [4] define races in terms of consecutive occurrences in a sequentially consistent execution, as well as using a happens-before order, in the context of the C++ semantics. They show their notions are equivalent as far as race-free programs go. As pointed out earlier, the definition of races as consecutive occurrences is inadequate in our setting. Schwarz *et al.* [26] define a notion of data race for priority-based interrupt-driven programs, where there is a single main task and multiple ISRs. A race occurs when the main thread is accessing a variable at a certain dynamic priority, and an ISR thread with higher priority also accesses the variable. Our definition can be seen to be stronger and more accurately captures racy situations. In particular,

if the ISR thread with higher priority does not actually execute the conflicting access, due to say a condition not being enabled, then we would *not* call it a race. The term "high-level" race was coined by Artho *et al.* [2]. Our definition of a high-level race follows that of [20].

Analysis of Interrupt-Driven Programs. Regehr and Cooprider [23] describe a source-to-source translation of an interrupt-driven program to a standard multi-threaded program, and analyze the translated program for races. Their translation is inadequate for our setting in many ways: in particular, disable-enable of interrupts is translated by acquiring and releasing all ISR-specific locks; however this does not prevent interaction with another task while one task has disabled interrupts. In [8] they also describe an analysis framework for constant-propagation analysis on TinyOS applications. They use a similar idea of adding "control-flow" edges between disable-enable blocks and ISRs. However no soundness argument is given, and other kinds of blocks (suspend/resume, flag-based synchronization) are not handled. The works in [5,6,13] analyze timing properties, interrupt-latency, and stack sizes for interrupt-driven programs, using model-checking, algebraic, and algorithmic approaches. Schwarz *et al.* [25,26] give analyses for race-detection and invariants based on linear-equalities for their aforementioned class of priority-based interrupt-driven programs. Our work differs in several ways: Their analysis is directed towards *applications* (we target *libraries* where task priorities do not matter), their analyses are specific (we provide a basis for carrying out a variety of value-set and relational analyses, targeting race-free programs), they consider priority and flag-based synchronization (but not disable-enable and suspend-resume based synchronization). Sung and others [27] consider interrupt-driven applications in the form of ISRs with different priorities, and perform interval-based static analysis for checking assertions. They do not handle libraries and do not leverage race-freedom. Finally, [20] uses a model-checking approach to find all high-level races in FreeRTOS with a completeness guarantee.

Analysis of Race-Free Programs. Chugh *et al.* [7] use race information to do thread-modular null-dereference analysis, by killing facts at a point whenever a notional read of a variable is found to be racy. De *et al.* [11] propose the sync-CFG and value-set analysis for race-free programs, while Mukherjee *et al.* [21] extend the framework to region and relational analyses. Gotsman *et al.* [12] and Miné *et al.* [18,19] define relational shape/value analyses for concurrent programs that exploit race-freedom and lock invariants respectively. All these works are for classical lock-based synchronization while we target interrupt-driven programs.

9 Conclusion

In this paper our aim has been to give efficient static analyses for classes of non-standard concurrent programs like interrupt-driven kernels, that exploit the property of race-freedom. Towards this goal, we have proposed a definition of

data races which we feel is applicable to general concurrent programs. We have also proposed a general principle for defining synchronizes-with edges, which is the key ingredient of a happens-before relation, based on the notion of disjoint blocks. We have implemented our theory to perform sound and effective static analysis for race-detection and invariant inference, on the popular real-time kernel FreeRTOS.

We feel this framework should be applicable to other kinds of concurrent systems, like other embedded kernels (for example TI-RTOS [14]) and application programs, and event-driven programs. There are additional challenges in these systems like priority-based preemption and priority inheritance conventions which need to be addressed. Apart from investigating these systems we would like to apply this theory to perform other static analyses like null-dereference, points-to, and shape analysis, for these non-standard classes of concurrent programs.

References

1. Adve, S.V., Hill, M.D.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. **4**(6), 613–624 (1993)
2. Artho, C., Havelund, K., Biere, A.: High-level data races. J. Softw. Test. Verif. Reliab. **13**, 207–227 (2003)
3. Barry, R.: The FreeRTOS kernel, v10.0.0 (2017). https://freertos.org
4. Boehm, H., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, USA, pp. 68–78. ACM (2008)
5. Brylow, D., Damgaard, N., Palsberg, J.: Static checking of interrupt-driven software. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, Toronto, Ontario, Canada, 12–19 May 2001, pp. 47–56 (2001)
6. Chatterjee, K., Ma, D., Majumdar, R., Zhao, T., Henzinger, T.A., Palsberg, J.: Stack size analysis for interrupt-driven programs. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 109–126. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_7
7. Chugh, R., Voung, J.W., Jhala, R., Lerner, S.: Dataflow analysis for concurrent programs using data race detection. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, 7–13 June 2008, pp. 316–326 (2008)
8. Cooprider, N., Regehr, J.: Pluggable abstract domains for analyzing embedded software. In: Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2006), Ottawa, Canada, 14–16 June 2006, pp. 44–53 (2006)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM (1977)
10. De, A.: Access path based dataflow analysis for sequential and concurrent programs. Ph.D. thesis, Indian Institute of Science, Bangalore, December 2012

11. De, A., D'Souza, D., Nasre, R.: Dataflow analysis for data race-free programs. In: Proceedings of the 20th European Symposium on Programming ESOP 2011, Saarbrücken, Germany, 26 March – 3 April 2011, pp. 196–215 (2011)
12. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, 10–13 June 2007, pp. 266–277 (2007)
13. Huang, Y., Zhao, Y., Shi, J., Zhu, H., Qin, S.: Investigating time properties of interrupt-driven programs. In: Gheyi, R., Naumann, D. (eds.) SBMF 2012. LNCS, vol. 7498, pp. 131–146. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33296-8_11
14. Texas Instruments: TI-RTOS: A Real-Time Operating System for Microcontrollers (2017). http://www.ti.com/tool/ti-rtos
15. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
16. Jeannet Bertrand, M.A.: Apron numerical abstract domain library (2009). http://apron.cri.ensmp.fr/library/
17. Kini, D., Mathur, U., Viswanathan, M.: Dynamic race prediction in linear time. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pp. 157–170. ACM, New York (2017)
18. Miné, A.: Relational thread-modular static value analysis by abstract interpretation. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 39–58. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_3
19. Monat, R., Miné, A.: Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 386–404. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_21
20. Mukherjee, S., Kumar, A., D'Souza, D.: Detecting all high-level dataraces in an RTOS kernel. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 405–423. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_22
21. Mukherjee, S., Padon, O., Shoham, S., D'Souza, D., Rinetzky, N.: Thread-local semantics and its efficient sequential abstractions for race-free programs. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 253–276. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_13
22. Necula, G.: CIL – infrastructure for c program analysis and transformation (v. 1.3.7) (2002). http://people.eecs.berkeley.edu/~necula/cil/
23. Regehr, J., Cooprider, N.: Interrupt verification via thread verification. Electr. Notes Theor. Comput. Sci. **174**(9), 139–150 (2007)
24. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (1997)
25. Schwarz, M.D., Seidl, H., Vojdani, V., Apinis, K.: Precise analysis of value-dependent synchronization in priority scheduled programs. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 21–38. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_2
26. Schwarz, M.D., Seidl, H., Vojdani, V., Lammich, P., Müller-Olm, M.: Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In: Proceedings of the ACM SIGPLAN-SIGACT Principles of Programming Languages (POPL), pp. 93–104 (2011)

27. Sung, C., Kusano, M., Wang, C.: Modular verification of interrupt-driven software. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, 30 October – 3 November 2017, pp. 206–216 (2017)
28. Tarski, A., et al.: A lattice-theoretical fixpoint theorem and its applications. Pac. J. Math. **5**, 285–309 (1955)

An Abstract Domain for Trees with Numeric Relations

Matthieu Journault[1(✉)], Antoine Miné[1,2(✉)], and Abdelraouf Ouadjaout[1(✉)]

[1] Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, 75005 Paris, France
{matthieu.journault,antoine.mine,abdelraouf.ouadjaout}@lip6.fr
[2] Institut universitaire de France, Paris, France

Abstract. We present an abstract domain able to infer invariants on programs manipulating trees. Trees considered in the article are defined over a finite alphabet and can contain unbounded numeric values at their leaves. Our domain can infer the possible shapes of the tree values of each variable and find numeric relations between: the values at the leaves as well as the size and depth of the tree values of different variables. The abstract domain is described as a product of (1) a symbolic domain based on a tree automata representation and (2) a numerical domain lifted, for the occasion, to describe numerical maps with potentially infinite and heterogeneous definition set. In addition to abstract set operations and widening we define concrete and abstract transformers on these environments. We present possible applications, such as the ability to describe memory zones, or track symbolic equalities between program variables. We implemented our domain in a static analysis platform and present preliminary results analyzing a tree-manipulating toy-language.

1 Introduction

The abstract interpretation framework [5] enables the development of sound static analyzers by inferring and proving invariants on reachable states of programs. Invariants in the scope of abstract interpretation are elements of a lattice called an abstract domain. Most domains focus on numeric or pointer variables. By contrast, we propose an abstract domain for variables whose values are tree data-structures. Tree values appear natively in some languages (such as OCaml) and applications (such as the DOM in web programming) or can be encoded through pointer manipulations (as in C). Trees can abstract terms in logic programming. A tree domain can also be useful to collect symbolic expressions appearing in a program.

This work is supported by the European Research Council under Consolidator Grant Agreement 681393 – MOPSA.

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 724–751, 2019.
https://doi.org/10.1007/978-3-030-17184-1_26

```
typedef struct node
{
  int data;
  struct node* next;
} node;

node* append(node* head, int data)
{
  if (head==NULL) {
    return (create(data, NULL));
  } else {
    node *cursor=head;
    while(cursor->next != NULL)
      cursor=cursor->next;
    node* new_node=create(data,NULL);
    cursor->next=new_node;
    return head;
  }
}
```

Program 1: Append to list in C

```
float golden_ratio(int n) {
  int i = 0;
  float r = 1;
  while (i < n) {
    r = 1 + 1 / r;
    i += 1;
  }
  return r;
}
```

Program 2: Golden ratio in C

```
let rec f x n =
  match n with
  | 0 -> []
  | _ -> (x+1)::(x-1)::(f x (n-1))

let () =
  (*Assume x:int and n:int>=0*)
  let t = f x n in
  match t with
  | [] -> ()
  | p :: q when p > x -> ()
  | _ -> assert false
```

Program 3: List type in OCaml

Used Memory Zones. Program 1 describes an `append` function defined in the C language, this function adds an integer at the end of a linked list. The infinite set of unbounded terms of the form `*(*( ...*(head + 4) ...+ 4) + 4)` represents memory zones that are used by the `append` function. Our analyzer is able to infer and represent such sets of terms. This provides the information that Program 1 does not use any of the `data` field of the linked list. Such a function would be fairly commonly called in a real-life project. In a classical top-down static analysis by abstract interpretation, function calls are inlined at each call site. A way to improve scalability is to design modular analyzers able to reuse previous analysis results (as emphasized in [7]). In order to be able to successfully reuse function body analysis, input states must be unified. Moreover the cost of performing the analysis of the body of functions grows with the number of variables that need to be tracked. A common way to deal with both problems is to use framing on the inputs of the functions (as in separation logic [25]). This improves (1) precision: as we know that they are not modified by the function call, (2) body analysis efficiency: as the input state is reduced and finally (3) modularity: as constraints on the usage of the first analysis are relaxed by the removal of constraints.

Symbolic Relations. Program 2 is a C function computing an approximation of the golden ration (as it is the limit of the sequence $r_0 = 1$, $r_{n+1} = 1 + \frac{1}{r_n}$). As classical numerical domains can not represent such numerical relations, methods were proposed to track symbolic equality between expressions (see [23]). However such methods can not handle the unbounded iteration of Program 2. The set of reachable states at the end of Program 2 can be expressed by $\mathtt{r} = 1 + 1/(1 + 1/\ldots 1\ldots)$ with depth `n`. Please note that to infer such results we need to express numerical relations between the size of trees and the numeric variables from the program.

Numerical Environment. Consider now the OCaml Program 3, we want to prove that the `assert false` expression is never reached. This program builds a list of size $2 * \mathtt{n}$ with alternating values $\mathtt{x}+1$ and $\mathtt{x}-1$. The assertion states that the head of the list is $\mathtt{x}+1$. After the definition of `t` there are two types of reachable states. (1) Those that have not gone through the loop ($\mathtt{t} \mapsto \mathtt{[]}, \mathtt{x} \mapsto \mathbb{Z}, \mathtt{n} \mapsto 0$), and (2) those that have gone through at least one iteration of the loop: ($\mathtt{t} \mapsto \mathtt{[}a_1\mathtt{;}a_2\mathtt{;}a_3\mathtt{;\ ...]}, x \mapsto \alpha, \mathtt{n} > 0, a_1 \mapsto \alpha + 1, a_2 \mapsto \alpha - 1, a_3 \mapsto \alpha + 1$), where $\alpha \in \mathbb{Z}$. Therefore we need to be able to keep numerical relations between the parametric and unbounded number of numeric values appearing in `t` and numeric variables from the program. Classical numeric domains do not provide out-of-the-box abstractions for sets of partially defined numerical functions, therefore we define such an abstraction. As an example of analysis result, the memory representation obtained by our analysis for `t` describes the set of trees of the form: `Cons(`a`, Cons(`b`, Cons(`a`, ..., Nil) ...))` where $a = \mathtt{x} + 1$ and $b = \mathtt{x} - 1$. Therefore we are able to prove that the `assert false` expression is never reached.

Contributions. The main contributions of the article are threefold: (1) The extension of results on tree automata to the abstract interpretation framework by definition of a widening operator, in order to represent the set of tree shapes that a variable can contain. (2) The definition of a numerical domain built upon classical abstract domains able to represent sets of partial numerical maps with heterogeneous and unbounded definition sets. This is necessary to represent the numeric values at the leaves of a set of trees, as trees are unbounded and can contain a different number of leaves. (3) The definition of a novel abstraction for trees that can contain numerical values at their leaves. This last domain combines the abstractions (1) and (2). Moreover it is relational as it can express relations between numerical values found in trees and in the rest of the program, and relations between trees. Finally all results were implemented in an existing framework and experimented on a toy-language.

Limitations. At this point, analyses can only be performed on the toy language presented thereinafter, not on real life code, therefore we do not present any benchmark results, even though examples of analysis results will be put forth. Indeed Programs 1, 2 and 3 were precisely analyzed once encoded into our toy-language (see Programs 4 and 5).

Outline. We start, in Sect. 2, by presenting the concrete semantic we want to abstract. In Sect. 3 we build a first abstraction which forgets numerical values and focuses on abstracting tree shapes. Section 4 presents a novel numerical abstract domain required for the definition of the abstract domain of Sect. 5, which aims at precisely representing numerical constraints between trees and program variables. In Sect. 6 we provide remarks on the implementation and results of the analyzer. Finally Sect. 7 mentions related works while Sect. 8 concludes.

Notations. Classical Galois connections (see [5]) are denoted $(A, \subseteq_A) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (B, \subseteq_B)$. When no best abstraction can be defined, we use the *representation* framework (as defined by Bourdoncle in [3], also known as concretization only framework), representations are denoted by $(A, \subseteq_A) \overset{\gamma}{\leftarrow} (B, \subseteq_B)$. $A \nrightarrow B$ denotes the set of partial maps from A to B, and $\lambda_{|A} x.f(x) \in B$ denotes the map in $A \to B$ that associates $f(x)$ to x. Finally when $f \in A \to C$ and $g \in B \to C$, with $A \cap B = \emptyset$, $f \uplus g$ is the function defined on $A \cup B$, that associates $f(x)$ (resp. $g(x)$) to x whenever $x \in A$ (resp. $x \in B$).

2 Syntax and Concrete Semantics

Definition 1. *An* alphabet $\mathcal{F}$ *is a finite set, a* ranked alphabet *is a pair* $\mathcal{R} = (\mathcal{F}, a)$ *where* $\mathcal{F}$ *is an alphabet and* $a \in \mathcal{F} \to \mathbb{N}$. *For* $f \in \mathcal{F}$, *we call* arity *of* f *the value* $a(f)$. *We assume that* $\mathbb{Z}$ *and* $\mathcal{F}$ *are disjoint and we define the set of* natural terms *over* $\mathcal{R}$ *(denoted* $T_{\mathbb{Z}}(\mathcal{R})$*) to be the smallest set defined by:*

- $\mathbb{Z} \subseteq T_{\mathbb{Z}}(\mathcal{R})$
- $\forall p \geq 0,\ f \in \mathcal{F}, t_1,\ \ldots, t_p \in T_{\mathbb{Z}}(\mathcal{R}),\ a(f) = p \Rightarrow f(t_1, \ldots, t_p) \in T_{\mathbb{Z}}(\mathcal{R})$

Moreover when $\mathcal{R}$ *contains at least one symbol of arity* 0, *we define* terms *over* $\mathcal{R}$ *(denoted* $T(\mathcal{R})$*) to be the smallest set defined by:*

- $\forall p \geq 0,\ f \in \mathcal{F}, t_1,\ \ldots, t_p \in T(\mathcal{R}),\ a(f) = p \Rightarrow f(t_1, \ldots, t_p) \in T(\mathcal{R})$

In the following, $\mathcal{F}_n$ *denotes the subset of* $\mathcal{F}$ *of arity* n. *Moreover given a term* $t \in T(\mathcal{R})$ *we denote* $f = \mathbf{head}(t) \in \mathcal{F}$ *and* $\mathbf{sons}(t)$ *a possibly empty tuple* $(t_1, \ldots, t_n)$ *of elements of* $T(\mathcal{R})$ *such that* $t = f(t_1, \ldots, t_n)$.

Remark 1. Numerical leaves are defined to contain integers, however this could be modified to rationals, real numbers or floats. We are parametric in the type of numeric values, as they are delegated to an underlying numerical domain.

Example 1. Consider the ranked alphabet $\mathcal{R} = \{\texttt{*}(1), \texttt{\&}(1), \texttt{+}(2), \texttt{x}(0)\}$, $u(n)$ means that symbol u has arity n. Then `&x` $\in T(\mathcal{R})$, but `*(&x+4)` $\in T_{\mathbb{Z}}(\mathcal{R})$, and `*(&x+4)` $\notin T(\mathcal{R})$. Using this alphabet we can model C pointer arithmetic.

Example 2. $U = \{+(x, y) \mid x \leq y\}$ and $V = \{+(x, +(z, y)) \mid x \leq y \wedge z \leq y\}$ are two sets of natural terms over $\mathcal{R} = \{+(2)\}$ which we use as running examples.

$$
\begin{array}{rcl}
\textit{tree-expr} & \triangleq & \mid \texttt{make\_symbolic}(\mathcal{F}, \\
& & \quad \textit{tree-expr}, \ldots, \textit{tree-expr}) \\
& & \mid \texttt{make\_integer}(\textit{expr}) \\
& & \mid \texttt{get\_son}(\textit{tree-expr}, \textit{expr}) \\
\textit{stmt} & \triangleq & \ldots \\
& & \mid \mathcal{T} \texttt{ = } \textit{tree-expr}
\end{array}
\qquad
\begin{array}{rcl}
\textit{sym-expr} & \triangleq & \mid \texttt{get\_sym\_head}(\textit{tree-expr}) \\
\textit{expr} & \triangleq & \ldots \\
& & \mid \texttt{get\_num\_head}(\textit{tree-expr}) \\
& & \mid \texttt{is\_symbol}(\textit{tree-expr}) \\
& & \mid \textit{sym-expr} \texttt{ == } \mathcal{F}
\end{array}
$$

Fig. 1. Syntax extension of the language

$$
\begin{aligned}
&\mathbb{E}[\![\texttt{make\_symbolic}(s \in \mathcal{F}_m, T_1, \ldots, T_m)]\!](E,F) = \{s(t_1, \ldots, t_m) \mid \forall i,\ t_i \in \mathbb{E}[\![T_i]\!](E,F)\} \\
&\mathbb{E}[\![\texttt{make\_integer}(e \in \textit{expr})]\!](E,F) = \mathbb{E}[\![e]\!](E,F) \\
&\mathbb{E}[\![\texttt{is\_symbol}(T)]\!](E,F) = \{\texttt{true} \mid \exists t \in \mathbb{E}[\![T]\!](E,F), \exists f \in \mathcal{R},\ t = f(\ldots)\} \\
&\qquad\qquad \cup \{\texttt{false} \mid \exists t \in \mathbb{E}[\![T]\!](E,F), t \in \mathbb{Z}\} \\
&\mathbb{E}[\![\texttt{get\_son}(T,e)]\!](E,F) = \{t \mid \exists i \in \mathbb{E}[\![e]\!](E,F),\ t' \in \mathbb{E}[\![T]\!](E,F), f \in \mathcal{F}_{m>i}, \\
&\qquad\qquad t' = f(t_0, \ldots, t_{m-1}) \wedge t_i = t\} \\
&\mathbb{E}[\![\texttt{get\_num\_head}(T)]\!](E,F) = \{i \in \mathbb{Z} \mid \exists t \in \mathbb{E}[\![T]\!](E,F),\ t = i\} \\
&\mathbb{E}[\![\texttt{get\_sym\_head}(T)]\!](E,F) = \{s \in \mathcal{R} \mid \exists t \in \mathbb{E}[\![T]\!](E,F),\ t = s(\ldots)\}
\end{aligned}
$$

Fig. 2. Concrete operations on natural terms

```
int i;
int n;
tree y;
assume(n >= 0);
i = 0;
y = make_symbolic("p",{});
while (i < n) {
  y = make_symbolic("*",
           {make_symbolic("+",
                 {y,
                  make_integer(4)
                 })
           });
  i = i+1;
 }
```

Program 4: `*(p+4)` iterated

```
int n; int i; int x; int rep;
tree t;
assume(n>=0);
i = 0;
t = make_symbolic("Nil",{});
while (i < n) {
  t = make_symbolic("Cons",
        {make_integer(x-1), t});
  t = make_symbolic("Cons",
        {make_integer(x+1), t});
  i = i + 1;
 };
if (get_sym_head(t) != "Nil") {
    rep = get_num_head(get_son(t,0));
    assert(rep > x);
 }
```

Program 5: List manipulation

Syntax of the Language and Concrete Operations. We assume already defined a small imperative language and extend it (in Fig. 1) with statements, tree expressions (*tree-expr*) which are expressions that are evaluated to trees, and simple symbol expressions (*sym-expr*) which enable the manipulation of symbols. We add the ability to build a tree which contains only a numerical leaf: $\texttt{make\_integer}(e)$, the ability to read the i-th son of a tree t: $\texttt{get\_son}(t,i)$, ... Figure 2 defines concrete operations over the set $\wp(T_{\mathbb{Z}}(\mathcal{R}))$. Figure 2 assumes given a set of program numerical variables $\mathcal{V}$, a set of numerical expressions (over $\mathcal{V}$) denoted *expr*, a set of statements *stmt*, a notion of numerical environment $E \in \mathfrak{E} = \mathcal{V} \to \mathbb{Z}$, a set of tree program variables $\mathcal{T}$, a notion of tree

environment $F \in \mathfrak{F} = \mathcal{T} \to \wp(T_{\mathbb{Z}}(\mathcal{R}))$, $D = E \times F$ is our concrete domain. Finally we assume already partially defined on numerical expressions an evaluation function $\mathbb{E}[\![e \in \mathit{expr}]\!](E \in \mathcal{V} \to \mathbb{Z}, F \in \mathcal{T} \to \wp(T_{\mathbb{Z}}(\mathcal{R}))) \in \wp(\mathbb{Z})$. Using this operator we are able to define Program 4 which computes the memory zones used by `append` from Program 1, and Program 5 that simulates the behavior of Program 3.

3 Natural Term Abstraction by Tree Automata

In this section we start by defining a value abstraction for tree sets (in Sect. 3.1), which is then lifted to an environment abstraction (in Sect. 3.2).

3.1 Value Abstraction

As a first abstraction for natural terms, we put aside numerical values and define an abstraction able to describe sets of tree shapes. Tree automata enable the description of set of terms built upon a finite ranked alphabet. The ranked alphabet of the language we want to analyze is extend with the $\square$ symbol to denote potential positions of numerical values.

Definition 2 (Finite tree automata). *A* finite tree automaton *(FTA) over a ranked alphabet $\mathcal{R}$ is a tuple $(Q, \mathcal{R}, Q_f, \delta)$, where Q is a (finite) set of states, $Q_f \subseteq Q$ is the set of final states, and $\delta \in \wp(\bigcup_{n\in\mathbb{N}} \mathcal{F}_n \times Q^n \times Q)$ is the set of transitions. We define $\overline{\delta} : (\bigcup_{n\in\mathbb{N}} \mathcal{F}_n \times Q^n) \to \wp(Q)$ by: $\overline{\delta}(f, \overrightarrow{q}) = \{q' \mid (f, \overrightarrow{q}, q') \in \delta\}$. When $\overline{\delta}$ is such that, $\forall n \in \mathbb{N},\ f \in \mathcal{F}_n,\ \overrightarrow{q} \in Q^n,\ |\overline{\delta}(f, \overrightarrow{q})| = 1$, we say that the automaton is complete and deterministic (CDFTA). We then abuse notations and denote by $\delta(f, \overrightarrow{q})$ the unique element in the set $\overline{\delta}(f, \overrightarrow{q})$.*

Definition 3 (Reachability). *Given a FTA $\mathcal{A} = (Q, \mathcal{R}, Q_f, \delta)$ we define, a reachability function* $\textsc{reach}_{\mathcal{A}} : T(\mathcal{R}) \to \wp(Q)$

$$\textsc{reach}_{\mathcal{A}}(t) = \textbf{let } t_1, \ldots, t_n = \textbf{sons}(t) \textbf{ in} \bigcup_{(q_1,\ldots,q_n)\in(\textsc{reach}_{\mathcal{A}}(t_1),\ldots,\textsc{reach}_{\mathcal{A}}(t_n))} \overline{\delta}(\textbf{head}(t), (q_1, \ldots, q_n))$$

If $\textbf{sons}(t)$ *is the empty tuple (which is the case when t is a constant a), the union is made over a unique element (which is the empty tuple), which then boils down to: $\overline{\delta}(a, ())$. If* $\textbf{sons}(t)$ *is not the empty tuple and for some i,* $\textsc{Reach}_{\mathcal{A}}(t_i)$ *is empty, then* $\textsc{Reach}_{\mathcal{A}}(t)$ *is also empty.*

Example 3. Consider the ranked alphabet $\mathcal{R} = \{f(2), a(0)\}$, and the automaton $\mathcal{A} = (\{u, v\}, \mathcal{R}, \{v\}, \{a() \to u, f(v, v) \to v, f(u, u) \to u, f(u, u) \to v\})$. Then $\textsc{reach}_{\mathcal{A}}(a) = \{u\}$, $\textsc{reach}_{\mathcal{A}}(f(a, a)) = \{u, v\}$, $\textsc{reach}_{\mathcal{A}}(f(f(a, a), a)) = \{u, v\}$.

Definition 4 (Acceptance). *Given a FTA* $\mathcal{A} = (Q, \mathcal{R}, Q_f, \delta)$*, a term* t*, we say that* t *is* accepted *by the automaton if* $\text{REACH}_{\mathcal{A}}(t) \cap Q_f \neq \emptyset$*.* $\mathcal{L}(\mathcal{A})$ *denotes the set of terms accepted by automaton* $\mathcal{A}$*.*

Example 4. With the definition of Example 3, $\mathcal{L}(\mathcal{A})$ is the set of terms over $\mathcal{R}$ that contain at least one f.

Definition 5 (Tree regular languages). *A set of terms* $\mathcal{T}$ *over a ranked alphabet* $\mathcal{R}$ *is called* tree regular *if there exists a FTA* $\mathcal{A}$ *over* $\mathcal{R}$ *such that* $\mathcal{L}(\mathcal{A}) = \mathcal{T}$*. The set of such languages is denoted* $TReg(\mathcal{R})$*.*

Remark 2. As for regular languages, for all $\mathcal{A} \in$ FTA there exists $\mathcal{A}' \in$ CDFTA such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$, moreover $\mathcal{A}'$ is computable (see [4]).

Example 5.
- As proved in Example 4 the set of all terms over $\{f(2), a(0)\}$ that contain at least one f is tree regular.
- Consider now the ranked alphabet $\{a(1), b(1), \epsilon(0)\}$ and the set of terms $\mathcal{T} = \{\epsilon, a(b(\epsilon)), a(a(b(b(\epsilon)))), \dots\}$. We can prove (in a similar way as for $a^n b^n$ in regular languages) that $\mathcal{T}$ is not tree regular.
- On every ranked alphabet $\mathcal{R}$: every finite language, the empty language and $T(\mathcal{R})$ are tree regular.

Proposition 1. $(TReg(\mathcal{R}), \subseteq, \cap, \cup, .^c, \emptyset, T(\mathcal{R}))$ *is a complemented lattice with infinite height, moreover it is not complete.* $\subseteq, \cap, \cup$ *and complementation* $(.^c)$ *are computable operations on tree automata [4].*

We denote by $\mathcal{R}^{\square}$ the ranked alphabet $\mathcal{R}$ after adding the symbol $\square$ of arity 0 (we assume that $\square \notin \mathcal{R}$). Given a natural term t, we define $t^{\square}$ to be the term obtained by replacing every integer with the $\square$ symbol.

Proposition 2. $(\wp(T_{\mathbb{Z}}(\mathcal{R})), \subseteq) \overset{\gamma}{\leftarrow} (TReg(\mathcal{R}^{\square}), \subseteq)$ *where* $\gamma(\mathcal{A}) = \{t \mid t^{\square} \in \mathcal{L}(\mathcal{A})\}$ *is a representation. Moreover with such a* γ *definition,* $\cup$*,* $\cap$ *soundly represent the union and the intersection.*

Remark 3. We only have a representation and not a Galois connection as language $\mathcal{T}$ of Example 5 does not have a best tree regular over approximation.

Example 6. Let $\mathcal{R} = \{+(2)\}$ and $\mathcal{A} = (\{0, 1\}, \mathcal{R}^{\square}, \{0, 1\}, \{(\square() \to 0, +(0, 0) \to 1, +(0, 1) \to 1)\})$. Examples of terms recognized by $\mathcal{A}$ are shown on Fig. 3. Natural terms from our running example U and V (defined in Example 2) are also contained in $\gamma(\mathcal{A})$. Moreover as we do not provide numerical constraints: $1 + (3 + 4)$, 23, $1 + (2 + (3 + 4))$ are also elements in $\gamma(\mathcal{A})$.

Due to the infinite height of the lattice, a widening operator is required. In the following, we assume given a constant $w \in \mathbb{N}$, this constant will be used to stabilize increasing chains, the greater the constant, the more precise our widening operator will be.

Definition 6. *Let $\mathcal{A} = (Q, \mathcal{R}, Q_f, \delta) \in FTA$, and $\sim$ be an equivalence relation on Q, such that $p \sim q \wedge p \in Q_f \Rightarrow q \in Q_f$. We define $\mathcal{A}/\sim = (Q/\sim, \mathcal{R}, Q_f/\sim, \bigcup_{(f,q_1,\ldots,q_n,q)\in\delta}\{(f, q_1^{\sim}, \ldots, q_n^{\sim}, q^{\sim})\})$ where $q^{\sim}$ is the equivalence class of q in $\sim$.*

Proposition 3. *For every $\mathcal{A} \in FTA$ and every $\sim$ equivalence relation on its states, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}/\sim)$.*

Therefore following the idea from [9] and in [11], we define a widening operation by quotienting states of automata by an equivalence relation of finite index. We define by induction a special sequence of equivalence relations on states of tree automata: $\sim_1 = \{Q_f, Q \setminus Q_f\}$ and $\sim_{k+1}$ is $\sim_k$ where we split equivalence classes not satisfying the following condition: $\forall f \in \mathcal{F}_n,\ \forall p_1, \ldots, p_n \in Q,\ \forall q_1, \ldots, q_n \in Q, (\bigwedge_{i=1}^{n} p_i \sim_k q_i) \Rightarrow \delta(f, p_1, \ldots, p_n) \sim_k \delta(f, q_1, \ldots, q_n)$ and $\forall q \in Q_f,\ q^{\sim_k} \subseteq Q_f$. This sequence of equivalence relations is the Myhill-Nerode sequence (see [4]). This sequence is of length at most the number of states of the automaton (before stabilization). Let $\phi(w) = \max\{i \leq |Q| \mid \text{index of } \sim_i \leq w\}$ (given an integer w, ϕ yields the index of the most precise of the equivalence relationships in the Myhill-Nerode sequence, that contains at most w equivalence classes) and $[\mathcal{A}]_w = \mathcal{A}/\sim_{\phi(w)}$. $[\mathcal{A}]_w$ is therefore a FTA with at most w states such that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}([\mathcal{A}]_w)$. As for regular languages, for every CDFTA a equivalent minimal CDFTA (in the sense of the number of states, and unique modulo state renaming) can be obtained by quotienting the automaton by $\sim_{|Q|}$. Therefore we define a widening operator on CDFTAs, which is then lifted to tree regular languages.

Definition 7 (Widening operator $\triangledown$). $\mathcal{A} \triangledown \mathcal{A}' = [\mathcal{A} \cup \mathcal{A}']_w$.

Proposition 4. *This widening is sound and stabilizes infinite sequences.*

Remark 4. Consider the two following complete and deterministic tree automata: $\mathcal{A} = (\{a, b, h\}, \{+(2)\}, \{a\}, \{\Box() \to b, +(b, b) \to a\})$ and $\mathcal{B} = (\{a, b, c, h\}, \{+(2)\}, \{a\}, \{\Box() \to b, +(b, b) \to c, +(b, c) \to a\})$ (unmentioned transitions go to h). $\mathcal{A}$ (resp. $\mathcal{B}$) recognizes the tree $+(\Box, \Box)$ (resp. $+(\Box, +(\Box, \Box))$), it over-approximates U (resp. V) from our running example. $\mathcal{A} \cup \mathcal{B}$ is recognized by the following complete and deterministic tree automaton: $\mathcal{C} = (\{a, b, c, h\}, \{+(2)\}, \{a, c\}, \{\Box() \to b, +(b, b) \to c, +(b, c) \to a\})$. If we want to widen $\mathcal{A}$ and $\mathcal{B}$ with parameter 3, the following equivalence relation is computed: $\{\{h\}, \{b\}, \{a, c\}\}$. Merging equivalent states produces $(\{a, b, h\}, \{+(2)\}, \{a\}, \{\Box() \to b, +(b, b) \to a, +(b, a) \to a\})$, which contains a loop and over-approximates the union.

3.2 Environment Abstraction

Now that we are given an abstraction for natural term sets, let us show how this is lifted to a notion of abstract natural term environments mapping variables to natural terms. Given a set of natural term variables $\mathcal{T}$, consider $\mathfrak{F}^\sharp = (\mathcal{T} \to \mathrm{TReg}(\mathcal{R}^\square)) \cup \{\bot\}$ and the set operators defined by the point-wise lifting of operators on $\mathrm{TReg}(\mathcal{R}^\square)$. We also lift the concretization function $\wp(T_\mathbb{Z}(\mathcal{R})) \leftarrow \mathrm{TReg}(\mathcal{R}^\square)$ to $\mathfrak{F} \leftarrow \mathfrak{F}^\sharp$. We assume given an abstract numerical environment $E^\sharp$ and an abstract evaluator $\mathbb{E}[\![e]\!]^\sharp$. Abstract transformers $[\![\texttt{make\_symbolic}]\!]^\sharp$, $[\![\texttt{is\_symbol}]\!]^\sharp$, $[\![\texttt{get\_son}(e)]\!]^\sharp$, $[\![\texttt{get\_sym\_head}]\!]^\sharp$ and $[\![\texttt{get\_num\_head}]\!]^\sharp$ are simple tree automata operations. For concision Fig. 4 only provides definitions of two of these operators. Please note that these definitions require all states of the automata to be reachable. An example of use of the `is_symbol` operator can be found in Example 7. Other abstract operators are similar.

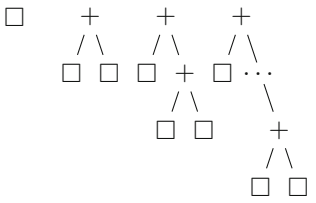


Fig. 3. Example of accepted trees from Example 6

$$\mathbb{E}^\sharp[\![\texttt{make\_integer}(e \in \mathit{expr})]\!](E^\sharp, F^\sharp) = \langle \{a\}, \mathcal{R}, \{a\}, \{\square() \to a\} \rangle$$

$$\mathbb{E}^\sharp[\![\texttt{get\_son}(T, e \in \mathit{expr})]\!](E^\sharp, F^\sharp) =$$

$$\bigcup_{\substack{(Q,\mathcal{R},Q_f,\delta) \in \mathbb{E}^\sharp[\![T]\!](E^\sharp,F^\sharp) \\ i \in \mathbb{E}^\sharp[\![e]\!](E^\sharp) \cap \{0,\ldots,m-1\}}} (Q, \mathcal{R}, \{q \in Q \mid \exists p \in Q_f,\ \exists s(p_0, \ldots, p_{m-1}) \to p \in \delta \wedge p_i = q\}, \delta)$$

Fig. 4. Abstract operators

Example 7. Consider the tree automaton $\mathcal{A}$ of Example 6, (Fig. 3), with $F^\sharp = (x \mapsto \mathcal{A})$: $[\![\texttt{get\_sym\_head}(x)]\!]^\sharp(E^\sharp, F^\sharp) = \{+\}$ and $[\![\texttt{get\_num\_head}(x)]\!]^\sharp(E^\sharp, F^\sharp) = \top$.

4 Numerical Abstractions

As emphasized in the introductory example, we rely on numerical domains to introduce constraints on numerical variables found in trees. In a classical numeric abstraction (e.g. intervals [6], octagons [22], polyhedra [8], ...), each abstract element represents a set of maps $\mathcal{V} \to \mathbb{R}$ for a fixed, finite set of variables $\mathcal{V}$. In contrast, our numeric variables are leaves of a possibly infinite set of trees of unbounded size. Hence before starting the presentation of the numerical abstraction for natural terms, we show how to extend in a generic way an abstract element in two steps. Firstly we want to be able to represent a set of maps, where each map is defined over a (possibly different) finite subset of an infinite set of variables (this is done in Sect. 4.1). Secondly, we use summarization variables to relax the finiteness constraint, so as to represent sets of maps over heterogeneous maps over infinitely many variables (done in Sect. 4.2).

4.1 Heterogeneous Support

We define $\mathfrak{M} \triangleq \wp(\mathcal{V} \nrightarrow \mathbb{R})$, the set of partial maps from $\mathcal{V}$, to $\mathbb{R}$. $\mathfrak{M}$ is ordered by the inclusion relation $\subseteq$. In the following $\mathbf{def}(f)$ denotes the definition set of f. We assume defined a representation $(\wp(\mathcal{S} \to \mathbb{R}), \subseteq) \xleftarrow{\gamma_0^{\mathcal{S}}} (N_{\mathcal{S}}, \sqsubseteq_0^{\mathcal{S}})$, for every finite set $\mathcal{S} \subseteq \mathcal{V}$ (such as octagons in $|S|$ dimensions). $N_{\mathcal{S}}$ comes with the usual abstract set operator $\sqcap_0^{\mathcal{S}}$, $\sqcup_0^{\mathcal{S}}$. Moreover if $x \in \mathcal{S}$, $y \notin \mathcal{S}$, $\mathcal{S}'$ is another finite set and $N^\sharp \in N_{\mathcal{S}}$ then $N^\sharp[x \mapsto y] \in N_{\mathcal{S} \cup \{y\} \setminus \{x\}}$ is the abstract element obtained by renaming x into y, $N^\sharp_{|\mathcal{S}'} \in N_{\mathcal{S}'}$ is obtained by existentially quantifying dimensions associated to elements in $\mathcal{S}$ and not in $\mathcal{S}'$ and adding unconstrained dimensions for elements in $\mathcal{S}'$ and not in $\mathcal{S}$. From now on we assume that this last operator is exact (as for intervals, octagons, polyhedra over $\mathbb{R}$). However results from this section can be extended to numerical domains that are able, given $N^\sharp \in N_{\mathcal{S}}$, $N^{\sharp\prime} \in N_{\mathcal{S}'}$, to check if $\gamma_0^{\mathcal{S}}(N^\sharp) \subseteq \gamma_0^{\mathcal{S}'}(N^{\sharp\prime})_{|\mathcal{S}}$. The precision of the extension defined in this subsection would then depend upon the precision of this test in the underlying domain. Finally $[\![.]\!]_0^{\mathcal{S}}$ (resp. $[\![.]\!]_0^{\sharp,\mathcal{S}}$) refers to the classical concrete (resp. abstract) semantic of operators on sets of numerical maps (resp. abstract elements). A classical method for the abstraction of heterogeneous maps is the use of a partitioning of the concrete element according to the definition set of its represented maps. However partitioning induces an increase in numerical operation cost (exponential in the number of variable) which we would like to avoid. Therefore in order to abstract sets of maps with heterogeneous definition sets, we start by abstracting the potential definition set. We choose a simple lower-bound/upper-bound abstraction (l and u in the following definition). Moreover we need to abstract the potential mappings given a definition set: this is done using a classical numerical domain. Contrary to partitioning, we will use only one numerical abstract element, defined on the upper-bound u, to represent all environments (instead of one abstract element by definition set). We also add a $\top$ element, used in the case where the upper bound u is infinite.

Definition 8 (Numerical abstraction). *Let us define the following set:* $\mathfrak{M}^\sharp \triangleq \{\langle N^\sharp, l, u\rangle \mid l, u \in \wp(V) \wedge l \text{ and } u \text{ are finite} \wedge l \subseteq u \wedge N^\sharp \in N_u \wedge N^\sharp \neq \bot_0^u\} \cup \{\top, \bot\}$. *An element of* $\mathfrak{M}^\sharp$ *is therefore: either* $\top$, $\bot$ *or a triple* $\langle N^\sharp, l, u\rangle$ *where* l *and* u *are finite sets of variables such that* $N^\sharp$ *is defined over* u.

Definition 9 (Concretization function). *Abstract elements from* $\mathfrak{M}^\sharp$ *are mapped to* $\mathfrak{M}$ *thanks to the following concretization function:* $\gamma(\bot) = \emptyset$, $\gamma(\top) = \mathfrak{M}$ *and* $\gamma(\langle N^\sharp, l, u\rangle) = \{\rho \in \mathcal{S} \to \mathbb{Z} \mid l \subseteq \mathcal{S} \subseteq u \wedge \rho \in \gamma_0^{\mathcal{S}}(N^\sharp)_{|\mathcal{S}})\}$.

Example 8. As an example consider $\gamma(\langle\{x = y, x \leq 3, z = 0\}, \{x\}, \{x, y, z\}\rangle) = \{(x \mapsto a) \mid a \leq 3\} \cup \{(x \mapsto a, y \mapsto a) \mid a \leq 3\} \cup \{(x \mapsto a, z \mapsto 0) \mid a \leq 3\} \cup \{(x \mapsto a, y \mapsto a, z \mapsto 0) \mid a \leq 3\}$. As intended, the resulting set of maps contains maps with different definition sets.

Definition 10 (Order). *On $\mathfrak{M}^\sharp$ we define the following comparison operator: $\langle N^\sharp, l, u\rangle \sqsubseteq \langle N^{\sharp\prime}, l', u'\rangle \Leftrightarrow l' \subseteq l \subseteq u \subseteq u' \wedge N^\sharp \sqsubseteq_0^u N^{\sharp\prime}_{|u}$, this comparison is trivially extended to $\top$ (resp. $\bot$) as being the biggest (resp. smallest) element in $\mathfrak{M}^\sharp$. In the following $\mathfrak{M}^\sharp_{\mathfrak{p}}$ denotes the subset of $\mathfrak{M}^\sharp$ where $u = \mathfrak{p}$ extended with $\top$ and $\bot$.*

Proposition 5. *γ is monotonic for $\sqsubseteq$.*

Figure 5 provides the definition of the concrete and abstract semantics of the classical numerical statements, `Assume` and `Assign` (denoted $x \leftarrow e$). We denote $\mathbf{vars}(e)$ the set of variables appearing in e. We recall that $[\![\texttt{Assume}(c)]\!]_0^{\mathcal{S}}(E \in \wp(\mathcal{S} \to \mathbb{R})) = \{f \in E \mid \texttt{true} \in \mathbb{E}[\![c]\!](f)\}$ and $[\![x \leftarrow e]\!]_0^{\mathcal{S}}(E \in \wp(\mathcal{S} \to \mathbb{R})) = \{f[x \mapsto e'] \mid f \in E \wedge e' \in \mathbb{E}[\![e]\!](f)\}$. In order to ease the lifting of these classical operators we define $[\![\texttt{stmt}]\!]_0(\mathcal{M} \in \mathfrak{M}) \triangleq \cup_{\mathcal{S} \text{ finite} \subseteq \mathcal{V}} [\![\texttt{stmt}]\!]_0^{\mathcal{S}}(\mathcal{M} \cap (\mathcal{S} \to \mathbb{R}))$, for every statement `stmt`. Moreover we assume the existence of the following abstract operators: $[\![\texttt{Assume}(c)]\!]_0^{\sharp,u}(N^\sharp)$ and $[\![x \leftarrow e]\!]_0^{\sharp,u} N^\sharp$ abstracting soundly their respective concrete transformers. Note that the concrete semantic of $\texttt{Assume}(c)$ (resp. $x \leftarrow e$) enforces that maps are defined at least on the variables appearing in c (resp. in e and on x). Abstract operators from Fig. 5 are sound with respect to γ and their concrete operators.

$$
\begin{aligned}
&[\![\texttt{Assume}(c)]\!](\mathcal{M}) = [\![\texttt{Assume}(c)]\!]_0(\{f \mid f \in \mathcal{M} \wedge \mathbf{vars}(c) \subseteq \mathbf{def}(f)\})\\
&[\![\texttt{Assume}(c)]\!]^\sharp(\langle N^\sharp, l, u\rangle) = \langle [\![\texttt{Assume}(c)]\!]_0^{\sharp,u}(N^\sharp), l \cup \mathbf{vars}(c), u\rangle\\
&[\![x \leftarrow e]\!](\mathcal{M}) = [\![x \leftarrow e]\!]_0(\{f \mid f \in \mathcal{M} \wedge \mathbf{vars}(e) \cup \{x\} \subseteq \mathbf{def}(f)\})\\
&[\![x \leftarrow e]\!]^\sharp(\langle N^\sharp, l, u\rangle) = \langle [\![x \leftarrow e]\!]_0^{\sharp,u}(N^\sharp), l \cup \mathbf{vars}(e) \cup \{x\}, u\rangle
\end{aligned}
$$

Fig. 5. Concrete and abstract semantic of usual numerical operators

We now need to define $\sqcup$ that abstracts the classic set operator $\cup$. We can not directly apply the corresponding abstract operator on the numerical component of the abstractions as they might have different definition sets. A first naive solution would be to extend their respective definition set and to perform the abstract operation on the resulting elements: $N^\sharp_{|u\cup u'} \sqcup_0^{u\cup u'} N^{\sharp\prime}_{|u\cup u'}$. However consider $M = \langle\{x = y\}(= U^\sharp), \{x, y\}, \{x, y\}\rangle$ and $N = \langle\{x = z\}(= V^\sharp), \{x, z\}, \{x, z\}\rangle$, where the underlying domain is the octagon domain where elements are represented as a set of linear constraints (e.g. $\{x = y\}$). We have $U^\sharp_{|\{x,y,z\}} = \{x = y\}$ and $V^\sharp_{|\{x,y,z\}} = \{x = z\}$, hence $U^\sharp_{|\{x,y,z\}} \sqcup_0^{\{x,y,z\}} V^\sharp_{|\{x,y,z\}} = \top$. Consider now the abstract element in $\mathfrak{M}^\sharp$: $R = \langle\{x = y, x = z\}(= W^\sharp), \{x\}, \{x, y, z\}\rangle$. The concretization of R over-approximates the union of the concretization of M and N, and its numerical component is more precise than $\top$. We note that the numerical constraints appearing in $W^\sharp$ could be found in $U^\sharp$ or $V^\sharp$, therefore in order to remove the aforementioned imprecision we define a refined abstract union operator, denoted as $\boxplus$, that uses constraints found in the inputs in order to refine its

Algorithm 1. strengthening operator

Input : $X^\sharp$, C: a set of constraints, $U^\sharp \in N_u$: a soundness threshold on environment u, $V^\sharp \in N_v$: a soundness threshold on environment v
Output: $Z^\sharp$ an abstract element over-approximating $U^\sharp$ on u and $V^\sharp$ on v

```
Z♯ ← X♯;
foreach c ∈ C do
    T♯ ← [[Assume(c)]]♯,u∪v_0(Z♯);
    if U♯ ⊑⁰_u T♯_|u ∧ V♯ ⊑⁰_v T♯_|v then
        Z♯ ← T♯;
    end
return Z♯;
```

result. This is done using the **strenghtening** operator of Algorithm 1 which adds constraints from C that do not make the projection of $X^\sharp$ to u (resp. v) lower than the threshold $U^\sharp$ (resp. $V^\sharp$). We assume that, given an abstract element $U^\sharp$, we can extract a finite set of constraints satisfied by $U^\sharp$, those are denoted $\mathbf{constraints}(U^\sharp)$ (the more constraints can be extracted, the more precise the result will be). For example if the numerical domain is the interval domain, constraints have the form $\pm x \geq a$. If the numerical domain is the octagon domain the **constraints** operator yields all the linear relations among variables that define the octagon.

Definition 11 ($\boxplus$ operator). *Let $U^\sharp \in N_u$, $V^\sharp \in N_v$ be two numerical environments, let $X^\sharp \in N_{u\cup v}$, let C be a sequence of numerical constraints over $u \cup v$, let $\mathfrak{c} = u \cap v$ we define:*

$$
\begin{aligned}
U^\sharp \boxplus V^\sharp = \ & \textbf{let } X^\sharp = (U^\sharp_{|\mathfrak{c}} \sqcup^{\mathfrak{c}}_0 V^\sharp_{|\mathfrak{c}})_{|u\cup v} \textbf{ in} \\
& \textbf{let } C = \mathbf{constraints}(U^\sharp) \cup \mathbf{constraints}(V^\sharp) \textbf{ in} \\
& \mathbf{strengthening}(X^\sharp, C, U^\sharp, V^\sharp)
\end{aligned}
$$

Remark 5.
- The precision of $\boxplus$ depends upon the order of iteration over constraints $c \in C$ in Algorithm 1. Our implementation currently iterates in the order in which constraints are returned from the abstract domains. More clever heuristics will be considered in future work.
- $U^\sharp \boxplus V^\sharp$ starts by performing the join over the domain $\mathfrak{c}$, the result is then strengthened. Other $\mathbf{strenghtening}(X^\sharp, U^\sharp \in N_u, V^\sharp \in N_v)$ operator could be defined, however in order to ensure soundness of $\boxplus$, it must satisfy the following constraints: $U^\sharp \sqsubseteq^u_0 \mathbf{strenghtening}(X^\sharp, U^\sharp, V^\sharp)$ and $V^\sharp \sqsubseteq^v_0 \mathbf{strenghtening}(X^\sharp, U^\sharp, V^\sharp)$.

Example 9. Let us now consider the example introduced thereinbefore $U^\sharp \boxplus V^\sharp = \{x = y, y = z\} \in N_{\{x,y,z\}}$. Indeed using the notations of Definition 11: $Z^\sharp \triangleq X^\sharp = \top \in N_{\{x,y,z\}}$, $C = \{x = y, y = z\}$, moreover $[\![\texttt{Assume}(x = y)]\!]^{\sharp,u\cup v}_0(\top) =$

$\{x = y\}(\triangleq T^\sharp)$, $U^\sharp \sqsubseteq_0^{\{x,y\}} \{x = y\} = T^\sharp_{|\{x,y\}}$ and $V^\sharp \sqsubseteq_0^{\{x,z\}} \top = T^\sharp_{|\{x,z\}}$. Therefore constraint $x = y$ is added to $Z^\sharp$. At the next loop iteration: $[\![\texttt{Assume}(x = z)]\!]_0^{\sharp,u\cup v}(\{x = y\}) = \{x = y, x = z\}(\triangleq T^\sharp)$, $U^\sharp \sqsubseteq_0^{\{x,y\}} \{x = y\} = T^\sharp_{|\{x,y\}}$ and $V^\sharp \sqsubseteq_0^{\{x,z\}} \{x = z\} = T^\sharp_{|\{x,z\}}$. Therefore constraint $x = z$ is added to $Z^\sharp$.

Proposition 6 (Soundness of $\boxplus$). *let $U^\sharp \in N_u$ and $V^\sharp \in N_v$, then $\gamma_0^u(U^\sharp) \subseteq (\gamma_0^{u\cup v}(U^\sharp \boxplus V^\sharp))_{|u}$ and $\gamma_0^v(V^\sharp) \subseteq (\gamma_0^{u\cup v}(U^\sharp \boxplus V^\sharp))_{|v}$.*

Definition 12 (Union abstract operators). *We define the following abstract set operator: $\langle N^\sharp, l, u\rangle \sqcup \langle N^{\sharp\prime}, l', u'\rangle \triangleq \langle N^\sharp \boxplus N^{\sharp\prime}, l \cap l', u \cup u'\rangle$. This operator soundly abstracts the union. Moreover in order to ensure the stabilization of infinitely increasing chains in $\mathfrak{M}^\sharp$ we define the following widening operator:*

$$\langle N^\sharp, l, u\rangle \triangledown \langle N^{\sharp\prime}, l', u'\rangle = \begin{cases} \langle N^\sharp \triangledown_0^u N^{\sharp\prime}_{|u}, l, u\rangle & \text{when } l \subseteq l' \wedge u' \subseteq u \\ \langle N^\sharp \boxplus N^{\sharp\prime}, l', u\rangle & \text{when } l' \subset l \wedge u' \subseteq u \\ \top & \text{otherwise} \end{cases}$$

Remark 6. This widening operator over-approximates to $\top$ whenever the upper-bound on the definition set is growing. This yields a huge loss of information however this numerical domain is designed as a tool domain used by a higher level abstraction in charge of stabilizing the environment before applying the widening, so that this case will not be used in practice.

Subsequent tree abstractions require the definition of the following operators:

- $\langle N^\sharp, l, u\rangle_{|-x} \triangleq \langle N^\sharp_{|u\setminus\{x\}}, l \setminus \{x\}, u \setminus \{x\}\rangle$ and $\langle N^\sharp, l, u\rangle_{|+x} \triangleq \langle N^\sharp_{|u\cup\{x\}}, l \cup \{x\}, u \cup \{x\}\rangle$ which respectively removes (adds) a variable to the numerical environment.
- $\langle N^\sharp, l, u\rangle_{|\mathcal{S}}$ is computed by adding variables in $\mathcal{S}$ and not in u and removing variables in u that are not in $\mathcal{S}$.

4.2 Representation of Maps over Potentially Unbounded Sets

In this subsection we focus on the problem of defining abstract numerical environments on potentially infinite environments. A classical method we use here is variable summarization (see [13]). This is based on the folding of several concrete objects (a potentially infinite number) to an abstract element which summarizes all concrete objects. The folding is encoded in a function f mapping summarized variables to the set of concrete variables they abstract. Given an abstract numerical environment $N^\sharp$ and a mapping from summary variables: $\mathcal{V}'$ to sets of concrete variables $f \in \mathcal{V}' \to \wp(\mathcal{V})$ where $f(v_1) \cap f(v_2) \neq \emptyset \Rightarrow v_1 = v_2$, we define the collapsing of a partial map $\rho \in \mathcal{V} \nrightarrow \mathbb{Z}$ under a summarizing function f:

$$\downarrow_f(\rho) = \{\rho' \in \mathcal{V}' \nrightarrow \mathbb{Z} \mid \forall v' \in \mathcal{V}',\ (f(v') \cap \mathbf{def}(\rho) = \emptyset \wedge \rho'(v') = \mathbf{undefined}) \\ \vee (\exists v \in \mathcal{V},\ v \in f(v') \cap \mathbf{def}(\rho) \wedge \rho'(v') = \rho(v))\}$$

Example 10. Consider $\mathcal{V}' = \{x, y, z, t\}$ and $\mathcal{V} = \{a, b, c, d, g, h\}$, the environment $\rho = (a \mapsto 0, b \mapsto 1, c \mapsto 2, d \mapsto 3)$ and finally the summarizing function $f = (x \mapsto \{a\}, y \mapsto \{b, c\}, z \mapsto \{d\}, t \mapsto \{g\})$. Collapsing environment ρ under f yields the set of environments: $(x \mapsto 0, y \mapsto 1, z \mapsto 3)$ and $(x \mapsto 0, y \mapsto 2, z \mapsto 3)$.

Given a summarizing function f we can now define an extension of the concretization function γ of the previous subsection in the following manner:

$$\gamma[f](N^\sharp) = \{\rho \in \mathcal{V} \nrightarrow \mathbb{Z} \mid \downarrow_f (\rho) \subseteq \gamma(N^\sharp)\}$$

Example 11. Going back to Example 10 and considering the numerical abstract element: $N^\sharp = \langle\{x \leq y\}, \{x\}, \{x, y\}\rangle$, we have: $\gamma(N^\sharp) = \{(x \mapsto \alpha) \mid \alpha \in \mathbb{Z}\} \cup \{(x \mapsto \alpha, y \mapsto \beta) \mid \alpha \leq \beta\}$. We have: $m \in \gamma[f](N^\sharp) \Leftrightarrow \downarrow_f (m) \subseteq \gamma(N^\sharp) \Rightarrow \{x\} \subseteq \mathbf{def}(\downarrow_f (m)) \subseteq \{x, y\}$. Therefore if we assume m defined on d then $f(z) \cap \mathbf{def}(m) \neq \emptyset$ hence there would be an element in $\downarrow_f (m)$ defined on z. Hence m is not defined on d, similarly for g. Moreover $\{x\} \subseteq \mathbf{def}(\downarrow_f (m))$ implies that m is defined on a. Finally: defining $S = \{(a \mapsto \alpha) \mid \alpha \in \mathbb{Z}\} \cup \{(a \mapsto \alpha, b \mapsto \beta) \mid \alpha \leq \beta\} \cup \{(a \mapsto \alpha, c \mapsto \beta) \mid \alpha \leq \beta\} \cup \{(a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma) \mid \alpha \leq \beta \wedge \alpha \leq \gamma\}$. We have: $\gamma[f](N^\sharp) = S \cup (\bigcup_{f \in S}\{f \uplus (h \mapsto \delta) \mid \delta \in \mathbb{Z}\})$.

The abstract domains we will define in the following sections will employ this summarization framework. The manipulation of summarized variables requires the definition of a $\mathbf{fold}(E, x, \mathcal{S})$ (resp. $\mathbf{expand}(E, x, \mathcal{S})$) operator yielding a new environment where x is used as a summary variable for $\mathcal{S}$ (resp. where a summary variable x is desummarized into a set of variables $\mathcal{S}$). Let $\mathcal{S}$ and $\mathcal{S}'$ be two finite sets of elements such that $\mathcal{S}' \cap \mathcal{S} \subseteq \{x\}$, we define: $\mathbf{expand}_0(N^\sharp, x, \mathcal{S}'') = \prod_{v \in \mathcal{S}''} N^\sharp[x \mapsto v]_{|(\mathcal{S} \setminus \{x\}) \cup \mathcal{S}''}$ and $\mathbf{fold}_0(N^\sharp, x, \mathcal{S}'') = \bigsqcup_{v \in \mathcal{S}''} N^\sharp[v \mapsto x]_{|(\mathcal{S} \setminus \mathcal{S}'') \cup \{x\}}$ (which generalize the one introduced in [13]). These operations are lifted as operators on elements of $\mathfrak{M}^\sharp$:

$$\mathbf{expand}(\langle N^\sharp, l, u\rangle, x, \mathcal{S}) \triangleq \langle \mathbf{expand}_0(N^\sharp, x, \mathcal{S}), l \setminus \{x\}, (u \setminus \{x\}) \cup \mathcal{S}\rangle$$

$$\mathbf{fold}(\langle N^\sharp, l, u\rangle, x, \mathcal{S}) \triangleq \langle \mathbf{fold}_0(N^\sharp, x, \mathcal{S}), \begin{cases} (l \setminus \mathcal{S}) \cup \{x\} & \text{if } \mathcal{S} \subseteq l \\ (l \setminus \mathcal{S}) & \text{otherwise} \end{cases}, (u \setminus \mathcal{S}) \cup \{x\}\rangle$$

5 Natural Term Abstraction by Numerical Constraints

We are now able to represent sets of maps with heterogeneous supports and to lift their concretization (modulo a summarization function) to sets of maps with infinite and heterogeneous supports. Given a tree shape (in the sense of Sect. 3), we can associate a numeric variable to each numeric leaf, and use a numeric abstract element to represent the possible values of these leaves. We will name the variable of each leaf as the path from the root to the leaf, i.e., $\mathcal{V}$ is a set of words in $\{0, ..., n-1\}$ where n is the maximum arity of the considered ranked alphabet. In order to avoid confusion such paths will be denoted $\wr 0, 1, 1 \wr$ for the word $(0, 1, 1)$. A summarized variable then represents a set of such paths. We will abstract such sets as regular expressions. Using the summarization extended

to heterogeneous supports presented in the previous section, it will be possible to represent, using a single numeric abstract element, a set of contraints over the numeric leaves of an infinite set of unbounded trees of arbitrary shape.

5.1 Hole Positions and Numerical Constraints

The presentation of our computable abstraction able to represent numerical values in trees is broken down (for presentation purposes) into two consecutive abstractions. The first one is not computable, as natural terms are abstracted as partial environments over tree paths to numerical values. This abstraction looses most of the tree shapes but focuses on their numerical environment. A second abstraction will show how partial environments over paths are abstracted into numerical abstract elements defined over a regular expression environment.

In the following, when $\mathcal{R}$ is a ranked alphabet of maximum arity n, we call *words* sequences of integers, $w = (w_0, \ldots, w_{p-1}) \in \{0, \ldots, (n-1)\}^p$ will be called a word of length p (denoted $|w|$), w_i denotes the i-th integer of the sequence, $\overline{w} = (w_1, \ldots, w_{p-1})$ is the tail of word w, $\mathcal{W}(\mathcal{R}) = \{0, \ldots, (n-1)\}^\star$ is the set of all words over $\{0, \ldots, n-1\}$ of arbitrary size.

Definition 13 (Position in a term). *Given a natural term t and a word w we inductively define the subterm of t at position w (denoted $t_{|w}$) to be:*

$$t_{|w} = \begin{cases} (t_{w_0})_{|\overline{w}} & \text{when } |w| > 0 \wedge t = f(t_0, \ldots, t_{p-1}) \text{ with } w_0 < p \\ t & \text{when } |w| = 0 \\ \textbf{undefined} & \text{otherwise} \end{cases}$$

Moreover we denote by $\textbf{numeric}(t) = \{w \in \mathbb{N}^\star \mid t_{|w} \in \mathbb{Z}\}$.

Definition 14 (Positioning lattice with exact numerical constraints). *We define $\mathcal{C}(\mathcal{R}) \triangleq \wp(\mathcal{W}(\mathcal{R}) \nrightarrow \mathbb{Z})$, an element of $\mathcal{C}(\mathcal{R})$ is therefore a set of partial maps that are acceptable bindings of positions to integers.*

Proposition 7 (Galois connection with natural terms). *When t is a natural term, $t_{\mathbb{Z}}$ is the partial map: $\lambda_{|\textbf{numeric}(t)} w.t_w$. We have the following Galois connection: $(\wp(T_{\mathbb{Z}}(\mathcal{R})), \subseteq) \underset{\alpha_{\mathcal{C}(\mathcal{R})}}{\overset{\gamma_{\mathcal{C}(\mathcal{R})}}{\leftrightarrows}} (\mathcal{C}(\mathcal{R}), \subseteq)$, with:*

$$\gamma_{\mathcal{C}(\mathcal{R})}(\Gamma) = \{t \in T_{\mathbb{Z}}(\mathcal{R}) \mid t_{\mathbb{Z}} \in \Gamma\} \quad \alpha_{\mathcal{C}(\mathcal{R})}(\mathcal{T}) = \{t_{\mathbb{Z}} \mid t \in \mathcal{T}\}$$

Example 12. Consider our running example (introduced in Example 2), $V = \{+(x, +(z, y)) \mid x \leq y \wedge z \leq y\}$, we have $\alpha_{\mathcal{C}(\mathcal{R})}(V) = \{\wr 0 \wr \mapsto \alpha, \wr 1, 0 \wr \mapsto \gamma, \wr 1, 1 \wr \mapsto \beta \mid \alpha \leq \beta \wedge \gamma \leq \beta\}$. The concretization of which is exactly V.

Example 13. Consider however the ranked alphabet $\{f(2), g(2), a(0)\}$, and the tree a. Its abstraction contains only the empty map, the concretization of which is the set of all terms that do not contain any numerical value. For example: $f(g(a, a), a), g(a, a), \ldots$. This emphasizes that we loose information on:

– the labels in the natural terms: we only have the path from the root of the term to leaves with numerical labels, not the actual symbols along the path.
– the shape of the natural terms: we do not keep any information on subterms that do not contain numerical values.

Now that we have abstracted away the shape of the terms, we are left with numerical environments with potentially infinite dimensions (that are words over the alphabet $\{0, \ldots, n-1\}$) and different definition sets. Therefore following the idea of Sect. 4 we want to define a summarization for sets of words over the alphabet $\{0, \ldots, n-1\}$. A summarization of such a language can be expressed as a partition into sub-languages. The set of regular languages over the alphabet $\{0, \ldots, n-1\}$ is a subset of the set of languages over this alphabet, that is closed under common set operations. Hence given a set $\{r_1, \ldots, r_m\}$ of regular expressions (with respective recognized language $\{L_1, \ldots, L_m\}$), we summarize all words in L_i inside a common variable r_i and therefore $\uparrow \{r_1, \ldots, r_m\}$ denotes the summarization function: $\lambda r_i.L_i$. In the following, Reg_n denotes the set of regular expressions over the alphabet $A_n = \{0, \ldots, n-1\}$. As for tree regular expressions, $(\mathrm{Reg}_n, \subset, \cap, \cup, .^c, \emptyset, A_n^\star)$ is a (non complete) complemented lattice of infinite height, upon which we can define a widening operator $\triangledown$ (see [10]) in a similar manner as for tree regular expressions (this widening is also parameterized by an integer constant). We recall moreover that operators $\subset, \cap, \cup$ and complementation $(.^c)$ are computable, and that every finite set of words is regular. Moreover we have the following representation: $(A_n^\star, \sqsubseteq) \xleftarrow{\gamma_{\mathrm{Reg}_n} = Id} (\mathrm{Reg}_n, \sqsubseteq)$. Finally in order to disambiguate regular expressions from integers we will typeset them within $\lfloor . \rfloor$ in a bold font as in: $\lfloor \mathbf{0} + \mathbf{0}.\mathbf{1}^\star \rfloor$.

Example 14. Using notations from Sect. 4.2, $\mathcal{V}' = \mathrm{Reg}_n$ and $\mathcal{V} = \mathcal{W}(\mathcal{R})$. Consider our running example (introduced in Example 2), natural terms from $V = \{+(x, +(z, y)) \mid x \leq y \wedge z \leq y\}$ contain three paths to numerical values: $\wr 0 \wr$, $\wr 1, 0 \wr$ and $\wr 1, 1 \wr$. Numerical constraints on $\wr 0 \wr$ and $\wr 1, 0 \wr$ are similar, therefore the two paths are summarized into one regular expression: $\lfloor \mathbf{0} + \mathbf{1.0} \rfloor$, $\wr 1, 1 \wr$ is left alone in its regular expression: $\lfloor \mathbf{1.1} \rfloor$. The two constraints $x \leq y \wedge z \leq y$ can now be expressed as one: $\lfloor \mathbf{0} + \mathbf{1.0} \rfloor \leq \lfloor \mathbf{1.1} \rfloor$.

In Example 14, we saw that tree paths with similar numerical constraints can be summarized in one regular expression. However, for precision purposes, we do not want to summarize all tree paths into one regular expression. Hence, we will keep several disjoint regular expressions, which we call a subpartitioning.

Definition 15 (Subpartitioning). *Given a regular expression s, a subpartitioning of s is a set $\{s_1, \ldots, s_n\}$ of regular expressions such that $\forall i \neq j$, $s_i \cap s_j = \emptyset$ and $\bigcup_{i=1}^n s_i \subseteq s$. We note $P(s)$ the set of all subpartitioning of s. Moreover if $S = \{s_1, \ldots, s_n\}$ is a set of regular expressions, $[S]_\emptyset = S \setminus \{\emptyset\}$.*

Remark 7. Contrary to a partitioning of s, we do not require that the set of partitions covers s. Indeed when a set of tree paths is unconstrained we can just remove it from the partitioning, therefore no dimension in the numerical abstract environment will be allocated for this path.

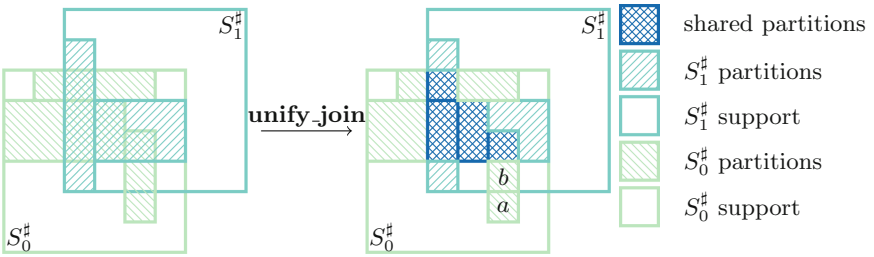


Fig. 6. Unification operator

Definition 16 (Positioning lattice with numerical abstraction). *Given a ranked alphabet $\mathcal{R}$, where the maximum arity of symbols is n, we define $\mathcal{C}^\sharp(\mathcal{R}) = \{\langle s, \mathfrak{p}, R^\sharp\rangle \mid s \in Reg_n, \mathfrak{p} \in P(s), R^\sharp \in \mathfrak{M}^\sharp_\mathfrak{p}\}$. Therefore $\mathcal{C}^\sharp(\mathcal{R})$ are triples containing:*

- *s: (called support) a regular expression coding for positions at which numerical values can be located.*
- *$\mathfrak{p}$: a subpartitioning of s. Elements of the same partition are subject to the same numerical constraints. Note that these partitions are regular.*
- *$R^\sharp$: an abstract numeric element where a dimension is associated to each partition, this dimension plays the role of a summary dimension.*

Remark 8. In the following, numerical abstract elements described in the form $\{c\}$, where c is a set of constraints, refer to $\langle c, \mathbf{vars}(c), \mathbf{vars}(c)\rangle \in \mathfrak{M}^\sharp$.

Algorithm 2. unify_join operator

Input : $\langle s, \{p_1, \ldots, p_n\}, R^\sharp\rangle, \langle s', \{p'_1, \ldots, p'_m\}, R^{\sharp\prime}\rangle$ two abstract elements
Output: two unified abstract elements

$(\underline{c_{i,j}})_{i\leq n, j\leq m} \leftarrow p_i \cap p'_j$;
$(\underline{p_i})_{i\leq n} \leftarrow p_i \cap s'^c$;
$(\underline{p'_j})_{j\leq m} \leftarrow p'_j \cap s^c$;
$(\underline{q_i})_{i\leq n} \leftarrow p_i \cap s' \cap (\cup_{j\leq m}\underline{c_{i,j}})^c$;
$(\underline{q'_j})_{j\leq m} \leftarrow p'_j \cap s \cap (\cup_{i\leq n}\underline{c_{i,j}})^c$;
$\underline{R^\sharp} \leftarrow R^\sharp$;
$\underline{R^{\sharp\prime}} \leftarrow R^{\sharp\prime}$;
for $i = 1$ **to** n **do**
  $R^\sharp \leftarrow$ **expand**$(R^\sharp, p_i, [\{c_{i,j}\}_{j\leq m} \cup \{\underline{p_i}\} \cup \{\underline{q_i}\}]_\emptyset)$;
for $j = 1$ **to** m **do**
  $R^{\sharp\prime} \leftarrow$ **expand**$(R^{\sharp\prime}, p'_j, [\{c_{i,j}\}_{i\leq n} \cup \{\underline{p'_j}\} \cup \{\underline{q'_j}\}]_\emptyset)$;
return $\langle s, \bigcup_{i\leq n, j\leq m}[\{\underline{q_i}, \underline{p_i}, \underline{c_{i,j}}\}]_\emptyset, \underline{R^\sharp}\rangle, \langle s', \bigcup_{i\leq n, j\leq m}[\{\underline{q'_i}, \underline{p'_j}, \underline{c_{i,j}}\}]_\emptyset, \underline{R^{\sharp\prime}}\rangle$;

Unification. The previous definition shows that two elements $U^\sharp = \langle s, \mathfrak{p}, R^\sharp \rangle$ and $V^\sharp = \langle s', \mathfrak{p}', R^{\sharp\prime} \rangle$ can have different subpartitionings ($\mathfrak{p}$ and $\mathfrak{p}'$). However the partitions in $\mathfrak{p}$ and in $\mathfrak{p}'$ might overlap, thus giving constraints to similar tree paths. Therefore in order to define the classical operators: $\sqsubseteq, \sqcup$ and $\triangledown$, we need to unify the two abstract elements ($U^\sharp$ and $V^\sharp$) so that given a tree path and the partition in which it is contained in $U^\sharp$, it is contained in the same partition in $V^\sharp$. This will enable us to rely on abstract operators on the numerical domain. In order to perform unification, we rely on the **expand** and **fold** operators. Indeed consider our running example, $U^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1} \rfloor, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} \rfloor\}, \{\lfloor \mathbf{0} \rfloor \leq \lfloor \mathbf{1} \rfloor\} \rangle$ and $V^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1}.(\mathbf{0}+\mathbf{1}) \rfloor, \{\lfloor \mathbf{0}+\mathbf{1.0} \rfloor, \lfloor \mathbf{1.1} \rfloor\}, \{\lfloor \mathbf{0}+\mathbf{1.0} \rfloor \leq \lfloor \mathbf{1.1} \rfloor\} \rangle$. We see that constraints on tree path $\wr 0 \wr$ is given: in $U^\sharp$ by partition $\lfloor \mathbf{0} \rfloor$ and in $V^\sharp$ by partition $\lfloor \mathbf{0}+\mathbf{1.0} \rfloor$. However we can split the partition $\lfloor \mathbf{0}+\mathbf{1.0} \rfloor$ into two partitions: $\lfloor \mathbf{0} \rfloor$ and $\lfloor \mathbf{1.0} \rfloor$, and expand variable $\lfloor \mathbf{0}+\mathbf{1.0} \rfloor$ into the two variables $\lfloor \mathbf{0} \rfloor$ and $\lfloor \mathbf{1.0} \rfloor$ in the numeric component: $\mathbf{expand}(\{\lfloor \mathbf{0}+\mathbf{1.0} \rfloor \leq \lfloor \mathbf{1.1} \rfloor\}, \lfloor \mathbf{0}+\mathbf{1.0} \rfloor, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1.0} \rfloor\}) = \{\lfloor \mathbf{0} \rfloor \leq \lfloor \mathbf{1.1} \rfloor, \lfloor \mathbf{1.0} \rfloor \leq \lfloor \mathbf{1.1} \rfloor\}$. Once $U^\sharp$ and $V^\sharp$ are unified we can rely on the numerical join to soundly abstract the union. Note that splitting partitions is more precise than merging them. Indeed, consider the example where: in $U^\sharp$ we have $\lfloor \mathbf{0} \rfloor \geq 0$ and $\lfloor \mathbf{1} \rfloor \leq 0$ and in $V^\sharp$ we have $\lfloor \mathbf{0}+\mathbf{1} \rfloor = 0$. Splitting partition in $V^\sharp$ yields: $\lfloor \mathbf{0} \rfloor = 0, \lfloor \mathbf{1} \rfloor = 0$, after joining we get $\lfloor \mathbf{0} \rfloor \geq 0, \lfloor \mathbf{1} \rfloor \leq 0$. Whereas merging partitions in $U^\sharp$ yields $\lfloor \mathbf{0}+\mathbf{1} \rfloor$ unconstrained, after joining we also get that $\lfloor \mathbf{0}+\mathbf{1} \rfloor$ is unconstrained. However unifying by splitting or merging partitions in both abstract elements might result in an over-approximation of the initial elements. This does not pose a threat to the soundness of the join operator, but it does for the inclusion test. Unifying by splitting partitions induces an increase in the number of partitions which we want to avoid when trying to stabilize abstract elements in the widening. Hence, we define three unification operators:

- An operator **unify_join** that splits partitions from $U^\sharp$ and $V^\sharp$, this operator might induce an over-approximation for both $U^\sharp$ and $V^\sharp$ and is used in the join operation. This operator is presented in Algorithm 2, and illustrated in Fig. 6.
- An operator **unify_subset** that does not modify $V^\sharp$ (in order to avoid over-approximated it), we only split and merge (using the **fold** operator) partitions from $U^\sharp$ as, if the over-approximated $U^\sharp$ is smaller than $V^\sharp$, then so is the original $U^\sharp$.
- An operator **unify_widen** that unifies $U^\sharp$ and $V^\sharp$ by only merging partitions so that the number of partitions does not increase. This operator is used in the widening definition.

Operators **unify_subset** and **unify_widen** are very similar to **unify_join**.

Definition 17 (Comparison $\sqsubseteq_{\mathcal{C}^\sharp(\mathcal{R})}$). *Using* **unify_subset** *we define a relation on* $\mathcal{C}^\sharp(\mathcal{R})$*:* $\sqsubseteq_{\mathcal{C}^\sharp(\mathcal{R})} = \{(U^\sharp, V^\sharp) \mid (\langle s, \mathfrak{p}, N^\sharp \rangle, \langle s', \mathfrak{p}', N^{\sharp\prime} \rangle) = \mathbf{unify\_subset}(U^\sharp, V^\sharp) \Rightarrow s \subseteq s' \wedge \forall b \in \mathfrak{p}', (b \subseteq s^c \vee \exists! a \in \mathfrak{p},\, b \cap s = a) \wedge N^\sharp \sqsubseteq N^{\sharp\prime}[\phi]\}$ *where* ϕ *is the renaming from* $\mathfrak{p}'$ *into* $\mathfrak{p}$ *that renames* b *to* a *when such an* a *exists.*

Example 15. Going back to our running example: $U^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1} \rfloor, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} \rfloor\}, \{\lfloor \mathbf{0} \rfloor \leq \lfloor \mathbf{1} \rfloor\} (= A^\sharp) \rangle$ and $V^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1}.(\mathbf{0}+\mathbf{1}) \rfloor, \{\lfloor \mathbf{0}+\mathbf{1.0} \rfloor, \lfloor \mathbf{1.1} \rfloor\}, \{\lfloor \mathbf{0}+\mathbf{1.0} \rfloor \leq \lfloor \mathbf{1.1} \rfloor\} \rangle$. We have $s \not\subseteq s'$ hence $U^\sharp \not\sqsubseteq V^\sharp$. However if we now consider $W^\sharp$: $\langle \lfloor (\epsilon+\mathbf{1}).(\mathbf{0}+\mathbf{1}) \rfloor, \{\lfloor (\epsilon+\mathbf{1}).\mathbf{0} \rfloor, \lfloor (\epsilon+\mathbf{1}).\mathbf{1} \rfloor\}, \{\lfloor (\epsilon+\mathbf{1}).\mathbf{0} \rfloor \leq \lfloor (\epsilon+\mathbf{1}).\mathbf{1} \rfloor\} (= B^\sharp) \rangle$. $W^\sharp$ is already unified with $U^\sharp$, we have $s \subseteq s'$ and $\phi : (\lfloor (\epsilon+\mathbf{1}).\mathbf{0} \rfloor \mapsto \mathbf{0}, \lfloor (\epsilon+\mathbf{1}).\mathbf{1} \rfloor \mapsto \lfloor \mathbf{1} \rfloor)$. Moreover $A^\sharp \sqsubseteq B^\sharp[\phi] = \{\lfloor \mathbf{0} \rfloor \leq \lfloor \mathbf{1} \rfloor\}$. Hence $U^\sharp \sqsubseteq W^\sharp$.

Proposition 8. *We have:* $(\mathcal{C}(\mathcal{R}), \sqsubseteq_{\mathcal{C}(\mathcal{R})}) \xleftarrow{\gamma_1} (\mathcal{C}^\sharp(\mathcal{R}), \sqsubseteq_{\mathcal{C}^\sharp(\mathcal{R})})$, *where:* $\gamma_1(\langle s, \mathfrak{p}, R^\sharp \rangle) = \{f \mid \mathbf{def}(f) \subseteq \gamma_{Reg_n}(s) \wedge f \in \gamma[\uparrow \mathfrak{p}](R^\sharp)\}$. *By composition we get:* $(\wp(T_{\mathbb{Z}}(\mathcal{R})), \subseteq) \xleftarrow{\gamma_2} (\mathcal{C}^\sharp(\mathcal{R}), \sqsubseteq_{\mathcal{C}^\sharp\mathcal{R}})$, *with* $\gamma_2 = \gamma_{\mathcal{C}(\mathcal{R})} \circ \gamma_1$.

Example 16. Going back to our running example: $V^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1}.(\mathbf{0}+\mathbf{1}) \rfloor, \{\lfloor \mathbf{0}+\mathbf{1.0} \rfloor, \lfloor \mathbf{1.1} \rfloor\}, \{\lfloor \mathbf{0}+\mathbf{1.0} \rfloor \leq \lfloor \mathbf{1.1} \rfloor\} \rangle$. We have: $\uparrow \mathfrak{p} = (\lfloor \mathbf{0}+\mathbf{1.0} \rfloor \mapsto \{\wr 0 \wr, \wr 1, 0 \wr\}, \lfloor \mathbf{1} \rfloor \mapsto \wr 1 \wr)$. Hence, $\gamma_1(V^\sharp) = \{(\wr 0 \wr \mapsto \alpha, \wr 1 \wr \mapsto \beta) \mid \alpha \leq \beta\} \cup \{(\wr 1, 0 \wr \mapsto \alpha, \wr 1 \wr \mapsto \beta) \mid \alpha \leq \beta\} \cup \{(\wr 0 \wr \mapsto \alpha, \wr 1, 0 \wr \mapsto \gamma, \wr 1 \wr \mapsto \beta) \mid \alpha \leq \beta \wedge \gamma \leq \beta\}$. The product with tree automata refines this result so that only the last set is left.

We now define the $\sqcup$ operator that relies on the **unify_join** operator of Algorithm 2. Once elements are unified we can distinguish three kinds of partitions: (1) Partitions found in both abstract elements (e.g. ❁ in Fig. 6). (2) Partitions found in only one of the two, which do not overlap over the support of the other abstract element (denoted u^o), these are outer-partitions. Information on such partitions can be soundly kept when joining two abstract elements (e.g. partition a in Fig. 6). (3) Partitions found in only one of the two, which overlap over the support of the other abstract element, these are inner-partitions. Information on such partitions can not be soundly kept when joining two abstract elements. (e.g. partition b in Fig. 6). Therefore in the following definition of the join operator, we compute (once elements are unified) the common partitions and both outer-partitions and merge them to form the resulting subpartitioning.

Definition 18 (Union abstract operator). *Given* $U^\sharp, V^\sharp \in \mathcal{C}^\sharp(\mathcal{R})$, *if* $(\langle s, \mathfrak{p}, R^\sharp \rangle, \langle s', \mathfrak{p}', R^{\sharp\prime} \rangle) = \mathbf{unify\_join}(U^\sharp, V^\sharp)$, *let* $\mathfrak{c}$ *be* $\mathfrak{p} \cup \mathfrak{p}'$, *let* u^o *(*$U^\sharp$ *outer-partition) be* $\{e \in \mathfrak{p} \mid e \subseteq s'^c\}$, *let* v^o *(*$V^\sharp$ *outer-partition) be* $\{e \in \mathfrak{p}' \mid e \subseteq s^c\}$, *we then define:*

$$U^\sharp \sqcup_{\mathcal{C}^\sharp(\mathcal{R})} V^\sharp = \langle s \cup s', \mathfrak{c} \cup u^o \cup v^o, R^\sharp_{|\mathfrak{c} \cup u^o} \sqcup R^{\sharp\prime}_{|\mathfrak{c} \cup v^o} \rangle$$

Proposition 9. *We have:* $\gamma_1(U^\sharp) \cup \gamma_1(V^\sharp) \subseteq \gamma_1(U^\sharp \sqcup_{\mathcal{C}^\sharp(\mathcal{R})} V^\sharp)$.

Example 17. Consider the two following abstract elements (this is the particular case of our running example where all numerical values are equal): $V^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1}.(\mathbf{0}+\mathbf{1}) \rfloor (= s), \{\lfloor \mathbf{0}+\mathbf{1.0} \rfloor (= a), \lfloor \mathbf{1.1} \rfloor (= b), \{a = b\}\} \rangle$, and $U^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1} \rfloor (= s'), \{\lfloor \mathbf{0} \rfloor (= c), \lfloor \mathbf{1} \rfloor (= d)\}, \{c = d\} \rangle$. Intuitively $U^\sharp$ could encode the term $(x+x)$ and $V^\sharp$ the term $(x+(x+x))$. The unification of those two elements is: $V_1^\sharp = \langle s, \{c, b, \lfloor \mathbf{1.0} \rfloor (= e)\}, R^\sharp \rangle$ where $R^\sharp = \langle \{c = b, e = b\}, \{b\}, \{c, b, e\} \rangle$ and $U_1^\sharp = U^\sharp$, moreover the common environment ($\mathfrak{c}$ in previous definition) is: $\{c\}$,

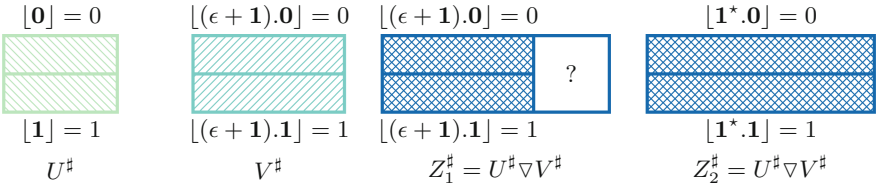


Fig. 7. Widening illustration

$V^\sharp$ outer-partitioning is $\{e, f\}$, $U^\sharp$ outer-partitioning is $\{d\}$. Hence: the numerical component resulting of the join is: $\langle\{c = d\}, \{c, d\}, \{c, d\}\rangle \sqcup \langle\{c = b, e = b\}, \{b\}, \{c, b, e\}\rangle$ which is: $\langle\{c = b, e = b, c = d\}, \emptyset, \{c, d, e, b\}\rangle$. We see here that using a naive numerical join operator, we would not have been able to get such a precise result (the numerical join would have yielded $\top$).

unify_widen $\mathcal{C}^\sharp(\mathcal{R})$ contains infinite increasing chains, therefore, we need to provide a widening operator. As for the other operators, widening is computed on unified abstract elements. A **unify_widen** operator is defined: it produces $U^\sharp$ and $V^\sharp$, over-approximations of its inputs with the same number of partitions. Moreover it ensures that each partition of $U^\sharp$ intersects exactly one partition of $V^\sharp$. This can be obtained by iterative merging partitions that overlap in both arguments until the abstract elements have the exact same partitions. Therefore from the result of **unify_widen** we can extract a list of pairs (a, b) where a is a partition from $U^\sharp$, b is a partition from $V^\sharp$ and $a \cap b \neq \emptyset$. This defines a bijection from partitions of $U^\sharp$ onto partitions of $V^\sharp$.

compose. In order to ensure stabilization we first need to stabilize the supports on which abstract elements are defined. This is easily done using the automaton widening ($s_1 \nabla s_2$ in Algorithm 3). Figure 7 illustrates the following simple example: $U^\sharp$ is an abstract element with support $\lfloor \mathbf{0} + \mathbf{1} \rfloor$, two partitions $u = \lfloor \mathbf{0} \rfloor$ and $u' = \lfloor \mathbf{1} \rfloor$, and numerical constraints $u' = 1$ and $u = 0$. $V^\sharp$ is an abstract element with support $\lfloor (\epsilon + \mathbf{1}).(\mathbf{0} + \mathbf{1}) \rfloor$, two partitions $v = \lfloor (\epsilon + \mathbf{1}).\mathbf{0} \rfloor$ and $v' = \lfloor (\epsilon + \mathbf{1}).\mathbf{1} \rfloor$ with the numerical constraints that $v = 0$ and $v' = 1$. Supports are unstable, therefore we start by widening them, which yields a new support: $\lfloor \mathbf{1}^\star.(\mathbf{0} + \mathbf{1}) \rfloor$. The unification of $U^\sharp$ and $V^\sharp$ leaves subpartitionings unchanged and yields the bijection $(u \mapsto v, u' \mapsto v')$. Given this information we now need to provide a new subpartitioning for the result of the widening. We see in this example that we could soundly use the subpartitioning from $V^\sharp$, this would produce the abstract element $Z_1^\sharp$ depicted in Fig. 7. However due to the widening of the support, paths of the form $\wr 1, 1, 1, 0 \wr$ are in the support of the result but are left unconstrained as they are not in any of the partitions. Therefore we need to use the opportunity of the extension of the support to place constraints on the newly added paths. In order to do so we would like to force the extension of the existing partitions from $U^\sharp$ and $V^\sharp$ into the new support. Therefore we need to define a **compose** operator that produces a sound new partition, given: (1) a pair a, b of partitions (such as the one produced by

Algorithm 3. widening operator

Input : $U^\sharp, V^\sharp$ two abstract elements
1 $(\langle s_1, \mathfrak{p}_1, R_1^\sharp\rangle, \langle s_2, \mathfrak{p}_2, R_2^\sharp\rangle) \leftarrow$ **unify_widen**$(U^\sharp, V^\sharp)$;
2 $s \leftarrow s_1 \triangledown s_2$;
3 $r \leftarrow s \setminus (s_1 \cup s_2)$;
4 **foreach** $a \in \mathfrak{p}_1$ **do**
5 $\quad b \leftarrow$ the unique element from $\mathfrak{p}_2$ such that $b \cap a \neq \emptyset$;
6 $\quad p \leftarrow$ **compose**(a, b, s_1, s_2, r);
7 $\quad \mathfrak{p} \leftarrow \{p\} \cup \mathfrak{p}$;
8 $\quad R_1^{\sharp\star} \leftarrow R_1^{\sharp\star}[a \mapsto p]$;
9 $\quad R_2^{\sharp\star} \leftarrow R_1^{\sharp\star}[b \mapsto p]$;
10 $\quad r \leftarrow r \setminus p$;
11 **if** $\mathfrak{p} = \mathfrak{p}_1$ **then**
12 $\quad$ **return** $\langle s, \mathfrak{p}, R_1^{\sharp\star} \triangledown R_2^{\sharp\star}\rangle$;
13 **else**
14 $\quad$ **return** $\langle s, \mathfrak{p}, R_1^{\sharp\star} \sqcup R_2^{\sharp\star}\rangle$;

unify_widen), (2) the support s_1 (resp s_2) in which a (resp. b) lives and (3) a space to occupy r. The following criteria must be verified by the resulting partition p in order to be sound and to terminate: $p \cap s_1 = a$, $p \cap s_2 = b$ and $p \setminus (s_1 \cup s_2) \subseteq r$. A variety of **compose** operators could be defined, we chose: **compose**$(a, b, s_1, s_2, r) = a \cup (b \cap (s_2 \setminus s_1)) \cup ((a \triangledown (a \cup b)) \cap r)$. The idea is the following: we keep a (as it is always sound thanks to the definition of the **unify_widen** operator), we keep the part from b that satisfies the soundness condition, and we extend into the space left to occupy according to the automata widening of a and $a \cup b$. In our example, considering the pair (u, v), this would translate as: $a = \mathbf{0}$, $b \cap (s_2 \setminus s_1) = \lfloor \mathbf{1}.\mathbf{0} \rfloor$ and $(a \triangledown (a \cup b)) \cap r = \lfloor \mathbf{0} \rfloor \triangledown \lfloor (\epsilon+1).\mathbf{0} \rfloor \cap \lfloor \mathbf{1}^{\geq 2}(\mathbf{0}+\mathbf{1}) \rfloor = \lfloor \mathbf{1}^{\geq 2}.\mathbf{0} \rfloor$. We get the new partition: $\lfloor \mathbf{1}^\star.\mathbf{0} \rfloor$. Doing the same with the pair (v, v') yields $\lfloor \mathbf{1}^\star.\mathbf{1} \rfloor$. Finally we get the abstract element $Z_2^\sharp$ from Fig. 7, which is more precise than $Z_1^\sharp$.

Definition 19 (Widening). *Algorithm 3 provides the definition of a widening operator using the* **unify_widen** *operator and parameterized by a* **compose** *function.*

Widening Stabilization. Our abstraction contains three components: (1) a support that describes the set of paths (2) a subpartitioning of this support and (3) a numerical component giving constraints on partitions in the subpartitioning. We show how the widening operator stabilizes all three components.

- Regular expression widening is used on supports when widening is called. Therefore ensuring support stabilization.
- Once supports are stable (this means $s_2 \subseteq s_1$), we have $p = a$ for every pair (a, b) of partitions. Meaning that once shapes stabilize, the only modifications

allowed on the subpartitionings are those made by the **unify_widen** operator. Each partition resulting from the operator is the union of input partitions, hence the subpartitioning will stabilize.
– Once subpartitionings are stable ($\mathfrak{p}_1 = \mathfrak{p}$ in Algorithm 3) numerical widening is applied on the numerical component in order to ensure stabilization.

Example 18 (Numerical example). Consider the simple example where: $\mathcal{R} = \{f(2)\}$, $U^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1} \rfloor, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} \rfloor\}, \{\lfloor \mathbf{1} \rfloor = \lfloor \mathbf{0} \rfloor\}\rangle$ and $V^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1} \rfloor, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} \rfloor\}, \{\lfloor \mathbf{1} \rfloor \geq \lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} \rfloor \leq \lfloor \mathbf{0} \rfloor + 1\}\rangle$. $U^\sharp$ and $V^\sharp$ have the same shape, therefore widening will be performed on the numerical component of the abstraction, therefore: $U^\sharp \triangledown V^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1} \rfloor, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} \rfloor\}, \{\lfloor \mathbf{1} \rfloor \geq \lfloor \mathbf{0} \rfloor\}\rangle$.

Reducing Dimensionality and Improving Precision. As emphasized by the previous examples, definitions and illustrations, the numerical component of an abstract state is used as a container for constraints on regular expressions, every node in a regular expression must then satisfy all numerical constraints on the underlying regular expression. Therefore when two nodes of a tree satisfy the same constraints, they should be stored in the same partition so as to reduce the dimension of the numerical domain (thus improving efficiency). Moreover the widening operator provided in Algorithm 3 relies (for precision) on the fact that partitions are built by similarity of constraints, therefore partition merging, when it does not result in an over-approximation, also leads to a precision gain. The unification operator defined in Algorithm 2 tends to split partitions whereas the widening operator defined in Algorithm 3 tends to merge them. In order to reduce dimensionality, we would like to define a $\mathbf{reduce} : \mathcal{C}^\sharp(\mathcal{R}) \to \mathcal{C}^\sharp(\mathcal{R})$ operator, that folds variables with similar constraints into one. Please note that $\forall S \cap S' \subseteq \{x\}$, $x \in S$ and $R^\sharp \in N_S$, we have that $R^\sharp \sqsubseteq_{N_S} \mathbf{expand}(\mathbf{fold}(R^\sharp, x, S'), x, S')$. This means that when variables are folded into one, expanding them afterwards would yield a bigger abstract element. For example, consider the octagon $R^\sharp = \{x \geq 2, y \geq 2, x = y\}$ then $\mathbf{fold}(R^\sharp, z, \{x, y\}) = \{z \geq 2\}(\triangleq R^{\sharp\prime})$ and $\mathbf{expand}(R^{\sharp\prime}, z, \{x, y\}) = \{x \geq 2, y \geq 2\}$. However if we consider $R^\sharp = \{x \geq 2, y \geq 2\}$ then $\mathbf{fold}(\mathbf{expand}(R^\sharp, z, \{x, y\}), z, \{x, y\}) = R^\sharp$. Therefore if we assume given a score function $\mathbf{score}(R^\sharp, x, S')$ ranging in $[0, 1]$ such that $\mathbf{score}(R^\sharp, x, S') = 1 \Leftrightarrow R^\sharp = \mathbf{expand}(\mathbf{fold}(R^\sharp, x, S'), x, S')$, we are able to define a generic **reduce** operator parameterized by a value α. This **reduce** operator merges partitions until no more set of partitions has a high enough score according to the **score** function. Finding a good **score** function is a work in progress. As a first approximation we used the following trivial one: $\mathbf{score}_0(R^\sharp, S) = 1$ when $\mathbf{expand}(\mathbf{fold}(R^\sharp, x, S), x, S) = R^\sharp$ and 0 otherwise. This $\mathbf{score}_0$ guarantees there is no loss of precision, but can miss opportunities for simplification.

Example 19. Consider the following example: $U^\sharp = \langle \lfloor \mathbf{0}+\mathbf{1} \rfloor, \{\lfloor \mathbf{0} \rfloor, \lfloor \mathbf{1} \rfloor\}, \{\lfloor \mathbf{0} \rfloor = 0, \lfloor \mathbf{1} \rfloor = 0\}\rangle$. Relations on $\lfloor \mathbf{0} \rfloor$ and $\lfloor \mathbf{1} \rfloor$ can be expressed in one relation using the summarizing variable $\lfloor \mathbf{0}+\mathbf{1} \rfloor$. This yields: $\mathbf{reduce}(U^\sharp) = \langle \lfloor \mathbf{0}+\mathbf{1} \rfloor, \{\lfloor \mathbf{0}+\mathbf{1} \rfloor\}, \{\lfloor \mathbf{0}+\mathbf{1} \rfloor = 0\}\rangle$. Note that $\mathbf{expand}(\{\lfloor \mathbf{0}+\mathbf{1} \rfloor = 0\}, \lfloor \mathbf{0}+\mathbf{1} \rfloor, \{\lfloor \mathbf{1} \rfloor, \lfloor \mathbf{0} \rfloor\}) = \{\lfloor \mathbf{0} \rfloor = 0, \lfloor \mathbf{1} \rfloor = 0\}$. Therefore no information is lost.

Abstract Semantic of Operators. As for tree automata, abstract semantic of operators defined in Sect. 2 can be defined as simple transformations on regular automata. Indeed the `make_symbolic`($s \in \mathcal{R}$) (resp. `get_son`) operator, amounts to adding (resp. removing) an integer letter to: (1) the partitions in the subpartitioning and (2) the support. `make_integer`($e \in expr$) amounts to building an abstract element with support $\lfloor \epsilon \rfloor$ and a subpartitioning containing only $\{\lfloor \epsilon \rfloor\}$, on which we put the constraint that it is equal to e. `is_symbol` needs only split the support and each partition, in the two language $L = \{\epsilon\}$ and $A_n^\star \setminus L$. Indeed in order to restrict to terms having only an integer as root, the support must be reduced to ϵ. The `get_sym_head` operator always yields the whole ranked alphabet (as this was abstracted away and will be refined by the automaton abstraction). Finally for `get_num_head`: (1) if the empty path $\wr\int$ is in the support we produce the set of integers satisfying the numerical constraints on the partition containing ϵ, and $\top$ in case no such partition could be found, and (2) otherwise we know that no numerical value is produced.

5.2 Product of Tree Automata and Numerical Constraints

The abstraction by tree automata defined in Sect. 3 and the abstraction by numerical constraints on tree paths defined in Sect. 5.1 provide non comparable information on the set of terms they abstract. Indeed the former describes precisely the shape of the term but can not express numerical constraints whereas the latter abstracts away most of the shape and focuses on numerical constraints. To benefit from both kinds of information, we use a reduced product between the two domains. Both abstractions in the product contain information on potential integer positions. The position of the $\square$ symbol in the tree automaton abstraction and the support in the numerical constraints abstractions both yield this information. We remove the support component from the product as the information can be retrieved from the tree abstraction. The definitions of the abstract operators in Sect. 5.1 require the support to be a regular language. We show in this subsection how to retrieve the support of a tree automaton with holes and that it is regular.

Given a FTA$(Q, \mathcal{R}, Q_f, \delta)$ over a ranked alphabet $\mathcal{R}$ with maximum arity n. We assume that every node in Q is reachable. Consider the following system over variables v_p for $p \in Q$ with values in the set of languages over the alphabet A_n (. designates the classical concatenation operator lifted to languages):

$$\{v_p = \bigcup_{(s,(q_1,\ldots,q_m),q)\in\delta | q_i = p} v_q.\{i\} \cup \begin{cases} \{\epsilon\} \text{ if } p \in Q_f \\ \emptyset \quad \text{otherwise} \end{cases} \mid p \in Q\}$$

Every language $\{i\}$ for $i \in \mathbb{N}$ is regular and does not contain ϵ, moreover $\emptyset$ and $\{\epsilon\}$ are regular languages. By application of Arden's rule (see [18]) and Gauss elimination we can compute the unique solution of this system, moreover every v_p is regular. Variable v_p is defined so that: $w \in v_p$ if and only if there exists a tree t recognized by the automaton such that $p \in \text{REACH}(t_{|w})$. If $\square \in \mathcal{R}$ we have that the regular language: $\cup_{(\square,(),p)\in\delta} v_p$ represents exactly the potential positions of integers in trees accepted by the tree automaton.

Height and Size. The product is enriched with a simple height and size abstraction: numerical variables (encoding heights and sizes) are added to the numerical component of the abstraction.

5.3 Environment Abstraction

In the previous section, we designed abstractions for sets of trees. However in order to be able to tackle the examples from the introductory section (Sect. 1) we need to design an abstraction able to represent maps from a set of variables to natural terms. In Sect. 3 we have shown how to lift abstractions on natural terms to abstractions of environments over a given finite set of finite term variables $\mathcal{T}$. We apply the same mechanism here to lift the product presented in Sect. 5.2. However lifting the product would result in abstract environments being maps from natural term variables to abstractions containing a numerical environment. In order to be able to express numerical relations between two sets of natural terms or even between numerical program variables and numerical values of natural terms we factor away the numerical environment so that it is shared by all natural term abstractions in the term environment and by the program variables in the numerical environment. Therefore the final abstraction is a pair $(m, R^\sharp)$ where: (1) m is a map from $\mathcal{T}$ to an abstract element that is a product of the automaton abstraction and the hole positioning abstraction. Moreover as all the numerical constraints are stored in a common numerical environment the product abstraction amounts to a pair $(\mathcal{A}, \mathfrak{p})$ where $\mathcal{A}$ is an element of the automaton abstraction and $\mathfrak{p}$ is a partitioning of its support. (2) $R^\sharp$ is an element of $\mathfrak{M}^\sharp$ binding in the same numerical element: numerical program variables and all partitions found in the mapping m.

6 Implementation and Example

6.1 Implementation

The analyzer was implemented in OCaml (∼5000 loc) in the novel and still in development Mopsa framework (see [21]). Mopsa enables a modular development of static analyzers defined by abstract interpretation. An analyzer is built by choosing abstract domains, and combining them according to the user specification. Mopsa comes with pre-existing iterators and domains (e.g. interprocedural analysis, loop iterators, numerical domains, ...), and new ones can be added (e.g. tree abstract domain). A key feature of Mopsa is the ability of an abstract domain to use the abstract knowledge it maintains to transform dynamically expressions into other expressions that can be manipulated more easily by further domains, providing a flexible way to combine relational domains. For instance, assume that a domain abstracts arrays by associating a scalar variable a_0, a_1, ..., to each element $a[0]$, $a[1]$, ..., of an array a, and delegating the abstraction of the array contents to a numeric domain for scalars. It can then evaluate $\mathbb{E}^\sharp[\![2 * a[i] + i]\!](i \mapsto [0, 1])$ into the disjunction

$(2 * a_0 + i, i \mapsto [0,0]) \vee (2 * a_1 + i, i \mapsto [1,1])$, indicating that $2 * a[i] + i$ is equivalent to $2 * a_0 + i$ in the sub-environment where $i = 0$ and to $2 * a_1 + i$ in the sub-environment where $i = 1$. Each term of the disjunction contains an array-free expression that can be handled by the scalar domain in the corresponding sub-environment. In the abstract, expressions can be evaluated by induction on the syntax into symbolic expressions to retain the full power of relational domains and disjunctive reasoning (see [21] for more details). We exploit this feature in our implementation to combine our tree abstractions. We implemented (in the Mopsa framework) libraries for regular and tree regular languages that offer the usual lattice interface enriched with a widening operator. These libraries can be reused for the definition of other abstract domains. The overall complexity of the analysis is driven by the complexity of the lattice operations in the regular and tree regular libraries. These are exponential in the number of states of the considered automata, which is bounded by the widening parameter.

6.2 Examples of Analysis

Numerical variables of the form $\mathtt{t}.x$, where $\mathtt{t}$ is a natural term variable, represent a variable allocated for tree $\mathtt{t}$. For example: $\mathtt{t}.r$ where r is a regular expression is the variable allocated for partition r in tree $\mathtt{t}$.

C Introductory Example. Let us consider the introductory example Program 4. The loop invariant inferred with our analysis is the following abstract element: $U^\sharp = (\mathtt{y} \mapsto (\mathcal{A}, \{\lfloor \mathbf{0}.(\mathbf{0}.\mathbf{0})^\star.\mathbf{1} \rfloor (= r)\}), R^\sharp)$, with $\mathcal{A} = \langle \{a,b,c,d\}, \{*(1), +(2), \square(0), (p,0)\}, \{c\}, \{*(d) \to c, +(c,a) \to d, \square() \to a, p \to c\}\rangle$, and $R^\sharp$ satisfies the constraints: $\{\mathtt{i} \geq 0, \mathtt{i} \leq \mathtt{n}, \mathtt{y}.r = 4\}$. This describes precisely the set of terms of the form: p, $*(p{+}4)$, $*(*(p{+}4){+}4)$, ... As mentioned in Sect. 6.1 evaluations of tree expressions yield pairs containing an expression and an abstract environment. Tree expressions are pairs $(\mathcal{A}, \mathfrak{p})$, partitions in $\mathfrak{p}$ are bound by the adjoined environment. Let us now present the result of the evaluation of the `make_integer(4)` expression in the abstract environment $U^\sharp$. Here we get the expression $(\mathcal{A}', \{\lfloor \epsilon \rfloor\})$ (where $\mathcal{A}'$ recognizes only $\square$) in the environment: $(\mathtt{y} \mapsto (\mathcal{A}, \{r\}), R^{\sharp\prime})$ where $R^{\sharp\prime} = R^\sharp \cup \{\lfloor \epsilon \rfloor = 4\}$. This emphasizes how the environment is used to give constraints on the adjoined expression. This transports numerical relations from the leafs of the expression up to the assigned variable $\mathtt{t}$.

OCaml Introductory Example. Let us now consider the introductory example Program 5. The inferred loop invariant is the following ($r = \lfloor (\mathbf{1}.\mathbf{1})^\star.\mathbf{0} \rfloor$ and $r' = \lfloor (\mathbf{1}.\mathbf{1})^\star.\mathbf{1}.\mathbf{0} \rfloor$): $(\mathtt{t} \mapsto (\mathcal{A}, \{r, r'\}), R^\sharp)$ and $R^\sharp$ satisfies the constraints: $\{\mathtt{t}.r' = \mathtt{x} - 1, \mathtt{t}.r = \mathtt{t}.r' + 2, i \geq 0, i \leq \mathtt{n}\}$ and $\mathcal{A} = (\{a,b,c,d\}, \{\mathtt{Cons}(2), \mathtt{Nil}(0), \square(0)\}, \{a\}, \{\mathtt{Cons}(c,a) \to d, \mathtt{Cons}(c,d) \to a, \mathtt{Nil} \to a, \square \to c\})$. Please note that at the end of the `while` loops the two numerical environments that need to be joined are not defined over the same set of variables (in the environments that have not gone through the loop, variables $\mathtt{t}.r'$ and $\mathtt{t}.r$ are not present). However thanks to the $\bowtie$ operator, we do not have to

loose the numerical relations between these variables and `x`. Hence we are able to prove that the assertion holds.

The analyzer was able to successfully analyze and infer the expected invariants for both examples.

7 Related Works

Previous works on sets of trees abstractions [20] were able to recognize larger classes of tree languages than tree automata. However we focused here on the abstraction of trees labeled with numerical values, therefore the work closest to ours would be [12]. Indeed it defines tree automata where leaves can be elements of a lattice (for example an interval). They are therefore able to represent sets of natural terms, but can not express numerical relations between the leaves of trees. Moreover they rely on a partitioning of the leaf lattice for tree automata operations. In [1] (and [2]) tree automata and regular automata are used for the model checking of programs manipulating C pointers and structures. Other uses have been made of tree automata in verification: shape analysis of C programs as in [15], computation of an over-approximation of terms computable by attackers of cryptographic protocols as in [24]. Widening regular languages by the computation of an equivalence relation of bounded index is also done in [9] and in [11]. As mentioned, variable summarization is often used to represent unbounded memory locations as in [17] or [14]. Moreover numerical abstract domains able to handle optional variables have been defined such as [19]. Finally termination analyses have been proposed for the analysis of programs manipulating tree structures (AVL, red-black trees) see [16].

8 Conclusion

In this article we presented a relational abstract environment for sets of trees over a finite algebra, with numerically labeled leaves. We emphasized the potential applications of being able to describe such trees: description of reachable memory zones, tracking symbolic equalities between program variables, description of tree like structures. In order to improve the precision of the analysis while not blowing up its cost we defined a novel abstraction for sets of maps with heterogeneous supports. This numeric abstraction is able to represent optional dimensions in numerical domains without losing relations with optional variables. All domains presented in the article were implemented as a library in the Mopsa framework.

References

1. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006). https://doi.org/10.1007/11823230_5
2. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_29
3. Bourdoncle, F.: Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite. Ph.D. thesis, Ecole polytechnique (1992)
4. Comon, H., et al.: Tree automata techniques and applications (2007). Release October, 12th 2007
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of POPL, pp. 238–252. ACM (1977)
6. Cousot, P., Cousot, R.: Static determination of dynamic properties of generalized type unions. In: Language Design for Reliable Software, pp. 77–94 (1977)
7. Cousot, P., Cousot, R.: Modular static program analysis. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 159–179. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_13
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of POPL, pp. 84–96. ACM Press (1978)
9. Feret, J.: Abstract interpretation-based static analysis of mobile ambients. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 412–430. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-47764-0_24
10. Le Gall, T.: Abstract lattices for the verification of systèmes with stacks and queues. Ph.D. thesis, University of Rennes 1, France (2008)
11. Le Gall, T., Jeannet, B., Jéron, T.: Verification of communication protocols using abstract interpretation of FIFO queues. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 204–219. Springer, Heidelberg (2006). https://doi.org/10.1007/11784180_17
12. Genet, T., Le Gall, T., Legay, A., Murat, V.: Tree regular model checking for lattice-based automata. CoRR, abs/1203.1495 (2012)
13. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 512–529. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_38
14. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: Proceedings of POPL, pp. 338–350. ACM (2005)
15. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 424–440. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_34
16. Habermehl, P., Iosif, R., Rogalewicz, A., Vojnar, T.: Proving termination of tree manipulating programs. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 145–161. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75596-8_12
17. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Proceedings of PLDI, pp. 339–348. ACM (2008)

18. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley Longman Publishing Co., Inc, Boston (2006)
19. Liu, J., Rival, X.: Abstraction of optional numerical values. In: Feng, X., Park, S. (eds.) APLAS 2015. LNCS, vol. 9458, pp. 146–166. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26529-2_9
20. Mauborgne, L.: Representation of sets of trees for abstract interpretation. Ph.D. thesis, Ecole polytechnique (1999)
21. Miné, A., Ouadjaout, A., Journault, M.: Design of a modular platform for static analysis. In: The Ninth Workshop on Tools for Automatic Program Analysis (TAPAS 2018), Fribourg-en-Brisgau, Germany, August 2018. https://hal.sorbonne-universite.fr/hal-01870001/file/mine-al-tapas18.pdf
22. Miné, A.: The octagon abstract domain. In: Proceedings of WCRE, p. 310. IEEE Computer Society (2001)
23. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 348–363. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_23
24. Monniaux, D.: Abstracting cryptographic protocols with tree automata. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 149–163. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48294-6_10
25. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of 17th IEEE (LICS 2002), pp. 55–74. IEEE Computer Society (2002)

A Static Higher-Order Dependency Pair Framework

Carsten Fuhs[1(✉)] and Cynthia Kop[2(✉)]

[1] Department of Computer Science and Information Systems, Birkbeck, University of London, London, UK
carsten@dcs.bbk.ac.uk
[2] Department of Software Science, Radboud University Nijmegen, Nijmegen, The Netherlands
c.kop@cs.ru.nl

Abstract. We revisit the static dependency pair method for proving termination of higher-order term rewriting and extend it in a number of ways: (1) We introduce a new rewrite formalism designed for general applicability in termination proving of higher-order rewriting, Algebraic Functional Systems with Meta-variables. (2) We provide a syntactically checkable soundness criterion to make the method applicable to a large class of rewrite systems. (3) We propose a modular dependency pair *framework* for this higher-order setting. (4) We introduce a fine-grained notion of *formative* and *computable* chains to render the framework more powerful. (5) We formulate several existing and new termination proving techniques in the form of processors within our framework.

The framework has been implemented in the (fully automatic) higher-order termination tool WANDA.

1 Introduction

Term rewriting [3,48] is an important area of logic, with applications in many different areas of computer science [4,11,18,23,25,36,41]. *Higher-order* term rewriting – which extends the traditional *first-order* term rewriting with higher-order types and binders as in the λ-calculus – offers a formal foundation of functional programming and a tool for equational reasoning in higher-order logic. A key question in the analysis of both first- and higher-order term rewriting is *termination*; both for its own sake, and as part of confluence and equivalence analysis.

In first-order term rewriting, a hugely effective method for proving termination (both manually and automatically) is the *dependency pair (DP) approach* [2]. This approach has been extended to the *DP framework* [20,22], a highly modular methodology which new techniques for proving termination *and non-termination* can easily be plugged into in the form of *processors*.

In higher-order rewriting, two DP approaches with distinct costs and benefits are used: *dynamic* [31,45] and *static* [6,32–34,44,46] DPs. Dynamic DPs are more broadly applicable, yet static DPs often enable more powerful analysis techniques. Still, neither approach has the modularity and extendability of

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 752–782, 2019.
https://doi.org/10.1007/978-3-030-17184-1_27

the DP framework, nor can they be used to prove non-termination. Also, these approaches consider different styles of higher-order rewriting, which means that for all results certain language features are not available.

In this paper, we address these issues for the *static* DP approach by extending it to a full higher-order *dependency pair framework* for both termination and non-termination analysis. For broad applicability, we introduce a new rewriting formalism, *AFSMs*, to capture several flavours of higher-order rewriting, including *AFSs* [26] (used in the annual Termination Competition [50]) and *pattern HRSs* [37,39] (used in the annual Confluence Competition [10]). To show the versatility and power of this methodology, we define various processors in the framework – both adaptations of existing processors from the literature and entirely new ones.

Detailed Contributions. We reformulate the results of [6,32,34,44,46] into a DP framework for AFSMs. In doing so, we instantiate the applicability restriction of [32] by a very liberal syntactic condition, and add two new flags to track properties of DP problems: one completely new, one from an earlier work by the authors for the *first-order* DP framework [16]. We give eight *processors* for reasoning in our framework: four translations of techniques from static DP approaches, three techniques from first-order or dynamic DPs, and one completely new.

This is a *foundational* paper, focused on defining a general theoretical framework for higher-order termination analysis using dependency pairs rather than questions of implementation. We have, however, implemented most of these results in the fully automatic termination analysis tool WANDA [28].

Related Work. There is a vast body of work in the first-order setting regarding the DP approach [2] and framework [20,22,24]. We have drawn from the ideas in these works for the core structure of the higher-order framework, but have added some new features of our own and adapted results to the higher-order setting.

There is no true higher-order DP *framework* yet: both static and dynamic approaches actually lie halfway between the original "DP approach" of first-order rewriting and a full DP framework as in [20,22]. Most of these works [30–32,34,46] prove "non-loopingness" or "chain-freeness" of a set $\mathcal{P}$ of DPs through a number of theorems. Yet, there is no concept of *DP problems*, and the set $\mathcal{R}$ of rules cannot be altered. They also fix assumptions on dependency chains – such as minimality [34] or being "tagged" [31] – which frustrate extendability and are more naturally dealt with in a DP framework using flags.

The static DP approach for higher-order term rewriting is discussed in, e.g., [34,44,46]. The approach is limited to *plain function passing (PFP)* systems. The definition of PFP has been made more liberal in later papers, but always concerns the position of higher-order variables in the left-hand sides of rules. These works include non-pattern HRSs [34,46], which we do not consider, but do not employ formative rules or meta-variable conditions, or consider non-termination, which we do. Importantly, they do not consider strictly positive inductive types, which could be used to significantly broaden the PFP restriction. Such types *are* considered in an early paper which defines a variation of static higher-order

dependency pairs [6] based on a computability closure [7,8]. However, this work carries different restrictions (e.g., DPs must be type-preserving and not introduce fresh variables) and considers only one analysis technique (reduction pairs).

Definitions of DP approaches for *functional programming* also exist [32,33], which consider applicative systems with ML-style polymorphism. These works also employ a much broader, semantic definition than PFP, which is actually more general than the syntactic restriction we propose here. However, like the static approaches for term rewriting, they do not truly exploit the computability [47] properties inherent in this restriction: it is only used for the initial generation of dependency pairs. In the present work, we will take advantage of our exact computability notion by introducing a `computable` flag that can be used by the computable subterm criterion processor (Theorem 63) to handle benchmark systems that would otherwise be beyond the reach of static DPs. Also in these works, formative rules, meta-variable conditions and non-termination are not considered.

Regarding *dynamic* DP approaches, a precursor of the present work is [31], which provides a halfway framework (methodology to prove "chain-freeness") for dynamic DPs, introduces a notion of formative rules, and briefly translates a basic form of static DPs to the same setting. Our formative *reductions* consider the shape of reductions rather than the rules they use, and they can be used as a flag in the framework to gain additional power in other processors. The adaptation of static DPs in [31] was very limited, and did not for instance consider strictly positive inductive types or rules of functional type.

For a more elaborate discussion of both static and dynamic DP approaches in the literature, we refer to [31] and the second author's PhD thesis [29].

Organisation of the Paper. Section 2 introduces higher-order rewriting using AFSMs and recapitulates computability. In Sect. 3 we impose restrictions on the input AFSMs for which our framework is soundly applicable. In Sect. 4 we define static DPs for AFSMs, and derive the key results on them. Section 5 formulates the DP framework and a number of DP processors for existing and new termination proving techniques. Section 6 concludes. Detailed proofs for all results in this paper and an experimental evaluation are available in a technical report [17]. In addition, many of the results have been informally published in the second author's PhD thesis [29].

2 Preliminaries

In this section, we first define our notation by introducing the AFSM formalism. Although not one of the standards of higher-order rewriting, AFSMs combine features from various forms of higher-order rewriting and can be seen as a form of IDTSs [5] which includes application. We will finish with a definition of *computability*, a technique often used for higher-order termination methods.

2.1 Higher-Order Term Rewriting Using AFSMs

Unlike first-order term rewriting, there is no single, unified approach to higher-order term rewriting, but rather a number of similar but not fully compatible systems aiming to combine term rewriting and typed λ-calculi. For generality, we will use *Algebraic Functional Systems with Meta-variables*: a formalism which admits translations from the main formats of higher-order term rewriting.

Definition 1 (Simple types). *We fix a set $\mathcal{S}$ of* sorts*. All sorts are simple types, and if σ, τ are simple types, then so is $\sigma \to \tau$.*

We let $\to$ be right-associative. Note that all types have a unique representation in the form $\sigma_1 \to \ldots \to \sigma_m \to \iota$ with $\iota \in \mathcal{S}$.

Definition 2 (Terms and meta-terms). *We fix disjoint sets $\mathcal{F}$ of* function symbols*, $\mathcal{V}$ of* variables *and $\mathcal{M}$ of* meta-variables*, each symbol equipped with a type. Each meta-variable is additionally equipped with a natural number. We assume that both $\mathcal{V}$ and $\mathcal{M}$ contain infinitely many symbols of all types. The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of* terms *over $\mathcal{F}, \mathcal{V}$ consists of expressions s where $s : \sigma$ can be derived for some type σ by the following clauses:*

(V) $x : \sigma$ *if* $x : \sigma \in \mathcal{V}$ *(@)* $s\ t : \tau$ *if* $s : \sigma \to \tau$ *and* $t : \sigma$

(F) $\mathtt{f} : \sigma$ *if* $\mathtt{f} : \sigma \in \mathcal{F}$ *(Λ)* $\lambda x.s : \sigma \to \tau$ *if* $x : \sigma \in \mathcal{V}$ *and* $s : \tau$

Meta-terms *are expressions whose type can be derived by those clauses and:*

(M) $Z\langle s_1, \ldots, s_k\rangle : \sigma_{k+1} \to \ldots \to \sigma_m \to \iota$
if $Z : (\sigma_1 \to \ldots \to \sigma_k \to \ldots \to \sigma_m \to \iota,\ k) \in \mathcal{M}$ *and* $s_1 : \sigma_1, \ldots, s_k : \sigma_k$

The λ binds variables as in the λ-calculus; unbound variables are called free*, and $FV(s)$ is the set of free variables in s. Meta-variables cannot be bound; we write $FMV(s)$ for the set of meta-variables occurring in s. A meta-term s is called* closed *if $FV(s) = \emptyset$ (even if $FMV(s) \neq \emptyset$). Meta-terms are considered modulo α-conversion. Application (@) is left-associative; abstractions (Λ) extend as far to the right as possible. A meta-term s* has type σ *if $s : \sigma$; it* has base type *if $\sigma \in \mathcal{S}$. We define $\mathsf{head}(s) = \mathsf{head}(s_1)$ if $s = s_1\ s_2$, and $\mathsf{head}(s) = s$ otherwise.*

A (meta-)term s has a sub-(meta-)term *t, notation $s \trianglerighteq t$, if either $s = t$ or $s \rhd t$, where $s \rhd t$ if (a) $s = \lambda x.s'$ and $s' \trianglerighteq t$, (b) $s = s_1\ s_2$ and $s_2 \trianglerighteq t$ or (c) $s = s_1\ s_2$ and $s_1 \trianglerighteq t$. A (meta-)term s has a* fully applied sub-(meta-)term *t, notation $s \blacktriangleright\!\!\trianglerighteq t$, if either $s = t$ or $s \blacktriangleright t$, where $s \blacktriangleright t$ if (a) $s = \lambda x.s'$ and $s' \blacktriangleright\!\!\trianglerighteq t$, (b) $s = s_1\ s_2$ and $s_2 \blacktriangleright\!\!\trianglerighteq t$ or (c) $s = s_1\ s_2$ and $s_1 \blacktriangleright t$ (so if $s = x\ s_1\ s_2$, then x and $x\ s_1$ are not fully applied subterms, but s and both s_1 and s_2 are).*

For $Z : (\sigma, k) \in \mathcal{M}$, we call k the arity *of Z, notation $arity(Z)$.*

Clearly, all fully applied subterms are subterms, but not all subterms are fully applied. Every term s has a form $t\ s_1 \cdots s_n$ with $n \geq 0$ and $t = \mathsf{head}(s)$ a variable, function symbol, or abstraction; in meta-terms t may also be a meta-variable application $F\langle s_1, \ldots, s_k\rangle$. *Terms* are the objects that we will rewrite; *meta-terms* are used to define rewrite rules. Note that all our terms (and meta-terms) are, by definition, well-typed. For rewriting, we will employ *patterns*:

Definition 3 (Patterns). *A meta-term is a* pattern *if it has one of the forms* $Z\langle x_1, \ldots, x_k\rangle$ *with all* x_i *distinct variables;* $\lambda x.\ell$ *with* $x \in \mathcal{V}$ *and* ℓ *a pattern; or a* $\ell_1 \cdots \ell_n$ *with* $a \in \mathcal{F} \cup \mathcal{V}$ *and all* ℓ_i *patterns* ($n \geq 0$).

In rewrite rules, we will use meta-variables for *matching* and variables only with *binders*. In terms, variables can occur both free and bound, and meta-variables cannot occur. Meta-variables originate in very early forms of higher-order rewriting (e.g., [1,27]), but have also been used in later formalisms (e.g., [8]). They strike a balance between matching modulo β and syntactic matching. By using meta-variables, we obtain the same expressive power as with Miller patterns [37], but do so without including a reversed β-reduction as part of matching.

Notational Conventions: We will use x, y, z for variables, X, Y, Z for meta-variables, b for symbols that could be variables or meta-variables, $\mathtt{f}, \mathtt{g}, \mathtt{h}$ or more suggestive notation for function symbols, and s, t, u, v, q, w for (meta-)terms. Types are denoted σ, τ, and ι, κ are sorts. We will regularly overload notation and write $x \in \mathcal{V}$, $\mathtt{f} \in \mathcal{F}$ or $Z \in \mathcal{M}$ without stating a type (or minimal arity). For meta-terms $Z\langle\rangle$ we will usually omit the brackets, writing just Z.

Definition 4 (Substitution). *A* meta-substitution *is a type-preserving function* γ *from variables and meta-variables to meta-terms. Let the* domain *of* γ *be given by:* $\mathtt{dom}(\gamma) = \{(x : \sigma) \in \mathcal{V} \mid \gamma(x) \neq x\} \cup \{(Z : (\sigma, k)) \in \mathcal{M} \mid \gamma(Z) \neq \lambda y_1 \ldots y_k.Z\langle y_1, \ldots, y_k\rangle\}$*; this domain is allowed to be infinite. We let* $[b_1 := s_1, \ldots, b_n := s_n]$ *denote the meta-substitution* γ *with* $\gamma(b_i) = s_i$ *and* $\gamma(z) = z$ *for* $(z : \sigma) \in \mathcal{V} \setminus \{b_1, \ldots, b_n\}$*, and* $\gamma(Z) = \lambda y_1 \ldots y_k.Z\langle y_1, \ldots, y_k\rangle$ *for* $(Z : (\sigma, k)) \in \mathcal{M} \setminus \{b_1, \ldots, b_n\}$*. We assume there are infinitely many variables* x *of all types such that (a)* $x \notin \mathtt{dom}(\gamma)$ *and (b) for all* $b \in \mathtt{dom}(\gamma)$*:* $x \notin FV(\gamma(b))$.

A substitution *is a meta-substitution mapping everything in its domain to terms. The result* $s\gamma$ *of applying a meta-substitution* γ *to a term* s *is obtained by:*

$$x\gamma = \gamma(x) \text{ if } x \in \mathcal{V} \qquad (s\ t)\gamma = (s\gamma)\ (t\gamma)$$
$$\mathtt{f}\gamma = \mathtt{f} \quad \text{if } \mathtt{f} \in \mathcal{F} \qquad (\lambda x.s)\gamma = \lambda x.(s\gamma) \quad \text{if } \gamma(x) = x \wedge x \notin \textstyle\bigcup_{y \in \mathtt{dom}(\gamma)} FV(\gamma(y))$$

For meta-terms, the result $s\gamma$ *is obtained by the clauses above and:*

$$Z\langle s_1, \ldots, s_k\rangle\gamma = \gamma(Z)\langle s_1\gamma, \ldots, s_k\gamma\rangle \quad \text{if } Z \notin \mathtt{dom}(\gamma)$$
$$Z\langle s_1, \ldots, s_k\rangle\gamma = \gamma(Z)\langle\langle s_1\gamma, \ldots, s_k\gamma\rangle\rangle \quad \text{if } Z \in \mathtt{dom}(\gamma)$$
$$(\lambda x_1 \ldots x_k.s)\langle\langle t_1, \ldots, t_k\rangle\rangle = s[x_1 := t_1, \ldots, x_k := t_k]$$
$$(\lambda x_1 \ldots x_n.s)\langle\langle t_1, \ldots, t_k\rangle\rangle = s[x_1 := t_1, \ldots, x_n := t_n]\ t_{n+1} \cdots t_k \quad \text{if } n < k \text{ and } s \text{ is not an abstraction}$$

Note that for fixed k, any term has exactly one of the two forms above ($\lambda x_1 \ldots x_n.s$ with $n < k$ and s not an abstraction, or $\lambda x_1 \ldots x_k.s$).

Essentially, applying a meta-substitution that has meta-variables in its domain combines a substitution with (possibly several) β-steps. For example, we have that: $\mathtt{deriv}\ (\lambda x.\mathtt{sin}\ (F\langle x\rangle))[F := \lambda y.\mathtt{plus}\ y\ x]$ equals $\mathtt{deriv}\ (\lambda z.\mathtt{sin}\ (\mathtt{plus}\ z\ x))$. We also have: $X\langle \mathtt{0}, \mathtt{nil}\rangle[X := \lambda x.\mathtt{map}\ (\lambda y.x)]$ equals $\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}$.

Definition 5 (Rules and rewriting). *Let $\mathcal{F}, \mathcal{V}, \mathcal{M}$ be fixed sets of function symbols, variables and meta-variables respectively. A* rule *is a pair $\ell \Rightarrow r$ of closed meta-terms of the same type such that ℓ is a pattern of the form $\mathtt{f}\ \ell_1 \cdots \ell_n$ with $\mathtt{f} \in \mathcal{F}$ and $FMV(r) \subseteq FMV(\ell)$. A set of rules $\mathcal{R}$ defines a rewrite relation $\Rightarrow_{\mathcal{R}}$ as the smallest monotonic relation on terms which includes:*
(Rule) $\quad \ell\delta \quad \Rightarrow_{\mathcal{R}} \quad r\delta \quad$ *if* $\ell \Rightarrow r \in \mathcal{R}$ *and* $\mathtt{dom}(\delta) = FMV(\ell)$
(Beta) $(\lambda x.s)\ t \ \Rightarrow_{\mathcal{R}} \ s[x := t]$
We say $s \Rightarrow_\beta t$ if $s \Rightarrow_{\mathcal{R}} t$ is derived using a (Beta) step. A term s is terminating *under $\Rightarrow_{\mathcal{R}}$ if there is no infinite reduction $s = s_0 \Rightarrow_{\mathcal{R}} s_1 \Rightarrow_{\mathcal{R}} \ldots$, is* in normal form *if there is no t such that $s \Rightarrow_{\mathcal{R}} t$, and is β*-normal *if there is no t with $s \Rightarrow_\beta t$. Note that we are allowed to reduce at any position of a term, even below a λ. The relation $\Rightarrow_{\mathcal{R}}$ is terminating if all terms over $\mathcal{F}, \mathcal{V}$ are terminating. The set $\mathcal{D} \subseteq \mathcal{F}$ of* defined symbols *consists of those $(\mathtt{f} : \sigma) \in \mathcal{F}$ such that a rule $\mathtt{f}\ \ell_1 \cdots \ell_n \Rightarrow r$ exists; all other symbols are called* constructors.

Note that $\mathcal{R}$ is allowed to be infinite, which is useful for instance to model polymorphic systems. Also, right-hand sides of rules do not have to be in β-normal form. While this is rarely used in practical examples, non-β-normal rules may arise through transformations, and we lose nothing by allowing them.

Example 6. Let $\mathcal{F} \supseteq \{\mathtt{0} : \mathtt{nat},\ \mathtt{s} : \mathtt{nat} \to \mathtt{nat},\ \mathtt{nil} : \mathtt{list}, \mathtt{cons} : \mathtt{nat} \to \mathtt{list} \to \mathtt{list},\ \mathtt{map} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{list} \to \mathtt{list}\}$ and consider the following rules $\mathcal{R}$:

$$\mathtt{map}\ (\lambda x.Z\langle x\rangle)\ \mathtt{nil} \Rightarrow \mathtt{nil}$$
$$\mathtt{map}\ (\lambda x.Z\langle x\rangle)\ (\mathtt{cons}\ H\ T) \Rightarrow \mathtt{cons}\ Z\langle H\rangle\ (\mathtt{map}\ (\lambda x.Z\langle x\rangle)\ T)$$

Then $\mathtt{map}\ (\lambda y.\mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ \mathtt{0}\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ \mathtt{0}\ \mathtt{nil}$. Note that the bound variable y does not need to occur in the body of $\lambda y.\mathtt{0}$ to match $\lambda x.Z\langle x\rangle$. However, a term like $\mathtt{map}\ \mathtt{s}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil})$ *cannot* be reduced, because $\mathtt{s}$ does not instantiate $\lambda x.Z\langle x\rangle$. We could alternatively consider the rules:

$$\mathtt{map}\ Z\ \mathtt{nil} \Rightarrow \mathtt{nil}$$
$$\mathtt{map}\ Z\ (\mathtt{cons}\ H\ T) \Rightarrow \mathtt{cons}\ (Z\ H)\ (\mathtt{map}\ Z\ T)$$

Where the system before had $(Z : (\mathtt{nat} \to \mathtt{nat}, 1)) \in \mathcal{M}$, here we assume $(Z : (\mathtt{nat} \to \mathtt{nat}, 0)) \in \mathcal{M}$. Thus, rather than meta-variable application $Z\langle H\rangle$ we use explicit application $Z\ H$. Then $\mathtt{map}\ \mathtt{s}\ (\mathtt{cons}\ \mathtt{0}\ \mathtt{nil}) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ (\mathtt{map}\ \mathtt{s}\ \mathtt{nil})$. However, we will often need explicit β-reductions; e.g., $\mathtt{map}\ (\lambda y.\mathtt{0})\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil}) \Rightarrow_{\mathcal{R}} \mathtt{cons}\ ((\lambda y.\mathtt{0})\ (\mathtt{s}\ \mathtt{0}))\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil}) \Rightarrow_\beta \mathtt{cons}\ \mathtt{0}\ (\mathtt{map}\ (\lambda y.\mathtt{0})\ \mathtt{nil})$.

Definition 7 (AFSM). *An* AFSM *is a tuple $(\mathcal{F}, \mathcal{V}, \mathcal{M}, \mathcal{R})$ of a signature and a set of rules built from meta-terms over $\mathcal{F}, \mathcal{V}, \mathcal{M}$; as types of relevant variables and meta-variables can always be derived from context, we will typically just refer to the AFSM $(\mathcal{F}, \mathcal{R})$. An AFSM implicitly defines the abstract reduction system $(\mathcal{T}(\mathcal{F}, \mathcal{V}), \Rightarrow_{\mathcal{R}})$: a set of terms and a rewrite relation on this set. An AFSM is terminating if $\Rightarrow_{\mathcal{R}}$ is terminating (on all terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$).*

Discussion: The two most common formalisms in termination analysis of higher-order rewriting are *algebraic functional systems* [26] (AFSs) and *higher-order rewriting systems* [37,39] (HRSs). AFSs are very similar to our AFSMs, but use variables for matching rather than meta-variables; this is trivially translated to the AFSM format, giving rules where all meta-variables have arity 0, like the "alternative" rules in Example 6. HRSs use matching modulo β/η, but the common restriction of *pattern HRSs* can be directly translated into AFSMs, provided terms are β-normalised after every reduction step. Even without this β-normalisation step, termination of the obtained AFSM implies termination of the original HRS; for second-order systems, termination is equivalent. AFSMs can also naturally encode CRSs [27] and several applicative systems (cf. [29, Chapter 3]).

Example 8 (Ordinal recursion). A running example is the AFSM $(\mathcal{F}, \mathcal{R})$ with $\mathcal{F} \supseteq \{\mathtt{0} : \mathtt{ord},\ \mathtt{s} : \mathtt{ord} \to \mathtt{ord}, \mathtt{lim} : (\mathtt{nat} \to \mathtt{ord}) \to \mathtt{ord},\ \mathtt{rec} : \mathtt{ord} \to \mathtt{nat} \to (\mathtt{ord} \to \mathtt{nat} \to \mathtt{nat}) \to ((\mathtt{nat} \to \mathtt{ord}) \to (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}) \to \mathtt{nat}\}$ and $\mathcal{R}$ given below. As all meta-variables have arity 0, this can be seen as an AFS.

$$\begin{aligned} \mathtt{rec}\ \mathtt{0}\ K\ F\ G &\Rightarrow K \\ \mathtt{rec}\ (\mathtt{s}\ X)\ K\ F\ G &\Rightarrow F\ X\ (\mathtt{rec}\ X\ K\ F\ G) \\ \mathtt{rec}\ (\mathtt{lim}\ H)\ K\ F\ G &\Rightarrow G\ H\ (\lambda m.\mathtt{rec}\ (H\ m)\ K\ F\ G) \end{aligned}$$

Observant readers may notice that by the given constructors, the type `nat` in Example 8 is not inhabited. However, as the given symbols are only a subset of $\mathcal{F}$, additional symbols (such as constructors for the `nat` type) may be included. The presence of additional function symbols does not affect termination of AFSMs:

Theorem 9 (Invariance of termination under signature extensions). *For an AFSM $(\mathcal{F}, \mathcal{R})$ with $\mathcal{F}$ at most countably infinite, let $\mathtt{funs}(\mathcal{R}) \subseteq \mathcal{F}$ be the set of function symbols occurring in some rule of $\mathcal{R}$. Then $(\mathcal{T}(\mathcal{F}, \mathcal{V}), \Rightarrow_{\mathcal{R}})$ is terminating if and only if $(\mathcal{T}(\mathtt{funs}(\mathcal{R}), \mathcal{V}), \Rightarrow_{\mathcal{R}})$ is terminating.*

Proof. Trivial by replacing all function symbols in $\mathcal{F} \setminus \mathtt{funs}(\mathcal{R})$ by corresponding variables of the same type. □

Therefore, we will typically only state the types of symbols occurring in the rules, but may safely assume that infinitely many symbols of all types are present (which for instance allows us to select unused constructors in some proofs).

2.2 Computability

A common technique in higher-order termination is Tait and Girard's *computability* notion [47]. There are several ways to define computability predicates; here we follow, e.g., [5,7–9] in considering *accessible meta-terms* using strictly positive inductive types. The definition presented below is adapted from these works, both to account for the altered formalism and to introduce (and obtain termination of) a relation $\Rrightarrow_C$ that we will use in the "computable subterm criterion processor" of Theorem 63 (a termination criterion that allows us to handle

systems that would otherwise be beyond the reach of static DPs). This allows for a minimal presentation that avoids the use of ordinals that would otherwise be needed to obtain $\Rightarrow\!\!\!\Rightarrow_C$ (see, e.g., [7,9]).

To define computability, we use the notion of an *RC-set*:

Definition 10. *A* set of reducibility candidates*, or* RC-set*, for a rewrite relation $\Rightarrow_{\mathcal{R}}$ of an AFSM is a set I of base-type terms s such that: every term in I is terminating under $\Rightarrow_{\mathcal{R}}$; I is closed under $\Rightarrow_{\mathcal{R}}$ (so if $s \in I$ and $s \Rightarrow_{\mathcal{R}} t$ then $t \in I$); if $s = x\ s_1 \cdots s_n$ with $x \in \mathcal{V}$ or $s = (\lambda x.u)\ s_0 \cdots s_n$ with $n \geq 0$, and for all t with $s \Rightarrow_{\mathcal{R}} t$ we have $t \in I$, then $s \in I$ (for any $u, s_0, \ldots, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$).*

We define I-computability for an RC-set I by induction on types. For $s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we say that s is I-computable if either s is of base type and $s \in I$; or $s : \sigma \to \tau$ and for all $t : \sigma$ that are I-computable, $s\ t$ is I-computable.

The traditional notion of computability is obtained by taking for I the set of all terminating base-type terms. Then, a term s is computable if and only if (a) s has base type and is terminating; or (b) $s : \sigma \to \tau$ and for all computable $t : \sigma$ the term $s\ t$ is computable. This choice is simple but, for reasoning, not ideal: we do not have a property like: "if $\mathtt{f}\ s_1 \cdots s_n$ is computable then so is each s_i". Such a property would be valuable to have for generalising termination proofs from first-order to higher-order rewriting, as it allows us to use computability where the first-order proof uses termination. While it is not possible to define a computability notion with this property alongside case (b) (as such a notion would not be well-founded), we can come *close* to this property by choosing a different set for I. To define this set, we will use the notion of *accessible arguments*, which is used for the same purpose also in the *General Schema* [8], the *Computability Path Ordering* [9], and the *Computability Closure* [7].

Definition 11 (Accessible arguments). *We fix a quasi-ordering $\succeq^{\mathcal{S}}$ on $\mathcal{S}$ with well-founded strict part $\succ^{\mathcal{S}} := \succeq^{\mathcal{S}} \setminus \preceq^{\mathcal{S}}$.*[1] *For a type $\sigma \equiv \sigma_1 \to \ldots \to \sigma_m \to \kappa$ (with $\kappa \in \mathcal{S}$) and sort ι, let $\iota \succeq^{\mathcal{S}}_{+} \sigma$ if $\iota \succeq^{\mathcal{S}} \kappa$ and $\iota \succ^{\mathcal{S}}_{-} \sigma_i$ for all i, and let $\iota \succ^{\mathcal{S}}_{-} \sigma$ if $\iota \succ^{\mathcal{S}} \kappa$ and $\iota \succeq^{\mathcal{S}}_{+} \sigma_i$ for all i.*[2]

For $\mathtt{f} : \sigma_1 \to \ldots \to \sigma_m \to \iota \in \mathcal{F}$, let $Acc(\mathtt{f}) = \{i \mid 1 \leq i \leq m \wedge \iota \succeq^{\mathcal{S}}_{+} \sigma_i\}$. For $x : \sigma_1 \to \ldots \to \sigma_m \to \iota \in \mathcal{V}$, let $Acc(x) = \{i \mid 1 \leq i \leq m \wedge \sigma_i$ has the form $\tau_1 \to \ldots \to \tau_n \to \kappa$ with $\iota \succeq^{\mathcal{S}} \kappa\}$. We write $s \trianglerighteq_{\mathtt{acc}} t$ if either $s = t$, or $s = \lambda x.s'$ and $s' \trianglerighteq_{\mathtt{acc}} t$, or $s = a\ s_1 \cdots s_n$ with $a \in \mathcal{F} \cup \mathcal{V}$ and $s_i \trianglerighteq_{\mathtt{acc}} t$ for some $i \in Acc(a)$ with $a \notin FV(s_i)$.

With this definition, we will be able to define a set C such that, roughly, s is C-computable if and only if (a) $s : \sigma \to \tau$ and $s\ t$ is C-computable for all C-computable t, or (b) s has base type, is terminating, and if $s = \mathtt{f}\ s_1 \cdots s_m$ then s_i is C-computable for all *accessible* i (see Theorem 13 below). The reason that $Acc(x)$ for $x \in \mathcal{V}$ is different is proof-technical: computability of $\lambda x.x\ s_1 \cdots s_m$

[1] Well-foundedness is immediate if $\mathcal{S}$ is finite, but we have not imposed that requirement.

[2] Here $\iota \succeq^{\mathcal{S}}_{+} \sigma$ corresponds to "ι occurs only positively in σ" in [5,8,9].

implies the computability of more arguments s_i than computability of $\mathtt{f}\ s_1 \cdots s_m$ does, since x can be instantiated by anything.

Example 12. Consider a quasi-ordering $\succeq^{\mathcal{S}}$ such that $\mathtt{ord} \succ^{\mathcal{S}} \mathtt{nat}$. In Example 8, we then have $\mathtt{ord} \succeq^{\mathcal{S}}_{+} \mathtt{nat} \to \mathtt{ord}$. Thus, $1 \in Acc(\mathtt{lim})$, which gives $\mathtt{lim}\ H \trianglerighteq_{\mathtt{acc}} H$.

Theorem 13. *Let $(\mathcal{F}, \mathcal{R})$ be an AFSM. Let $\mathtt{f}\ s_1 \cdots s_m \Rrightarrow_I s_i\ t_1 \cdots t_n$ if both sides have base type, $i \in Acc(\mathtt{f})$, and all t_j are I-computable. There is an RC-set C such that $C = \{s \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \mid s$ has base type $\wedge\ s$ is terminating under $\Rightarrow_{\mathcal{R}} \cup \Rrightarrow_C \wedge$ if $s \Rightarrow^*_{\mathcal{R}} \mathtt{f}\ s_1 \cdots s_m$ then s_i is C-computable for all $i \in Acc(\mathtt{f})\}$.*

Proof (sketch). Note that we cannot *define* C as this set, as the set relies on the notion of C-computability. However, we *can* define C as the fixpoint of a monotone function operating on RC-sets. This follows the proof in, e.g., [8,9]. □

The complete proof is available in [17, Appendix A].

3 Restrictions

The termination methodology in this paper is restricted to AFSMs that satisfy certain limitations: they must be *properly applied* (a restriction on the number of terms each function symbol is applied to) and *accessible function passing* (a restriction on the positions of variables of a functional type in the left-hand sides of rules). Both are syntactic restrictions that are easily checked by a computer (mostly; the latter requires a search for a sort ordering, but this is typically easy).

3.1 Properly Applied AFSMs

In *properly applied AFSMs*, function symbols are assigned a certain, minimal number of arguments that they must always be applied to.

Definition 14. *An AFSM $(\mathcal{F}, \mathcal{R})$ is* properly applied *if for every $\mathtt{f} \in \mathcal{D}$ there exists an integer k such that for all rules $\ell \Rightarrow r \in \mathcal{R}$: (1) if $\ell = \mathtt{f}\ \ell_1 \cdots \ell_n$ then $n = k$; and (2) if $r \blacktriangleright \mathtt{f}\ r_1 \cdots r_n$ then $n \geq k$. We denote $minar(\mathtt{f}) = k$.*

That is, every occurrence of a function symbol in the *right-hand* side of a rule has at least as many arguments as the occurrences in the *left-hand* sides of rules. This means that partially applied functions are often not allowed: an AFSM with rules such as $\mathtt{double}\ X \Rightarrow \mathtt{plus}\ X\ X$ and $\mathtt{doublelist}\ L \Rightarrow \mathtt{map}\ \mathtt{double}\ L$ is not properly applied, because `double` is applied to one argument in the left-hand side of some rule, and to zero in the right-hand side of another.

This restriction is not as severe as it may initially seem since partial applications can be replaced by λ-abstractions; e.g., the rules above can be made properly applied by replacing the second rule by: $\mathtt{doublelist}\ L \Rightarrow \mathtt{map}\ (\lambda x.\mathtt{double}\ x)\ L$. By using η-expansion, we can transform any AFSM to satisfy this restriction:

Definition 15 ($\mathcal{R}^{\uparrow}$). *Given a set of rules $\mathcal{R}$, let their* η-expansion *be given by* $\mathcal{R}^{\uparrow} = \{(\ell\ Z_1 \cdots Z_m)\uparrow^{\eta} \Rightarrow (r\ Z_1 \cdots Z_m)\uparrow^{\eta} \mid \ell \Rightarrow r \in \mathcal{R}$ *with* $r : \sigma_1 \to \ldots \to \sigma_m \to \iota$, $\iota \in \mathcal{S}$, *and* $Z_1, \ldots, Z_m$ *fresh meta-variables*$\}$, *where*

- $s\uparrow^{\eta} = \lambda x_1 \ldots x_m.\overline{s}\ (x_1\uparrow^{\eta}) \cdots (x_m\uparrow^{\eta})$ *if s is an application or element of* $\mathcal{V} \cup \mathcal{F}$, *and* $s\uparrow^{\eta} = \overline{s}$ *otherwise;*
- $\overline{\mathtt{f}} = \mathtt{f}$ *for* $\mathtt{f} \in \mathcal{F}$ *and* $\overline{x} = x$ *for* $x \in \mathcal{V}$, *while* $\overline{Z\langle s_1, \ldots, s_k\rangle} = Z\langle\overline{s_1}, \ldots, \overline{s_k}\rangle$ *and* $\overline{(\lambda x.s)} = \lambda x.(s\uparrow^{\eta})$ *and* $\overline{s_1\ s_2} = \overline{s_1}\ (s_2\uparrow^{\eta})$.

Note that $\ell\uparrow^{\eta}$ is a pattern if ℓ is. By [29, Thm. 2.16], a relation $\Rightarrow_{\mathcal{R}}$ is terminating if $\Rightarrow_{\mathcal{R}^{\uparrow}}$ is terminating, which allows us to transpose any methods to prove termination of properly applied AFSMs to all AFSMs.

However, there is a caveat: this transformation can introduce non-termination in some special cases, e.g., the terminating rule $\mathtt{f}\ X \Rightarrow \mathtt{g}\ \mathtt{f}$ with $\mathtt{f} : \mathtt{o} \to \mathtt{o}$ and $\mathtt{g} : (\mathtt{o} \to \mathtt{o}) \to \mathtt{o}$, whose η-expansion $\mathtt{f}\ X \Rightarrow \mathtt{g}\ (\lambda x.(\mathtt{f}\ x))$ is non-terminating. Thus, for a properly applied AFSM the methods in this paper apply directly. For an AFSM that is not properly applied, we can use the methods to prove *termination* (but not non-termination) by first η-expanding the rules. Of course, if this analysis leads to a *counterexample* for termination, we may still be able to verify whether this counterexample applies in the original, untransformed AFSM.

Example 16. Both AFSMs in Example 6 and the AFSM in Example 8 are properly applied.

Example 17. Consider an AFSM $(\mathcal{F}, \mathcal{R})$ with $\mathcal{F} \supseteq \{\mathtt{sin}, \mathtt{cos} : \mathtt{real} \to \mathtt{real}, \mathtt{times} : \mathtt{real} \to \mathtt{real} \to \mathtt{real},\ \mathtt{deriv} : (\mathtt{real} \to \mathtt{real}) \to \mathtt{real} \to \mathtt{real}\}$ and $\mathcal{R} = \{\mathtt{deriv}\ (\lambda x.\mathtt{sin}\ F\langle x\rangle) \Rightarrow \lambda y.\mathtt{times}\ (\mathtt{deriv}\ (\lambda x.F\langle x\rangle)\ y)\ (\mathtt{cos}\ F\langle y\rangle)\}$. Although the one rule has a functional output type ($\mathtt{real} \to \mathtt{real}$), this AFSM is properly applied, with `deriv` having always at least 1 argument. Therefore, we do not need to use $\mathcal{R}^{\uparrow}$. However, if $\mathcal{R}$ were to additionally include some rules that did not satisfy the restriction (such as the `double` and `doublelist` rules above), then η-expanding *all* rules, including this one, would be necessary. We have: $\mathcal{R}^{\uparrow} = \{\mathtt{deriv}\ (\lambda x.\mathtt{sin}\ F\langle x\rangle)\ Y \Rightarrow (\lambda y.\mathtt{times}\ (\mathtt{deriv}\ (\lambda x.F\langle x\rangle)\ y)\ (\mathtt{cos}\ F\langle y\rangle))\ Y\}$. Note that the right-hand side of the η-expanded `deriv` rule is not β-normal.

3.2 Accessible Function Passing AFSMs

In *accessible function passing* AFSMs, variables of functional type may not occur at arbitrary places in the left-hand sides of rules: their positions are restricted using the sort ordering $\succeq^{\mathcal{S}}$ and accessibility relation $\trianglerighteq_{\mathtt{acc}}$ from Definition 11.

Definition 18 (Accessible function passing). *An AFSM* $(\mathcal{F}, \mathcal{R})$ *is* accessible function passing (AFP) *if there exists a sort ordering* $\succeq^{\mathcal{S}}$ *following Definition 11 such that: for all* $\mathtt{f}\ \ell_1 \cdots \ell_n \Rightarrow r \in \mathcal{R}$ *and all* $Z \in FMV(r)$*: there are variables* $x_1, \ldots, x_k$ *and some i such that* $\ell_i \trianglerighteq_{\mathtt{acc}} Z\langle x_1, \ldots, x_k\rangle$.

The key idea of this definition is that computability of each ℓ_i implies computability of all meta-variables in r. This excludes cases like Example 20 below. Many common examples satisfy this restriction, including those we saw before:

Example 19. Both systems from Example 6 are AFP: choosing the sort ordering $\succeq^{\mathcal{S}}$ that equates `nat` and `list`, we indeed have $\texttt{cons}\ H\ T \trianglerighteq_{\texttt{acc}} H$ and $\texttt{cons}\ H\ T \trianglerighteq_{\texttt{acc}} T$ (as $Acc(\texttt{cons}) = \{1, 2\}$) and both $\lambda x.Z\langle x\rangle \trianglerighteq_{\texttt{acc}} Z\langle x\rangle$ and $Z \trianglerighteq_{\texttt{acc}} Z$. The AFSM from Example 8 is AFP because we can choose $\texttt{ord} \succ^{\mathcal{S}} \texttt{nat}$ and have $\texttt{lim}\ H \trianglerighteq_{\texttt{acc}} H$ following Example 12 (and also $\texttt{s}\ X \trianglerighteq_{\texttt{acc}} X$ and $K \trianglerighteq_{\texttt{acc}} K$, $F \trianglerighteq_{\texttt{acc}} F$, $G \trianglerighteq_{\texttt{acc}} G$). The AFSM from Example 17 is AFP, because $\lambda x.\texttt{sin}\ F\langle x\rangle \trianglerighteq_{\texttt{acc}} F\langle x\rangle$ for any $\succeq^{\mathcal{S}}$: $\lambda x.\texttt{sin}\ F\langle x\rangle \trianglerighteq_{\texttt{acc}} F\langle x\rangle$ because $\texttt{sin}\ F\langle x\rangle \trianglerighteq_{\texttt{acc}} F\langle x\rangle$ because $1 \in Acc(\texttt{sin})$.

In fact, *all* first-order AFSMs (where all fully applied sub-meta-terms of the left-hand side of a rule have base type) are AFP via the sort ordering $\succeq^{\mathcal{S}}$ that equates all sorts. Also (with the same sort ordering), an AFSM $(\mathcal{F}, \mathcal{R})$ is AFP if, for all rules $\texttt{f}\ \ell_1 \cdots \ell_k \Rightarrow r \in \mathcal{R}$ and all $1 \leq i \leq k$, we can write: $\ell_i = \lambda x_1 \ldots x_{n_i}.\ell'$ where $n_i \geq 0$ and all fully applied sub-meta-terms of ℓ' have base type.

This covers many practical systems, although for Example 8 we need a non-trivial sort ordering. Also, there are AFSMs that cannot be handled with *any* $\succeq^{\mathcal{S}}$.

Example 20 (Encoding the untyped λ-calculus). Consider an AFSM with $\mathcal{F} \supseteq \{\texttt{ap} : \texttt{o} \to \texttt{o} \to \texttt{o},\ \texttt{lm} : (\texttt{o} \to \texttt{o}) \to \texttt{o}\}$ and $\mathcal{R} = \{\texttt{ap}\ (\texttt{lm}\ F) \Rightarrow F\}$ (note that the only rule has type $\texttt{o} \to \texttt{o}$). This AFSM is not accessible function passing, because $\texttt{lm}\ F \trianglerighteq_{\texttt{acc}} F$ cannot hold for any $\succeq^{\mathcal{S}}$ (as this would require $\texttt{o} \succ^{\mathcal{S}} \texttt{o}$).

Note that this example is also not terminating. With $t = \texttt{lm}\ (\lambda x.\texttt{ap}\ x\ x)$, we get this self-loop as evidence: $\texttt{ap}\ t\ t \Rightarrow_{\mathcal{R}} (\lambda x.\texttt{ap}\ x\ x)\ t \Rightarrow_{\beta} \texttt{ap}\ t\ t$.

Intuitively: in an accessible function passing AFSM, meta-variables of a higher type may occur only in "safe" places in the left-hand sides of rules. Rules like the ones in Example 20, where a higher-order meta-variable is lifted out of a base-type term, are not admitted (unless the base type is greater than the higher type).

In the remainder of this paper, we will refer to a *properly applied, accessible function passing* AFSM as a PA-AFP AFSM.

Discussion: This definition is strictly more liberal than the notions of "plain function passing" in both [34] and [46] as adapted to AFSMs. The notion in [46] largely corresponds to AFP if $\succeq^{\mathcal{S}}$ equates all sorts, and the HRS formalism guarantees that rules are properly applied (in fact, all fully applied sub-meta-terms of both left- and right-hand sides of rules have base type). The notion in [34] is more restrictive. The current restriction of PA-AFP AFSMs lets us handle examples like ordinal recursion (Example 8) which are not covered by [34,46]. However, note that [34,46] consider a different formalism, which does take rules whose left-hand side is not a pattern into account (which we do not consider). Our restriction also quite resembles the "admissible" rules in [6] which

are defined using a pattern computability closure [5], but that work carries additional restrictions.

In later work [32,33], Kusakari extends the static DP approach to forms of polymorphic functional programming, with a very liberal restriction: the definition is parametrised with an *arbitrary* RC-set and corresponding accessibility ("safety") notion. Our AFP restriction is actually an instance of this condition (although a more liberal one than the example RC-set used in [32,33]). We have chosen a specific instance because it allows us to use dedicated techniques for the RC-set; for example, our *computable subterm criterion processor* (Theorem 63).

4 Static Higher-Order Dependency Pairs

To obtain sufficient criteria for both termination and non-termination of AFSMs, we will now transpose the definition of static dependency pairs [6,33,34,46] to AFSMs. In addition, we will add the new features of *meta-variable conditions*, *formative reductions*, and *computable chains*. Complete versions of all proof sketches in this section are available in [17, Appendix B].

Although we retain the first-order terminology of dependency *pairs*, the setting with meta-variables makes it more suitable to define DPs as *triples*.

Definition 21 ((Static) Dependency Pair). *A* dependency pair (DP) *is a triple* $\ell \Rrightarrow p\ (A)$*, where* ℓ *is a closed pattern* $\mathtt{f}\ \ell_1 \cdots \ell_k$*,* p *is a closed meta-term* $\mathtt{g}\ p_1 \cdots p_n$*, and* A *is a set of* meta-variable conditions*: pairs* $Z : i$ *indicating that* Z *regards its* i^{th} *argument. A DP is* conservative *if* $FMV(p) \subseteq FMV(\ell)$*.*

A substitution γ respects *a set of meta-variable conditions* A *if for all* $Z : i$ *in* A *we have* $\gamma(Z) = \lambda x_1 \ldots x_j.t$ *with either* $i > j$*, or* $i \leq j$ *and* $x_i \in FV(t)$*. DPs will be used only with substitutions that respect their meta-variable conditions.*

For $\ell \Rrightarrow p\ (\emptyset)$ *(so a DP whose set of meta-variable conditions is empty), we often omit the third component and just write* $\ell \Rrightarrow p$*.*

Like the first-order setting, the static DP approach employs *marked function symbols* to obtain meta-terms whose instances cannot be reduced at the root.

Definition 22 (Marked symbols). *Let* $(\mathcal{F}, \mathcal{R})$ *be an AFSM. Define* $\mathcal{F}^\sharp := \mathcal{F} \uplus \{\mathtt{f}^\sharp : \sigma \mid \mathtt{f} : \sigma \in \mathcal{D}\}$*. For a meta-term* $s = \mathtt{f}\ s_1 \cdots s_k$ *with* $\mathtt{f} \in \mathcal{D}$ *and* $k = minar(\mathtt{f})$*, we let* $s^\sharp = \mathtt{f}^\sharp\ s_1 \cdots s_k$*; for* s *of other forms* $s^\sharp$ *is not defined.*

Moreover, we will consider *candidates*. In the first-order setting, candidate terms are subterms of the right-hand sides of rules whose root symbol is a defined symbol. Intuitively, these subterms correspond to function calls. In the current setting, we have to consider also meta-variables as well as rules whose right-hand side is not β-normal (which might arise for instance due to η-expansion).

Definition 23 (β-reduced-sub-meta-term, $\trianglerighteq_\beta$, $\trianglerighteq_A$). *A meta-term* s *has a fully applied* β-reduced-sub-meta-term t *(shortly,* BRSMT*), notation* $s \trianglerighteq_\beta t$*, if there exists a set of meta-variable conditions* A *with* $s \trianglerighteq_A t$*. Here* $s \trianglerighteq_A t$ *holds if:*

- $s = t$*, or*
- $s = \lambda x.u$ *and* $u \trianglerighteq_A t$*, or*

- $s = (\lambda x.u)\ s_0 \cdots s_n$ *and some* $s_i \trianglerighteq_A t$, *or* $u[x := s_0]\ s_1 \cdots s_n \trianglerighteq_A t$, *or*
- $s = a\ s_1 \cdots s_n$ *with* $a \in \mathcal{F} \cup \mathcal{V}$ *and some* $s_i \trianglerighteq_A t$, *or*
- $s = Z\langle t_1, \ldots, t_k\rangle\ s_1 \cdots s_n$ *and some* $s_i \trianglerighteq_A t$, *or*
- $s = Z\langle t_1, \ldots, t_k\rangle\ s_1 \cdots s_n$ *and* $t_i \trianglerighteq_A t$ *for some* $i \in \{1, \ldots, k\}$ *with* $(Z : i) \in A$.

Essentially, $s \trianglerighteq_A t$ means that t can be reached from s by taking β-reductions at the root and "subterm"-steps, where $Z : i$ is in A whenever we pass into argument i of a meta-variable Z. BRSMTs are used to generate *candidates*:

Definition 24 (Candidates). *For a meta-term* s, *the set* $\mathsf{cand}(s)$ *of* candidates of s *consists of those pairs* $t\ (A)$ *such that (a)* t *has the form* $\mathtt{f}\ s_1 \cdots s_k$ *with* $\mathtt{f} \in \mathcal{D}$ *and* $k = minar(\mathtt{f})$, *and (b) there are* $s_{k+1}, \ldots, s_n$ *(with* $n \geq k$*) such that* $s \trianglerighteq_A t\ s_{k+1} \cdots s_n$, *and (c)* A *is minimal: there is no subset* $A' \subsetneq A$ *with* $s \trianglerighteq_{A'} t$.

Example 25. In AFSMs where all meta-variables have arity 0 and the right-hand sides of rules are β-normal, the set $\mathsf{cand}(s)$ for a meta-term s consists exactly of the pairs $t\ (\emptyset)$ where t has the form $\mathtt{f}\ s_1 \cdots s_{minar(\mathtt{f})}$ and t occurs as part of s. In Example 8, we thus have $\mathsf{cand}(G\ H\ (\lambda m.\mathtt{rec}\ (H\ m)\ K\ F\ G)) = \{\mathtt{rec}\ (H\ m)\ K\ F\ G\ (\emptyset)\}$.

If some of the meta-variables *do* take arguments, then the meta-variable conditions matter: candidates of s are pairs $t\ (A)$ where A contains exactly those pairs $Z : i$ for which we pass through the i^{th} argument of Z to reach t in s.

Example 26. Consider an AFSM with the signature from Example 8 but a rule using meta-variables with larger arities:

$$\mathtt{rec}\ (\mathtt{lim}\ (\lambda n.H\langle n\rangle))\ K\ (\lambda x.\lambda n.F\langle x, n\rangle)\ (\lambda f.\lambda g.G\langle f, g\rangle)\ \Rightarrow$$
$$G\langle \lambda n.H\langle n\rangle,\ \lambda m.\mathtt{rec}\ H\langle m\rangle\ K\ (\lambda x.\lambda n.F\langle x, n\rangle)\ (\lambda f.\lambda g.G\langle f, g\rangle)\rangle$$

The right-hand side has one candidate:

$$\mathtt{rec}\ H\langle m\rangle\ K\ (\lambda x.\lambda n.F\langle x, n\rangle)\ (\lambda f.\lambda g.G\langle f, g\rangle)\ (\{G : 2\})$$

The original static approaches define DPs as pairs $\ell^\sharp \Rrightarrow p^\sharp$ where $\ell \Rightarrow r$ is a rule and p a subterm of r of the form $\mathtt{f}\ r_1 \cdots r_m$ – as their rules are built using terms, not meta-terms. This can set variables bound in r free in p. In the current setting, we use candidates with their meta-variable conditions and implicit β-steps rather than subterms, and we replace such variables by meta-variables.

Definition 27 (*SDP*). *Let* s *be a meta-term and* $(\mathcal{F}, \mathcal{R})$ *be an AFSM. Let* $metafy(s)$ *denote* s *with all free variables replaced by corresponding meta-variables. Now* $SDP(\mathcal{R}) = \{\ell^\sharp \Rrightarrow metafy(p^\sharp)\ (A) \mid \ell \Rightarrow r \in \mathcal{R} \wedge p\ (A) \in \mathsf{cand}(r)\}$.

Although static DPs always have a pleasant form $\mathtt{f}^\sharp\ \ell_1 \cdots \ell_k \Rrightarrow \mathtt{g}^\sharp\ p_1 \cdots p_n\ (A)$ (as opposed to the *dynamic* DPs of, e.g., [31], whose right-hand sides can have a meta-variable at the head, which complicates various techniques

in the framework), they have two important complications not present in first-order DPs: the right-hand side p of a DP $\ell \Rrightarrow p\ (A)$ may contain meta-variables that do not occur in the left-hand side ℓ – traditional analysis techniques are not really equipped for this – and the left- and right-hand sides may have different types. In Sect. 5 we will explore some methods to deal with these features.

Example 28. For the non-η-expanded rules of Example 17, the set $SDP(\mathcal{R})$ has one element: $\mathtt{deriv}^\sharp\ (\lambda x.\mathtt{sin}\ F\langle x\rangle) \Rrightarrow \mathtt{deriv}^\sharp\ (\lambda x.F\langle x\rangle)$. (As `times` and `cos` are not defined symbols, they do not generate dependency pairs.) The set $SDP(\mathcal{R}^\uparrow)$ for the η-expanded rules is $\{\mathtt{deriv}^\sharp\ (\lambda x.\mathtt{sin}\ F\langle x\rangle)\ Y \Rrightarrow \mathtt{deriv}^\sharp\ (\lambda x.F\langle x\rangle)\ Y\}$. To obtain the relevant candidate, we used the β-reduction step of BRSMTs.

Example 29. The AFSM from Example 8 is AFP following Example 19; here $SDP(\mathcal{R})$ is:

$$\begin{array}{c}\mathtt{rec}^\sharp\ (\mathtt{s}\ X)\ K\ F\ G \Rrightarrow \mathtt{rec}^\sharp\ X\ K\ F\ G\ (\emptyset)\\ \mathtt{rec}^\sharp\ (\mathtt{lim}\ H)\ K\ F\ G \Rrightarrow \mathtt{rec}^\sharp\ (H\ M)\ K\ F\ G\ (\emptyset)\end{array}$$

Note that the right-hand side of the second DP contains a meta-variable that is not on the left. As we will see in Example 64, that is not problematic here.

Termination analysis using dependency pairs importantly considers the notion of a *dependency chain*. This notion is fairly similar to the first-order setting:

Definition 30 (Dependency chain). *Let $\mathcal{P}$ be a set of DPs and $\mathcal{R}$ a set of rules. A (finite or infinite) $(\mathcal{P},\mathcal{R})$-*dependency chain *(or just $(\mathcal{P},\mathcal{R})$-chain) is a sequence $[(\ell_0 \Rrightarrow p_0\ (A_0), s_0, t_0), (\ell_1 \Rrightarrow p_1\ (A_1), s_1, t_1), \ldots]$ where each $\ell_i \Rrightarrow p_i\ (A_i) \in \mathcal{P}$ and all s_i, t_i are terms, such that for all i:*

1. *there exists a substitution γ on domain $FMV(\ell_i) \cup FMV(p_i)$ such that $s_i = \ell_i\gamma$, $t_i = p_i\gamma$ and for all $Z \in \mathtt{dom}(\gamma)$: $\gamma(Z)$ respects A_i;*
2. *we can write $t_i = \mathtt{f}\ u_1 \cdots u_n$ and $s_{i+1} = \mathtt{f}\ w_1 \cdots w_n$ and each $u_j \Rightarrow_{\mathcal{R}}^* w_j$.*

Example 31. In the (first) AFSM from Example 6, we have $SDP(\mathcal{R}) = \{\mathtt{map}^\sharp\ (\lambda x.Z\langle x\rangle)(\mathtt{cons}\ H\ T) \Rrightarrow \mathtt{map}^\sharp\ (\lambda x.Z\langle x\rangle)\ T\}$. An example of a finite dependency chain is $[(\rho, s_1, t_1), (\rho, s_2, t_2)]$ where ρ is the one DP, $s_1 = \mathtt{map}^\sharp\ (\lambda x.\mathtt{s}\ x)\ (\mathtt{cons}\ \mathtt{0}\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ (\mathtt{map}\ (\lambda x.x)\ \mathtt{nil})))$ and $t_1 = \mathtt{map}^\sharp\ (\lambda x.\mathtt{s}\ x)\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ (\mathtt{map}\ (\lambda x.x)\ \mathtt{nil}))$ and $s_2 = \mathtt{map}^\sharp\ (\lambda x.\mathtt{s}\ x)\ (\mathtt{cons}\ (\mathtt{s}\ \mathtt{0})\ \mathtt{nil})$ and $t_2 = \mathtt{map}^\sharp\ (\lambda x.\mathtt{s}\ x)\ \mathtt{nil}$.

Note that here t_1 reduces to s_2 in a single step $(\mathtt{map}\ (\lambda x.x)\ \mathtt{nil} \Rightarrow_{\mathcal{R}} \mathtt{nil})$.

We have the following key result:

Theorem 32. *Let $(\mathcal{F},\mathcal{R})$ be a PA-AFP AFSM. If $(\mathcal{F},\mathcal{R})$ is non-terminating, then there is an infinite $(SDP(\mathcal{R}),\mathcal{R})$-dependency chain.*

Proof (sketch). The proof is an adaptation of the one in [34], altered for the more permissive definition of *accessible function passing* over *plain function passing* as well as the meta-variable conditions; it also follows from Theorem 37 below. □

By this result we can use dependency pairs to prove termination of a given properly applied and AFP AFSM: if we can prove that there is no infinite $(SDP(\mathcal{R}), \mathcal{R})$-chain, then termination follows immediately. Note, however, that the reverse result does *not* hold: it is possible to have an infinite $(SDP(\mathcal{R}), \mathcal{R})$-dependency chain even for a terminating PA-AFP AFSM.

Example 33. Let $\mathcal{F} \supseteq \{\mathtt{0}, \mathtt{1} : \mathtt{nat}, \mathtt{f} : \mathtt{nat} \to \mathtt{nat}, \mathtt{g} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}\}$ and $\mathcal{R} = \{\mathtt{f}\ \mathtt{0} \Rightarrow \mathtt{g}\ (\lambda x.\mathtt{f}\ x), \mathtt{g}\ (\lambda x.F\langle x\rangle) \Rightarrow F\langle \mathtt{1}\rangle\}$. This AFSM is PA-AFP, with $SDP(\mathcal{R}) = \{\mathtt{f}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{g}^\sharp\ (\lambda x.\mathtt{f}\ x),\ \mathtt{f}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{f}^\sharp\ X\}$; the second rule does not cause the addition of any dependency pairs. Although $\Rightarrow_{\mathcal{R}}$ is terminating, there is an infinite $(SDP(\mathcal{R}), \mathcal{R})$-chain $[(\mathtt{f}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{f}^\sharp\ X, \mathtt{f}^\sharp\ \mathtt{0}, \mathtt{f}^\sharp\ \mathtt{0}), (\mathtt{f}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{f}^\sharp\ X, \mathtt{f}^\sharp\ \mathtt{0}, \mathtt{f}^\sharp\ \mathtt{0}), \ldots]$.

The problem in Example 33 is the *non-conservative* DP $\mathtt{f}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{f}^\sharp\ X$, with X on the right but not on the left. Such DPs arise from *abstractions* in the right-hand sides of rules. Unfortunately, abstractions are introduced by the restricted η-expansion (Definition 15) that we may need to make an AFSM properly applied. Even so, often all DPs are conservative, like Examples 6 and 17. There, we do have the inverse result:

Theorem 34. *For any AFSM $(\mathcal{F}, \mathcal{R})$: if there is an infinite $(SDP(\mathcal{R}), \mathcal{R})$-chain $[(\rho_0, s_0, t_0), (\rho_1, s_1, t_1), \ldots]$ with all ρ_i conservative, then $\Rightarrow_{\mathcal{R}}$ is non-terminating.*

Proof (sketch). If $FMV(p_i) \subseteq FMV(\ell_i)$, then we can see that $s_i \Rightarrow_{\mathcal{R}} \cdot \Rightarrow_\beta^* t_i'$ for some term t_i' of which t_i is a subterm. Since also each $t_i \Rightarrow_{\mathcal{R}}^* s_{i+1}$, the infinite chain induces an infinite reduction $s_0 \Rightarrow_{\mathcal{R}}^+ t_0' \Rightarrow_{\mathcal{R}}^* s_1' \Rightarrow_{\mathcal{R}}^+ t_1'' \Rightarrow_{\mathcal{R}}^* \ldots$. □

The core of the dependency pair *framework* is to systematically simplify a set of pairs $(\mathcal{P}, \mathcal{R})$ to prove either absence or presence of an infinite $(\mathcal{P}, \mathcal{R})$-chain, thus showing termination or non-termination as appropriate. By Theorems 32 and 34 we can do so, although with some conditions on the non-termination result. We can do better by tracking certain properties of dependency chains.

Definition 35 (Minimal and Computable chains). *Let $(\mathcal{F}, \mathcal{U})$ be an AFSM and $C_{\mathcal{U}}$ an RC-set satisfying the properties of Theorem 13 for $(\mathcal{F}, \mathcal{U})$. Let $\mathcal{F}$ contain, for every type σ, at least countably many symbols $\mathtt{f} : \sigma$ not used in $\mathcal{U}$.*

A $(\mathcal{P}, \mathcal{R})$-chain $[(\rho_0, s_0, t_0), (\rho_1, s_1, t_1), \ldots]$ is $\mathcal{U}$-computable *if: $\Rightarrow_{\mathcal{U}} \supseteq \Rightarrow_{\mathcal{R}}$, and for all $i \in \mathbb{N}$ there exists a substitution γ_i such that $\rho_i = \ell_i \Rrightarrow p_i\ (A_i)$ with $s_i = \ell_i\gamma_i$ and $t_i = p_i\gamma_i$, and $(\lambda x_1 \ldots x_n.v)\gamma_i$ is $C_{\mathcal{U}}$-computable for all v and B such that $p_i \trianglerighteq_B v$, γ_i respects B, and $FV(v) = \{x_1, \ldots, x_n\}$.*

A chain is minimal *if the strict subterms of all t_i are terminating under $\Rightarrow_{\mathcal{R}}$.*

In the first-order DP framework, *minimal* chains give access to several powerful techniques to prove absence of infinite chains, such as the *subterm criterion* [24] and *usable rules* [22,24]. *Computable* chains go a step further, by building on the computability inherent in the proof of Theorem 32 and the notion of *accessible function passing* AFSMs. In computable chains, we can require that (some of) the subterms of all t_i are *computable* rather than merely *terminating*.

This property will be essential in the *computable subterm criterion processor* (Theorem 63).

Another property of dependency chains is the use of *formative rules*, which has proven very useful for dynamic DPs [31]. Here we go further and consider *formative reductions*, which were introduced for the first-order DP framework in [16]. This property will be essential in the *formative rules processor* (Theorem 58).

Definition 36 (Formative chain, formative reduction). *A $(\mathcal{P},\mathcal{R})$-chain $[(\ell_0 \Rrightarrow p_0\ (A_0), s_0, t_0), (\ell_1 \Rrightarrow p_1\ (A_1), s_1, t_1), \ldots]$ is* formative *if for all i, the reduction $t_i \Rightarrow_{\mathcal{R}}^* s_{i+1}$ is ℓ_{i+1}-formative. Here, for a pattern ℓ, substitution γ and term s, a reduction $s \Rightarrow_{\mathcal{R}}^* \ell\gamma$ is* ℓ-formative *if one of the following holds:*

- *ℓ is not a fully extended linear pattern; that is: some meta-variable occurs more than once in ℓ or ℓ has a sub-meta-term $\lambda x.C[Z\langle \boldsymbol{s} \rangle]$ with $x \notin \{\boldsymbol{s}\}$*
- *ℓ is a meta-variable application $Z\langle x_1, \ldots, x_k \rangle$ and $s = \ell\gamma$*
- *$s = a\ s_1 \cdots s_n$ and $\ell = a\ \ell_1 \cdots \ell_n$ with $a \in \mathcal{F}^\sharp \cup \mathcal{V}$ and each $s_i \Rightarrow_{\mathcal{R}}^* \ell_i\gamma$ by an ℓ_i-formative reduction*
- *$s = \lambda x.s'$ and $\ell = \lambda x.\ell'$ and $s' \Rightarrow_{\mathcal{R}}^* \ell'\gamma$ by an ℓ'-formative reduction*
- *$s = (\lambda x.u)\ v\ w_1 \cdots w_n$ and $u[x := v]\ w_1 \cdots w_n \Rightarrow_{\mathcal{R}}^* \ell\gamma$ by an ℓ-formative reduction*
- *ℓ is not a meta-variable application, and there are $\ell' \Rightarrow r' \in \mathcal{R}$, meta-variables $Z_1 \ldots Z_n$ $(n \geq 0)$ and δ such that $s \Rightarrow_{\mathcal{R}}^* (\ell'\ Z_1 \cdots Z_n)\delta$ by an $(\ell'\ Z_1 \cdots Z_n)$-formative reduction, and $(r'\ Z_1 \cdots Z_n)\delta \Rightarrow_{\mathcal{R}}^* \ell\gamma$ by an ℓ-formative reduction.*

The idea of a formative reduction is to avoid redundant steps: if $s \Rightarrow_{\mathcal{R}}^* \ell\gamma$ by an ℓ-formative reduction, then this reduction takes only the steps needed to obtain an instance of ℓ. Suppose that we have rules $\mathtt{plus}\ \mathtt{0}\ Y \Rightarrow Y$, $\mathtt{plus}\ (\mathtt{s}\ X)\ Y \Rightarrow \mathtt{s}\ (\mathtt{plus}\ X\ Y)$. Let $\ell := \mathtt{g}\ \mathtt{0}\ X$ and $t := \mathtt{plus}\ \mathtt{0}\ \mathtt{0}$. Then the reduction $\mathtt{g}\ t\ t \Rightarrow_{\mathcal{R}} \mathtt{g}\ \mathtt{0}\ t$ is ℓ-formative: we must reduce the first argument to get an instance of ℓ. The reduction $\mathtt{g}\ t\ t \Rightarrow_{\mathcal{R}} \mathtt{g}\ t\ \mathtt{0} \Rightarrow_{\mathcal{R}} \mathtt{g}\ \mathtt{0}\ \mathtt{0}$ is not ℓ-formative, because the reduction in the second argument does not contribute to the non-meta-variable positions of ℓ. This matters when we consider ℓ as the left-hand side of a rule, say $\mathtt{g}\ \mathtt{0}\ X \Rightarrow \mathtt{0}$: if we reduce $\mathtt{g}\ t\ t \Rightarrow_{\mathcal{R}} \mathtt{g}\ t\ \mathtt{0} \Rightarrow_{\mathcal{R}} \mathtt{g}\ \mathtt{0}\ \mathtt{0} \Rightarrow_{\mathcal{R}} \mathtt{0}$, then the first step was redundant: removing this step gives a shorter reduction to the same result: $\mathtt{g}\ t\ t \Rightarrow_{\mathcal{R}} \mathtt{g}\ \mathtt{0}\ t \Rightarrow_{\mathcal{R}} \mathtt{0}$. In an infinite reduction, redundant steps may also be postponed indefinitely.

We can now strengthen the result of Theorem 32 with two new properties.

Theorem 37. *Let $(\mathcal{F},\mathcal{R})$ be a* properly applied, accessible function passing *AFSM. If $(\mathcal{F},\mathcal{R})$ is non-terminating, then there is an infinite $\mathcal{R}$-computable formative $(SDP(\mathcal{R}),\mathcal{R})$-dependency chain.*

Proof (sketch). We select a *minimal non-computable (MNC)* term $s := \mathtt{f}\ s_1 \cdots s_k$ (where all s_i are $C_{\mathcal{R}}$-computable) and an infinite reduction starting in s. Then we stepwise build an infinite dependency chain, as follows. Since s is non-computable but each s_i terminates (as computability implies termination), there exist a rule

$\mathtt{f}\ \ell_1 \cdots \ell_k \Rightarrow r$ and substitution γ such that each $s_i \Rightarrow_{\mathcal{R}}^* \ell_i\gamma$ and $r\gamma$ is non-computable. We can then identify a candidate t (A) of r such that γ respects A and $t\gamma$ is a MNC subterm of $r\gamma$; we continue the process with $t\gamma$ (or a term at its head). For the *formative* property, we note that if $s \Rightarrow_{\mathcal{R}}^* \ell\gamma$ and u is terminating, then $u \Rightarrow_{\mathcal{R}}^* \ell\delta$ by an ℓ-formative reduction for substitution δ such that each $\delta(Z) \Rightarrow_{\mathcal{R}}^* \gamma(Z)$. This follows by postponing those reduction steps not needed to obtain an instance of ℓ. The resulting infinite chain is $\mathcal{R}$-computable because we can show, by induction on the definition of $\trianglerighteq_{\mathtt{acc}}$, that if $\ell \Rightarrow r$ is an AFP rule and $\ell\gamma$ is a MNC term, then $\gamma(Z)$ is $C_{\mathcal{R}}$-computable for all $Z \in FMV(r)$. □

As it is easily seen that all $C_{\mathcal{U}}$-computable terms are $\Rightarrow_{\mathcal{U}}$-terminating and therefore $\Rightarrow_{\mathcal{R}}$-terminating, every $\mathcal{U}$-computable $(\mathcal{P}, \mathcal{R})$-dependency chain is also minimal. The notions of $\mathcal{R}$-computable and formative chains still do not suffice to obtain a true inverse result, however (i.e., to prove that termination implies the absence of an infinite $\mathcal{R}$-computable chain over $SDP(\mathcal{R})$): the infinite chain in Example 33 is $\mathcal{R}$-computable.

To see why the two restrictions that the AFSM must be *properly applied* and *accessible function passing* are necessary, consider the following examples.

Example 38. Consider $\mathcal{F} \supseteq \{\mathtt{fix} : ((\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}\}$ and $\mathcal{R} = \{\mathtt{fix}\ F\ X \Rightarrow F\ (\mathtt{fix}\ F)\ X\}$. This AFSM is not properly applied; it is also not terminating, as can be seen by instantiating F with $\lambda y.y$. However, it does not have any static DPs, since $\mathtt{fix}\ F$ is not a candidate. Even if we altered the definition of static DPs to admit a dependency pair $\mathtt{fix}^\sharp\ F\ X \Rrightarrow \mathtt{fix}^\sharp\ F$, this pair could not be used to build an infinite dependency chain.

Note that the problem does not arise if we study the η-expanded rules $\mathcal{R}^\uparrow = \{\mathtt{fix}\ F\ X \Rightarrow F\ (\lambda z.\mathtt{fix}\ F\ z)\ X\}$, as the dependency pair $\mathtt{fix}^\sharp\ F\ X \Rrightarrow \mathtt{fix}^\sharp\ F\ Z$ does admit an infinite chain. Unfortunately, as the one dependency pair does not satisfy the conditions of Theorem 34, we cannot use this to prove non-termination.

Example 39. The AFSM from Example 20 is not accessible function passing, since $Acc(\mathtt{lm}) = \emptyset$. This is good because the set $SDP(\mathcal{R})$ is empty, which would lead us to falsely conclude termination without the restriction.

Discussion: Theorem 37 transposes the work of [34,46] to AFSMs and extends it by using a more liberal restriction, by limiting interest to *formative*, $\mathcal{R}$-*computable* chains, and by including meta-variable conditions. Both of these new properties of chains will support new termination techniques within the DP framework.

The relationship with the works for functional programming [32,33] is less clear: they define a different form of chains suited well to polymorphic systems, but which requires more intricate reasoning for non-polymorphic systems, as DPs can be used for reductions at the head of a term. It is not clear whether there are non-polymorphic systems that can be handled with one and not the other. The notions of formative and $\mathcal{R}$-computable chains are not considered there; meta-variable conditions are not relevant to their λ-free formalism.

5 The Static Higher-Order DP Framework

In first-order term rewriting, the DP *framework* [20] is an extendable framework to prove termination and non-termination. As observed in the introduction, DP analyses in higher-order rewriting typically go beyond the initial DP *approach* [2], but fall short of the full *framework*. Here, we define the latter for static DPs. Complete versions of all proof sketches in this section are in [17, Appendix C].

We have now reduced the problem of termination to non-existence of certain chains. In the DP framework, we formalise this in the notion of a *DP problem*:

Definition 40 (DP problem). *A* DP problem *is a tuple* $(\mathcal{P}, \mathcal{R}, m, f)$ *with* $\mathcal{P}$ *a set of DPs,* $\mathcal{R}$ *a set of rules,* $m \in \{\texttt{minimal}, \texttt{arbitrary}\} \cup \{\texttt{computable}_{\mathcal{U}} \mid$ *any set of rules* $\mathcal{U}\}$*, and* $f \in \{\texttt{formative}, \texttt{all}\}$.[3]

A DP problem $(\mathcal{P}, \mathcal{R}, m, f)$ *is* finite *if there exists no infinite* $(\mathcal{P}, \mathcal{R})$*-chain that is* $\mathcal{U}$*-computable if* $m = \texttt{computable}_{\mathcal{U}}$*, is minimal if* $m = \texttt{minimal}$*, and is formative if* $f = \texttt{formative}$*. It is* infinite *if* $\mathcal{R}$ *is non-terminating, or if there exists an infinite* $(\mathcal{P}, \mathcal{R})$*-chain where all DPs used in the chain are conservative.*

To capture the levels of permissiveness in the m *flag, we use a transitive-reflexive relation* $\succeq$ *generated by* $\texttt{computable}_{\mathcal{U}} \succeq \texttt{minimal} \succeq \texttt{arbitrary}$.

Thus, the combination of Theorems 34 and 37 can be rephrased as: an AFSM $(\mathcal{F}, \mathcal{R})$ is terminating if $(SDP(\mathcal{R}), \mathcal{R}, \texttt{computable}_{\mathcal{R}}, \texttt{formative})$ is finite, and is non-terminating if $(SDP(\mathcal{R}), \mathcal{R}, m, f)$ is infinite for some $m \in \{\texttt{computable}_{\mathcal{U}}, \texttt{minimal}, \texttt{arbitrary}\}$ and $f \in \{\texttt{formative}, \texttt{all}\}$.[4]

The core idea of the DP framework is to iteratively simplify a set of DP problems via *processors* until nothing remains to be proved:

Definition 41 (Processor). *A* dependency pair processor *(or just* processor*) is a function that takes a DP problem and returns either* `NO` *or a set of DP problems. A processor Proc is* sound *if a DP problem* M *is finite whenever* $Proc(M) \neq \texttt{NO}$ *and all elements of* $Proc(M)$ *are finite. A processor Proc is* complete *if a DP problem* M *is infinite whenever* $Proc(M) = \texttt{NO}$ *or contains an infinite element.*

To prove finiteness of a DP problem M with the DP framework, we proceed analogously to the first-order DP framework [22]: we repeatedly apply sound DP processors starting from M until none remain. That is, we execute the following rough procedure: (1) let $A := \{M\}$; (2) while $A \neq \emptyset$: select a problem $Q \in A$ and a sound processor *Proc* with $Proc(Q) \neq \texttt{NO}$, and let $A := (A \setminus \{Q\}) \cup Proc(Q)$. If this procedure terminates, then M is a finite DP problem.

[3] Our framework is implicitly parametrised by the signature $\mathcal{F}^\sharp$ used for term formation. As none of the processors we present modify this component (as indeed there is no need to by Theorem 9), we leave it implicit.

[4] The processors in this paper do not *alter* the flag m, but some *require* minimality or computability. We include the `minimal` option and the subscript $\mathcal{U}$ for the sake of future generalisations, and for reuse of processors in the *dynamic* approach of [31].

To prove termination of an AFSM $(\mathcal{F}, \mathcal{R})$, we would use as initial DP problem $(SDP(\mathcal{R}), \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})$, provided that $\mathcal{R}$ is properly applied and accessible function passing (where η-expansion following Definition 15 may be applied first). If the procedure terminates – so finiteness of M is proved by the definition of soundness – then Theorem 37 provides termination of $\Rightarrow_{\mathcal{R}}$.

Similarly, we can use the DP framework to prove infiniteness: (1) let $A := \{M\}$; (2) while $A \neq \mathtt{NO}$: select a problem $Q \in A$ and a complete processor *Proc*, and let $A := \mathtt{NO}$ if $Proc(Q) = \mathtt{NO}$, or $A := (A \setminus \{Q\}) \cup Proc(Q)$ otherwise. For non-termination of $(\mathcal{F}, \mathcal{R})$, the initial DP problem should be $(SDP(\mathcal{R}), \mathcal{R}, m, f)$, where m, f can be any flag (see Theorem 34). Note that the algorithms coincide while processors are used that are both sound *and* complete. In a tool, automation (or the user) must resolve the non-determinism and select suitable processors.

Below, we will present a number of processors within the framework. We will typically present processors by writing "for a DP problem M satisfying X, Y, Z, $Proc(M) = \ldots$". In these cases, we let $Proc(M) = \{M\}$ for any problem M not satisfying the given properties. Many more processors are possible, but we have chosen to present a selection which touches on all aspects of the DP framework:

- processors which map a DP problem to NO (Theorem 65), a singleton set (most processors) and a non-singleton set (Theorem 42);
- changing the set $\mathcal{R}$ (Theorems 54, 58) and various flags (Theorem 54);
- using specific values of the f (Theorem 58) and m flags (Theorems 54, 61, 63);
- using term orderings (Theorems 49, 52), a key part of many termination proofs.

5.1 The Dependency Graph

We can leverage reachability information to *decompose* DP problems. In first-order rewriting, a graph structure is used to track which DPs can possibly follow one another in a chain [2]. Here, we define this *dependency graph* as follows.

Definition 42 (Dependency graph). *A DP problem $(\mathcal{P}, \mathcal{R}, m, f)$ induces a graph structure DG, called its* dependency graph*, whose nodes are the elements of $\mathcal{P}$. There is a (directed) edge from ρ_1 to ρ_2 in DG iff there exist s_1, t_1, s_2, t_2 such that $[(\rho_1, s_1, t_1), (\rho_2, s_2, t_2)]$ is a $(\mathcal{P}, \mathcal{R})$-chain with the properties for m, f.*

Example 43. Consider an AFSM with $\mathcal{F} \supseteq \{\mathtt{f} : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat}\}$ and $\mathcal{R} = \{\mathtt{f}\ (\lambda x.F\langle x\rangle)\ (\mathtt{s}\ Y) \Rightarrow F\langle \mathtt{f}\ (\lambda x.\mathtt{0})\ (\mathtt{f}\ (\lambda x.F\langle x\rangle)\ Y)\rangle\}$. Let $\mathcal{P} := SDP(\mathcal{R}) =$

$$\left\{ \begin{array}{ll} (1)\ \mathtt{f}^\sharp\ (\lambda x.F\langle x\rangle)\ (\mathtt{s}\ Y) \Rrightarrow \mathtt{f}^\sharp\ (\lambda x.\mathtt{0})\ (\mathtt{f}\ (\lambda x.F\langle x\rangle)\ Y) & (\{F:1\}) \\ (2)\ \mathtt{f}^\sharp\ (\lambda x.F\langle x\rangle)\ (\mathtt{s}\ Y) \Rrightarrow \mathtt{f}^\sharp\ (\lambda x.F\langle x\rangle)\ Y & (\{F:1\}) \end{array} \right\}$$

The dependency graph of $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ is:

There is no edge from (1) to itself or (2) because there is no substitution γ such that $(\lambda x.\mathtt{0})\gamma$ can be reduced to a term $(\lambda x.F\langle x\rangle)\delta$ where $\delta(F)$ regards its first argument (as $\Rightarrow_{\mathcal{R}}^{*}$ cannot introduce new variables).

In general, the dependency graph for a given DP problem is undecidable, which is why we consider *approximations*.

Definition 44 (Dependency graph approximation [31]). *A finite graph G_θ* approximates *DG if θ is a function that maps the nodes of DG to the nodes of G_θ such that, whenever DG has an edge from ρ_1 to ρ_2, G_θ has an edge from $\theta(\rho_1)$ to $\theta(\rho_2)$. (G_θ may have edges that have no corresponding edge in DG.)*

Note that this definition allows for an *infinite* graph to be approximated by a *finite* one; infinite graphs may occur if $\mathcal{R}$ is infinite (e.g., the union of all simply-typed instances of polymorphic rules).

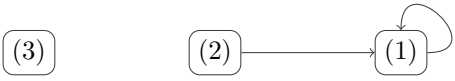
If $\mathcal{P}$ is finite, we can take a graph approximation $G_{\mathtt{id}}$ with the same nodes as DG. A simple approximation may have an edge from $\ell_1 \Rrightarrow p_1\ (A_1)$ to $\ell_2 \Rrightarrow p_2\ (A_2)$ whenever both p_1 and ℓ_2 have the form $\mathtt{f}^\sharp\ s_1 \cdots s_k$ for the same $\mathtt{f}$ and k. However, one can also take the meta-variable conditions into account, as we did in Example 43.

Theorem 45 (Dependency graph processor). *The processor $Proc_{G_\theta}$ that maps a DP problem $M = (\mathcal{P}, \mathcal{R}, m, f)$ to $\{(\{\rho \in \mathcal{P} \mid \theta(\rho) \in C_i\}, \mathcal{R}, m, f) \mid 1 \leq i \leq n\}$ if G_θ is an approximation of the dependency graph of M and $C_1, \ldots, C_n$ are the (nodes of the) non-trivial strongly connected components (SCCs) of G_θ, is both sound and complete.*

Proof (sketch). In an infinite $(\mathcal{P}, \mathcal{R})$-chain $[(\rho_0, s_0, t_0), (\rho_1, s_1, t_1), \ldots]$, there is always a path from ρ_i to ρ_{i+1} in DG. Since G_θ is finite, every infinite path in DG eventually remains in a cycle in G_θ. This cycle is part of an SCC. □

Example 46. Let $\mathcal{R}$ be the set of rules from Example 43 and G be the graph given there. Then $Proc_G(SDP(\mathcal{R}), \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative}) = \{(\{\mathtt{f}^\sharp\ (\lambda x.F\langle x\rangle)\ (\mathtt{s}\ Y) \Rrightarrow \mathtt{f}^\sharp\ (\lambda x.F\langle x\rangle)\ Y\ (\{F : 1\})\}, \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})\}$.

Example 47. Let $\mathcal{R}$ consist of the rules for $\mathtt{map}$ from Example 6 along with $\mathtt{f}\ L \Rightarrow \mathtt{map}\ (\lambda x.\mathtt{g}\ x)\ L$ and $\mathtt{g}\ X \Rightarrow X$. Then $SDP(\mathcal{R}) = \{(1)\ \mathtt{map}^\sharp\ (\lambda x.Z\langle x\rangle)\ (\mathtt{cons}\ H\ T) \Rrightarrow \mathtt{map}^\sharp\ (\lambda x.Z\langle x\rangle)\ T,\ (2)\ \mathtt{f}^\sharp\ L \Rrightarrow \mathtt{map}^\sharp\ (\lambda x.\mathtt{g}\ x)\ L,\ (3)\ \mathtt{f}^\sharp\ L \Rrightarrow \mathtt{g}^\sharp\ X\}$. DP (3) is not conservative, but it is not on any cycle in the graph approximation $G_{\mathtt{id}}$ obtained by considering head symbols as described above:



As (1) is the only DP on a cycle, $Proc_{SDP_{G_{\mathtt{id}}}}(SDP(\mathcal{R}), \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative}) = \{\ (\{(1)\}, \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})\ \}$.

Discussion: The dependency graph is a powerful tool for simplifying DP problems, used since early versions of the DP approach [2]. Our notion of a dependency graph approximation, taken from [31], strictly generalises the original notion in [2], which uses a graph on the same node set as DG with possibly further edges. One can get this notion here by using a graph $G_{\mathtt{id}}$. The advantage of our definition is that it ensures soundness of the dependency graph processor also for *infinite* sets of DPs. This overcomes a restriction in the literature [34, Corollary 5.13] to dependency graphs without non-cyclic infinite paths.

5.2 Processors Based on Reduction Triples

At the heart of most DP-based approaches to termination proving lie well-founded orderings to delete DPs (or rules). For this, we use *reduction triples* [24,31].

Definition 48 (Reduction triple). *A* reduction triple $(\succsim, \succcurlyeq, \succ)$ *consists of two quasi-orderings* $\succsim$ *and* $\succcurlyeq$ *and a well-founded strict ordering* $\succ$ *on meta-terms such that* $\succsim$ *is monotonic, all of* $\succsim, \succcurlyeq, \succ$ *are meta-stable (that is,* $\ell \succsim r$ *implies* $\ell\gamma \succsim r\gamma$ *if* ℓ *is a closed pattern and* γ *a substitution on domain* $FMV(\ell) \cup FMV(r)$*, and the same for* $\succcurlyeq$ *and* $\succ$*),* $\Rightarrow_\beta \subseteq \succsim$*, and both* $\succsim \circ \succ \subseteq \succ$ *and* $\succcurlyeq \circ \succ \subseteq \succ$*.*

In the first-order DP framework, the reduction pair processor [20] seeks to orient all rules with $\succsim$ and all DPs with either $\succsim$ or $\succ$; if this succeeds, those pairs oriented with $\succ$ may be removed. Using reduction *triples* rather than pairs, we obtain the following extension to the higher-order setting:

Theorem 49 (Basic reduction triple processor). *Let* $M = (\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R}, m, f)$ *be a DP problem. If* $(\succsim, \succcurlyeq, \succ)$ *is a reduction triple such that*

1. *for all* $\ell \Rightarrow r \in \mathcal{R}$*, we have* $\ell \succsim r$*;*
2. *for all* $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$*, we have* $\ell \succ p$*;*
3. *for all* $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$*, we have* $\ell \succcurlyeq p$*;*

then the processor that maps M *to* $\{(\mathcal{P}_2, \mathcal{R}, m, f)\}$ *is both sound and complete.*

Proof (sketch). For an infinite $(\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R})$-chain $[(\rho_0, s_0, t_0), (\rho_1, s_1, t_1), \ldots]$ the requirements provide that, for all i: (a) $s_i \succ t_i$ if $\rho_i \in \mathcal{P}_1$; (b) $s_i \succcurlyeq t_i$ if $\rho_i \in \mathcal{P}_2$; and (c) $t_i \succsim s_{i+1}$. Since $\succ$ is well-founded, only finitely many DPs can be in $\mathcal{P}_1$, so a tail of the chain is actually an infinite $(\mathcal{P}_2, \mathcal{R}, m, f)$-chain. □

Example 50. Let $(\mathcal{F}, \mathcal{R})$ be the (non-η-expanded) rules from Example 17, and $SDP(\mathcal{R})$ the DPs from Example 28. From Theorem 49, we get the following ordering requirements:

$$\begin{aligned} &\mathtt{deriv}\ (\lambda x.\mathtt{sin}\ F\langle x\rangle) \succsim \lambda y.\mathtt{times}\ (\mathtt{deriv}\ (\lambda x.F\langle x\rangle)\ y)\ (\mathtt{cos}\ F\langle y\rangle)\\ &\mathtt{deriv}^\sharp\ (\lambda x.\mathtt{sin}\ F\langle x\rangle) \succ \mathtt{deriv}^\sharp\ (\lambda x.F\langle x\rangle)\end{aligned}$$

We can handle both requirements by using a polynomial interpretation $\mathcal{J}$ to $\mathbb{N}$ [15,43], by choosing $\mathcal{J}_{\mathtt{sin}}(n) = n + 1$, $\mathcal{J}_{\mathtt{cos}}(n) = 0$, $\mathcal{J}_{\mathtt{times}}(n_1, n_2) = n_1$, $\mathcal{J}_{\mathtt{deriv}}(f) = \mathcal{J}_{\mathtt{deriv}^\sharp}(f) = \lambda n.f(n)$. Then the requirements are evaluated to: $\lambda n.f(n) + 1 \geq \lambda n.f(n)$ and $\lambda n.f(n) + 1 > \lambda n.f(n)$, which holds on $\mathbb{N}$.

Theorem 49 is not ideal since, by definition, the left- and right-hand side of a DP may have different types. Such DPs are hard to handle with traditional techniques such as HORPO [26] or polynomial interpretations [15,43], as these methods compare only (meta-)terms of the same type (modulo renaming of sorts).

Example 51. Consider the toy AFSM with $\mathcal{R} = \{\mathtt{f}\ (\mathtt{s}\ X)\ Y \Rightarrow \mathtt{g}\ X\ Y,\ \mathtt{g}\ X \Rightarrow \lambda z.\mathtt{f}\ X\ z\}$ and $SDP(\mathcal{R}) = \{\mathtt{f}^\sharp\ (\mathtt{s}\ X)\ Y \Rrightarrow \mathtt{g}^\sharp\ X,\ \mathtt{g}^\sharp\ X \Rrightarrow \mathtt{f}^\sharp\ X\ Z\}$. If $\mathtt{f}$ and $\mathtt{g}$ both have a type $\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$, then in the first DP, the left-hand side has type $\mathtt{nat}$ while the right-hand side has type $\mathtt{nat} \to \mathtt{nat}$. In the second DP, the left-hand side has type $\mathtt{nat} \to \mathtt{nat}$ and the right-hand side has type $\mathtt{nat}$.

To be able to handle examples like the one above, we adapt [31, Thm. 5.21] by altering the ordering requirements to have base type.

Theorem 52 (Reduction triple processor). *Let* Bot *be a set* $\{\bot_\sigma : \sigma \mid \sigma \text{ a type}\} \subseteq \mathcal{F}^\sharp$ *of unused constructors,* $M = (\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R}, m, f)$ *a DP problem and* $(\succsim, \succcurlyeq, \succ)$ *a reduction triple such that: (a) for all* $\ell \Rightarrow r \in \mathcal{R}$*, we have* $\ell \succsim r$*; and (b) for all* $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1 \uplus \mathcal{P}_2$ *with* $\ell : \sigma_1 \to \ldots \to \sigma_m \to \iota$ *and* $p : \tau_1 \to \ldots \to \tau_n \to \kappa$ *we have, for fresh meta-variables* $Z_1 : \sigma_1, \ldots, Z_m : \sigma_m$*:*

- $\ell\ Z_1 \cdots Z_m \succ p\ \bot_{\tau_1} \cdots \bot_{\tau_n}$ *if* $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$
- $\ell\ Z_1 \cdots Z_m \succcurlyeq p\ \bot_{\tau_1} \cdots \bot_{\tau_n}$ *if* $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$

Then the processor that maps M *to* $\{(\mathcal{P}_2, \mathcal{R}, m, f)\}$ *is both sound and complete.*

Proof (sketch). If $(\succsim, \succcurlyeq, \succ)$ is such a triple, then for $R \in \{\succcurlyeq, \succ\}$ define R' as follows: for $s : \sigma_1 \to \ldots \to \sigma_m \to \iota$ and $t : \tau_1 \to \ldots \to \tau_n \to \kappa$, let $s\ R'\ t$ if for all $u_1 : \sigma_1, \ldots, u_m : \sigma_m$ there exist $w_1 : \tau_1, \ldots, w_n : \tau_n$ such that $s\ u_1 \cdots u_m\ R\ t\ w_1 \cdots w_n$. Now apply Theorem 49 with the triple $(\succsim, \succcurlyeq', \succ')$. □

Here, the elements of Bot take the role of minimal terms for the ordering. We use them to flatten the type of the right-hand sides of ordering requirements, which makes it easier to use traditional methods to generate a reduction triple.

While $\succ$ and $\succcurlyeq$ may still have to orient meta-terms of distinct types, these are always *base* types, which we could collapse to a single sort. The only relation required to be monotonic, $\succsim$, regards pairs of meta-terms of the *same* type. This makes it feasible to apply orderings like HORPO or polynomial interpretations.

Both the basic and non-basic reduction triple processor are difficult to use for *non-conservative* DPs, which generate ordering requirements whose right-hand side contains a meta-variable not occurring on the left. This is typically difficult for traditional techniques, although possible to overcome, by choosing triples that do not regard such meta-variables (e.g., via an argument filtering [35,46]):

Example 53. We apply Theorem 52 on the DP problem $(SDP(\mathcal{R}), \mathcal{R}, \texttt{computable}_{\mathcal{R}}, \texttt{formative})$ of Example 51. This gives for instance the following ordering requirements:

$$\begin{array}{cc} \texttt{f}\ (\texttt{s}\ X)\ Y \gtrsim \texttt{g}\ X\ Y & \texttt{f}^\sharp\ (\texttt{s}\ X)\ Y \succ \texttt{g}^\sharp\ X\ \bot_{\texttt{nat}} \\ \texttt{g}\ X \gtrsim \lambda z.\texttt{f}\ X\ z & \texttt{g}^\sharp\ X\ Y \succcurlyeq \texttt{f}^\sharp\ X\ Z \end{array}$$

The right-hand side of the last DP uses a meta-variable Z that does not occur on the left. As neither $\succ$ nor $\succcurlyeq$ are required to be monotonic (only $\gtrsim$ is), function symbols do not have to regard all their arguments. Thus, we can use a polynomial interpretation $\mathcal{J}$ to $\mathbb{N}$ with $\mathcal{J}_{\bot_{\texttt{nat}}} = 0$, $\mathcal{J}_{\texttt{s}}(n) = n + 1$ and $\mathcal{J}_{\texttt{h}}(n_1, n_2) = n_1$ for $\texttt{h} \in \{\texttt{f}, \texttt{f}^\sharp, \texttt{g}, \texttt{g}^\sharp\}$. The ordering requirements then translate to $X + 1 \geq X$ and $\lambda y.X \geq \lambda z.X$ for the rules, and $X + 1 > X$ and $X \geq X$ for the DPs. All these inequalities on $\mathbb{N}$ are clearly satisfied, so we can remove the first DP. The remaining problem is quickly dispersed with the dependency graph processor.

5.3 Rule Removal Without Search for Orderings

While processors often simplify only $\mathcal{P}$, they can also simplify $\mathcal{R}$. One of the most powerful techniques in first-order DP approaches that can do this are *usable rules*. The idea is that for a given set $\mathcal{P}$ of DPs, we only need to consider a *subset* $UR(\mathcal{P}, \mathcal{R})$ of $\mathcal{R}$. Combined with the dependency graph processor, this makes it possible to split a large term rewriting system into a number of small problems.

In the higher-order setting, simple versions of usable rules have also been defined [31,46]. We can easily extend these definitions to AFSMs:

Theorem 54. *Given a DP problem $M = (\mathcal{P}, \mathcal{R}, m, f)$ with $m \succeq \texttt{minimal}$ and $\mathcal{R}$ finite, let $UR(\mathcal{P}, \mathcal{R})$ be the smallest subset of $\mathcal{R}$ such that:*

- *if a symbol $\texttt{f}$ occurs in the right-hand side of an element of $\mathcal{P}$ or $UR(\mathcal{P}, \mathcal{R})$, and there is a rule $\texttt{f}\ \ell_1 \cdots \ell_k \Rightarrow r$, then this rule is also in $UR(\mathcal{P}, \mathcal{R})$;*
- *if there exists $\ell \Rightarrow r \in \mathcal{R}$ or $\ell \Rrightarrow r\ (A) \in \mathcal{P}$ such that $r \trianglerighteq F\langle s_1, \ldots, s_k\rangle\ t_1 \cdots t_n$ with $s_1, \ldots, s_k$ not all distinct variables or with $n > 0$, then $UR(\mathcal{P}, \mathcal{R}) = \mathcal{R}$.*

Then the processor that maps M to $\{(\mathcal{P}, UR(\mathcal{P}, \mathcal{R}), \texttt{arbitrary}, \texttt{all})\}$ is sound.

For the proof we refer to the very similar proofs in [31,46].

Example 55. For the set $SDP(\mathcal{R})$ of the ordinal recursion example (Examples 8 and 29), all rules are usable due to the occurrence of $H\ M$ in the second DP. For the set $SDP(\mathcal{R})$ of the map example (Examples 6 and 31), there are no usable rules, since the one DP contains no defined function symbols or applied meta-variables.

This higher-order processor is much less powerful than its first-order version: if any DP or usable rule has a sub-meta-term of the form $F\ s$ or $F\langle s_1, \ldots, s_k\rangle$ with $s_1, \ldots, s_k$ not all distinct variables, then *all* rules are usable. Since applying a higher-order meta-variable to some argument is extremely common in higher-order rewriting, the technique is usually not applicable. Also, this processor

imposes a heavy price on the flags: minimality (at least) is required, but is lost; the formative flag is also lost. Thus, usable rules are often combined with reduction triples to temporarily disregard rules, rather than as a way to permanently remove rules.

To address these weaknesses, we consider a processor that uses similar ideas to usable rules, but operates from the *left-hand* sides of rules and DPs rather than the right. This adapts the technique from [31] that relies on the new *formative* flag. As in the first-order case [16], we use a semantic characterisation of formative rules. In practice, we then work with over-approximations of this characterisation, analogous to the use of dependency graph approximations in Theorem 45.

Definition 56. *A function FR that maps a pattern ℓ and a set of rules $\mathcal{R}$ to a set $FR(\ell, \mathcal{R}) \subseteq \mathcal{R}$ is a* formative rules approximation *if for all s and γ: if $s \Rightarrow^*_{\mathcal{R}} \ell\gamma$ by an ℓ-formative reduction, then this reduction can be done using only rules in $FR(\ell, \mathcal{R})$.*

We let $FR(\mathcal{P}, \mathcal{R}) = \bigcup\{FR(\ell_i, \mathcal{R}) \mid \mathtt{f}\ \ell_1 \cdots \ell_n \Rrightarrow p\,(A) \in \mathcal{P} \wedge 1 \leq i \leq n\}$.

Thus, a formative rules approximation is a subset of $\mathcal{R}$ that is *sufficient* for a formative reduction: if $s \Rightarrow^*_{\mathcal{R}} \ell\gamma$, then $s \Rightarrow^*_{FR(\ell,\mathcal{R})} \ell\gamma$. It is allowed for there to exist other formative reductions that do use additional rules.

Example 57. We define a simple formative rules approximation: (1) $FR(Z, \mathcal{R}) = \emptyset$ if Z is a meta-variable; (2) $FR(\mathtt{f}\ \ell_1 \cdots \ell_m, \mathcal{R}) = FR(\ell_1, \mathcal{R}) \cup \cdots \cup FR(\ell_m, \mathcal{R})$ if $\mathtt{f} : \sigma_1 \to \ldots \to \sigma_m \to \iota$ and no rules have type ι; (3) $FR(s, \mathcal{R}) = \mathcal{R}$ otherwise. This is a formative rules approximation: if $s \Rightarrow^*_{\mathcal{R}} Z\gamma$ by a Z-formative reduction, then $s = Z\gamma$, and if $s \Rightarrow^*_{\mathcal{R}} \mathtt{f}\ \ell_1 \cdots \ell_m$ and no rules have the same output type as s, then $s = \mathtt{f}\ s_1 \cdots s_m$ and each $s_i \Rightarrow^*_{\mathcal{R}} \ell_i\gamma$ (by an ℓ_i-formative reduction).

The following result follows directly from the definition of formative rules.

Theorem 58 (Formative rules processor). *For a formative rules approximation FR, the processor $Proc_{FR}$ that maps a DP problem $(\mathcal{P}, \mathcal{R}, m, \mathtt{formative})$ to $\{(\mathcal{P}, FR(\mathcal{P}, \mathcal{R}), m, \mathtt{formative})\}$ is both sound and complete.*

Proof (sketch). A processor that only removes rules (or DPs) is always complete. For soundness, if the chain is formative then each step $t_i \Rightarrow^*_{\mathcal{R}} s_{i+1}$ can be replaced by $t_i \Rightarrow^*_{FR(\mathcal{P},\mathcal{R})} s_{i+1}$. Thus, the chain can be seen as a $(\mathcal{P}, FR(\mathcal{P}, \mathcal{R}))$-chain. □

Example 59. For our ordinal recursion example (Examples 8 and 29), *none* of the rules are included when we use the approximation of Example 57 since all rules have output type `ord`. Thus, $Proc_{FR}$ maps $(SDP(\mathcal{R}), \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})$ to $(SDP(\mathcal{R}), \emptyset, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})$. *Note:* this example can also be completed without formative rules (see Example 64). Here we illustrate that, even with a simple formative rules approximation, we can often delete all rules of a given type.

Formative rules are introduced in [31], and the definitions can be adapted to a more powerful formative rules approximation than the one sketched in Example 59. Several examples and deeper intuition for the first-order setting are given in [16].

5.4 Subterm Criterion Processors

Reduction triple processors are powerful, but they exert a computational price: we must orient all rules in $\mathcal{R}$. The subterm criterion processor allows us to remove DPs without considering $\mathcal{R}$ at all. It is based on a *projection function* [24], whose higher-order counterpart [31,34,46] is the following:

Definition 60. *For $\mathcal{P}$ a set of DPs, let $\mathtt{heads}(\mathcal{P})$ be the set of all symbols $\mathtt{f}$ that occur as the head of a left- or right-hand side of a DP in $\mathcal{P}$. A* projection function *for $\mathcal{P}$ is a function $\nu : \mathtt{heads}(\mathcal{P}) \to \mathbb{N}$ such that for all DPs $\ell \Rrightarrow p\ (A) \in \mathcal{P}$, the function $\overline{\nu}$ with $\overline{\nu}(\mathtt{f}\ s_1 \cdots s_n) = s_{\nu(\mathtt{f})}$ is well-defined both for ℓ and for p.*

Theorem 61 (Subterm criterion processor). *The processor $Proc_{\mathtt{subcrit}}$ that maps a DP problem $(\mathcal{P}_1 \uplus \mathcal{P}_2, \mathcal{R}, m, f)$ with $m \succeq \mathtt{minimal}$ to $\{(\mathcal{P}_2, \mathcal{R}, m, f)\}$ if a projection function ν exists such that $\overline{\nu}(\ell) \rhd \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$ and $\overline{\nu}(\ell) = \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$, is sound and complete.*

Proof (sketch). If the conditions are satisfied, every infinite $(\mathcal{P}, \mathcal{R})$-chain induces an infinite $\trianglerighteq \cdot \Rightarrow_{\mathcal{R}}^*$ sequence that starts in a strict subterm of t_1, contradicting minimality unless all but finitely many steps are equality. Since every occurrence of a pair in $\mathcal{P}_1$ results in a strict $\rhd$ step, a tail of the chain lies in $\mathcal{P}_2$. □

Example 62. Using $\nu(\mathtt{map}^\sharp) = 2$, $Proc_{\mathtt{subcrit}}$ maps the DP problem $(\{(1)\}, \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})$ from Example 47 to $\{(\emptyset, \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})\}$.

The subterm criterion can be strengthened, following [34,46], to also handle DPs like the one in Example 28. Here, we focus on a new idea. For *computable* chains, we can build on the idea of the subterm criterion to get something more.

Theorem 63 (Computable subterm criterion processor). *The processor $Proc_{\mathtt{statcrit}}$ that maps a DP problem $(P_1 \uplus \mathcal{P}_2, \mathcal{R}, \mathtt{computable}_{\mathcal{U}}, f)$ to $\{(\mathcal{P}_2, \mathcal{R}, \mathtt{computable}_{\mathcal{U}}, f)\}$ if a projection function ν exists such that $\overline{\nu}(\ell) \sqsupset \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_1$ and $\overline{\nu}(\ell) = \overline{\nu}(p)$ for all $\ell \Rrightarrow p\ (A) \in \mathcal{P}_2$, is sound and complete. Here, $\sqsupset$ is the relation on base-type terms with $s \sqsupset t$ if $s \neq t$ and (a) $s \trianglerighteq_{\mathtt{acc}} t$ or (b) a meta-variable Z exists with $s \trianglerighteq_{\mathtt{acc}} Z\langle x_1, \ldots, x_k\rangle$ and $t = Z\langle t_1, \ldots, t_k\rangle\ s_1 \cdots s_n$.*

Proof (sketch). By the conditions, every infinite $(\mathcal{P}, \mathcal{R})$-chain induces an infinite $(\Rrightarrow_{C_{\mathcal{U}}} \cup \Rightarrow_\beta)^* \cdot \Rightarrow_{\mathcal{R}}^*$ sequence (where $C_{\mathcal{U}}$ is defined following Theorem 13). This contradicts computability unless there are only finitely many inequality steps. As pairs in $\mathcal{P}_1$ give rise to a strict decrease, they may occur only finitely often. □

Example 64. Following Examples 8 and 29, consider the projection function ν with $\nu(\mathtt{rec}^\sharp) = 1$. As $\mathtt{s}\ X \trianglerighteq_{\mathtt{acc}} X$ and $\mathtt{lim}\ H \trianglerighteq_{\mathtt{acc}} H$, both $\mathtt{s}\ X \sqsupset X$ and $\mathtt{lim}\ H \sqsupset H\ M$ hold. Thus $Proc_{\mathtt{statc}}(\mathcal{P}, \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative}) = \{(\emptyset, \mathcal{R}, \mathtt{computable}_{\mathcal{R}}, \mathtt{formative})\}$. By the dependency graph processor, the AFSM is terminating.

The computable subterm criterion processor fundamentally relies on the new $\mathtt{computable}_{\mathcal{U}}$ flag, so it has no counterpart in the literature so far.

5.5 Non-termination

While (most of) the processors presented so far are complete, none of them can actually return `NO`. We have not yet implemented such a processor; however, we can already provide a general specification of a *non-termination processor*.

Theorem 65 (Non-termination processor). *Let $M = (\mathcal{P}, \mathcal{R}, m, f)$ be a DP problem. The processor that maps M to `NO` if it determines that a sufficient criterion for non-termination of $\Rightarrow_{\mathcal{R}}$ or for existence of an infinite conservative $(\mathcal{P}, \mathcal{R})$-chain according to the flags m and f holds is sound and complete.*

Proof. Obvious. □

This is a very general processor, which does not tell us *how* to determine such a sufficient criterion. However, it allows us to conclude non-termination as part of the framework by identifying a suitable infinite chain.

Example 66. If we can find a finite $(\mathcal{P}, \mathcal{R})$-chain $[(\rho_0, s_0, t_0), \ldots, (\rho_n, s_n, t_n)]$ with $t_n = s_0\gamma$ for some substitution γ which uses only conservative DPs, is formative if $f = \texttt{formative}$ and is $\mathcal{U}$-computable if $m = \texttt{computable}_{\mathcal{U}}$, such a chain is clearly a sufficient criterion: there is an infinite chain $[(\rho_0, s_0, t_0), \ldots, (\rho_0, s_0\gamma, t_0\gamma), \ldots, (\rho_0, s_0\gamma\gamma, t_0\gamma\gamma), \ldots]$. If $m = \texttt{minimal}$ and we find such a chain that is however not minimal, then note that $\Rightarrow_{\mathcal{R}}$ is non-terminating, which also suffices.

For example, for a DP problem $(\mathcal{P}, \mathcal{R}, \texttt{minimal}, \texttt{all})$ with $\mathcal{P} = \{\texttt{f}^\sharp\ F\ X \Rrightarrow \texttt{g}^\sharp\ (F\ X),\ \texttt{g}^\sharp\ X \Rrightarrow \texttt{f}^\sharp\ \texttt{h}\ X\}$, there is a finite dependency chain: $[(\texttt{f}^\sharp\ F\ X \Rrightarrow \texttt{g}^\sharp\ (F\ X),\ \texttt{f}^\sharp\ \texttt{h}\ x,\ \texttt{g}^\sharp\ (\texttt{h}\ x)),\ (\texttt{g}^\sharp\ X \Rrightarrow \texttt{f}^\sharp\ \texttt{h}\ X,\ \texttt{g}^\sharp\ (\texttt{h}\ x),\ \texttt{f}^\sharp\ \texttt{h}\ (\texttt{h}\ x))]$. As $\texttt{f}^\sharp\ \texttt{h}\ (\texttt{h}\ x)$ is an instance of $\texttt{f}^\sharp\ \texttt{h}\ x$, the processor maps this DP problem to `NO`.

To instantiate Theorem 65, we can borrow non-termination criteria from first-order rewriting [13,21,42], with minor adaptions to the typed setting. Of course, it is worthwhile to also investigate dedicated higher-order non-termination criteria.

6 Conclusions and Future Work

We have built on the static dependency pair approach [6,33,34,46] and formulated it in the language of the DP *framework* from first-order rewriting [20,22]. Our formulation is based on AFSMs, a dedicated formalism designed to make termination proofs transferrable to various higher-order rewriting formalisms.

This framework has two important additions over existing higher-order DP approaches in the literature. First, we consider not only arbitrary and minimally non-terminating dependency chains, but also minimally *non-computable* chains; this is tracked by the $\texttt{computable}_{\mathcal{U}}$ flag. Using the flag, a dedicated processor allows us to efficiently handle rules like Example 8. This flag has no counterpart in the first-order setting. Second, we have generalised the idea of formative rules in [31] to a notion of formative *chains*, tracked by a `formative` flag. This makes it possible to define a corresponding processor that permanently removes rules.

Implementation and Experiments. To provide a strong formal groundwork, we have presented several processors in a general way, using semantic definitions of, e.g., the dependency graph approximation and formative rules rather than syntactic definitions using functions like *TCap* [21]. Even so, most parts of the DP framework for AFSMs have been implemented in the open-source termination prover WANDA [28], alongside a dynamic DP framework [31] and a mechanism to delegate some ordering constraints to a first-order tool [14]. For reduction triples, polynomial interpretations [15] and a version of HORPO [29, Ch. 5] are used. To solve the constraints arising in the search for these orderings, and also to determine sort orderings (for the accessibility relation) and projection functions (for the subterm criteria), WANDA employs an external SAT-solver. WANDA has won the higher-order category of the International Termination Competition [50] four times. In the International Confluence Competition [10], the tools ACPH [40] and CSIˆho [38] use WANDA as their "oracle" for termination proofs on HRSs.

We have tested WANDA on the *Termination Problems Data Base* [49], using AProVE [19] and MiniSat [12] as back-ends. When no additional features are enabled, WANDA proves termination of 124 (out of 198) benchmarks with static DPs, versus 92 with only a search for reduction orderings; a 34% increase. When all features except static DPs are enabled, WANDA succeeds on 153 benchmarks, versus 166 with also static DPs; an 8% increase, or alternatively, a 29% decrease in failure rate. The full evaluation is available in [17, Appendix D].

Future Work. While the static and the dynamic DP approaches each have their own strengths, there has thus far been little progress on a *unified* approach, which could take advantage of the syntactic benefits of both styles. We plan to combine the present work with the ideas of [31] into such a unified DP framework.

In addition, we plan to extend the higher-order DP framework to rewriting with *strategies*, such as implicit β-normalisation or strategies inspired by functional programming languages like OCaml and Haskell. Other natural directions are dedicated automation to detect non-termination, and reducing the number of term constraints solved by the reduction triple processor via a tighter integration with usable and formative rules with respect to argument filterings.

References

1. Aczel, P.: A general Church-Rosser theorem. Unpublished Manuscript, University of Manchester (1978)
2. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theor. Comput. Sci. **236**(1–2), 133–178 (2000). https://doi.org/10.1016/S0304-3975(99)00207-8
3. Baader, F., Nipkow, F.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
4. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. J. Logic Comput. **4**(3), 217–247 (1994). https://doi.org/10.1093/logcom/4.3.217

5. Blanqui, F.: Termination and confluence of higher-order rewrite systems. In: Bachmair, L. (ed.) RTA 2000. LNCS, vol. 1833, pp. 47–61. Springer, Heidelberg (2000). https://doi.org/10.1007/10721975_4
6. Blanqui, F.: Higher-order dependency pairs. In: Proceedings of the WST 2006 (2006)
7. Blanqui, F.: Termination of rewrite relations on λ-terms based on Girard's notion of reducibility. Theor. Comput. Sci. **611**, 50–86 (2016). https://doi.org/10.1016/j.tcs.2015.07.045
8. Blanqui, F., Jouannaud, J., Okada, M.: Inductive-data-type systems. Theor. Comput. Sci. **272**(1–2), 41–68 (2002). https://doi.org/10.1016/S0304-3975(00)00347-9
9. Blanqui, F., Jouannaud, J., Rubio, A.: The computability path ordering. Logical Methods Comput. Sci. **11**(4) (2015). https://doi.org/10.2168/LMCS-11(4:3)2015
10. Community. The International Confluence Competition (CoCo) (2018). http://project-coco.uibk.ac.at/
11. Dershowitz, N., Kaplan, S.: Rewrite, rewrite, rewrite, rewrite, rewrite. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, 11–13 January 1989, pp. 250–259. ACM Press (1989). https://doi.org/10.1145/75277.75299
12. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
13. Emmes, F., Enger, T., Giesl, J.: Proving non-looping non-termination automatically. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 225–240. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_19
14. Fuhs, C., Kop, C.: Harnessing first order termination provers using higher order dependency pairs. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS (LNAI), vol. 6989, pp. 147–162. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24364-6_11
15. Fuhs, C., Kop, C.: Polynomial interpretations for higher-order rewriting. In: Tiwari, A. (ed.) 23rd International Conference on Rewriting Techniques and Applications (RTA 2012) , RTA 2012. LIPIcs, vol. 15, Nagoya, Japan, 28 May–2 June 2012. pp. 176–192. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012). https://doi.org/10.4230/LIPIcs.RTA.2012.176
16. Fuhs, C., Kop, C.: First-order formative rules. In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 240–256. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_17
17. Fuhs, C., Kop, C.: A static higher-order dependency pair framework (extended version). Technical report arXiv:1902.06733 [cs.LO], CoRR (2019)
18. Fuhs, C., Kop, C., Nishida, N.: Verifying procedural programs via constrained rewriting induction. ACM Trans. Comput. Logic **18**(2), 14:1–14:50 (2017). https://doi.org/10.1145/3060143
19. Giesl, J., et al.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reasoning **58**(1), 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y
20. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32275-7_21

21. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: Gramlich, B. (ed.) FroCoS 2005. LNCS (LNAI), vol. 3717, pp. 216–231. Springer, Heidelberg (2005). https://doi.org/10.1007/11559306_12
22. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. J. Autom. Reasoning **37**(3), 155–203 (2006). https://doi.org/10.1007/s10817-006-9057-7
23. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12251-4_9
24. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool: techniques and features. Inf. Comput. **205**(4), 474–511 (2007). https://doi.org/10.1016/j.ic.2006.08.010
25. Hoe, J.C., Arvind: Hardware synthesis from term rewriting systems. In: Silveira, L.M., Devadas, S., Reis, R. (eds.) VLSI: Systems on a Chip. IFIPAICT, vol. 34, pp. 595–619. Springer, Boston (2000). https://doi.org/10.1007/978-0-387-35498-9_52
26. Jouannaud, J., Rubio, A.: The higher-order recursive path ordering. In: 14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, 2–5 July 1999, pp. 402–411. IEEE Computer Society (1999). https://doi.org/10.1109/LICS.1999.782635
27. Klop, J., Oostrom, V.V., Raamsdonk, F.V.: Combinatory reduction systems: introduction and survey. Theor. Comput. Sci. **121**(1–2), 279–308 (1993). https://doi.org/10.1016/0304-3975(93)90091-7
28. Kop, C.: WANDA - a higher-order termination tool. http://wandahot.sourceforge.net/
29. Kop, C.: Higher order termination. Ph.D. thesis, VU Amsterdam (2012)
30. Kop, C., van Raamsdonk, F.: Higher order dependency pairs for algebraic functional systems. In: Schmidt-Schauß, M. (ed.) Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011. LIPIcs, vol. 10, Novi Sad, Serbia, 30 May–1 June 2011, pp. 203–218. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011). https://doi.org/10.4230/LIPIcs.RTA.2011.203
31. Kop, C., van Raamsdonk, F.: Dynamic dependency pairs for algebraic functional systems. Logical Methods Comput. Sci. **8**(2), 10:1–10:51 (2012). https://doi.org/10.2168/LMCS-8(2:10)2012
32. Kusakari, K.: Static dependency pair method in rewriting systems for functional programs with product, algebraic data, and ML-polymorphic types. IEICE Trans. **96-D**(3), 472–480 (2013). https://doi.org/10.1587/transinf.E96.D.472
33. Kusakari, K.: Static dependency pair method in functional programs. IEICE Trans. Inf. Syst. **E101.D**(6), 1491–1502 (2018). https://doi.org/10.1587/transinf.2017FOP0004
34. Kusakari, K., Isogai, Y., Sakai, M., Blanqui, F.: Static dependency pair method based on strong computability for higher-order rewrite systems. IEICE Trans. Inf. Syst. **92**(10), 2007–2015 (2009). https://doi.org/10.1587/transinf.E92.D.2007
35. Kusakari, K., Nakamura, M., Toyama, Y.: Argument filtering transformation. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 47–61. Springer, Heidelberg (1999). https://doi.org/10.1007/10704567_3
36. Meadows, C.A.: Applying formal methods to the analysis of a key management protocol. J. Comput. Secur. **1**(1), 5–36 (1992). https://doi.org/10.3233/JCS-1992-1102

37. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. J. Logic Comput. **1**(4), 497–536 (1991). https://doi.org/10.1093/logcom/1.4.497
38. Nagele, J.: CoCo 2018 participant: CSI^ho 0.2 (2018). http://project-coco.uibk.ac.at/2018/papers/csiho.pdf
39. Nipkow, T.: Higher-order critical pairs. In: Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS 1991), Amsterdam, The Netherlands, 15–18 July 1991, pp. 342–349. IEEE Computer Society (1991). https://doi.org/10.1109/LICS.1991.151658
40. Onozawa, K., Kikuchi, K., Aoto, T., Toyama, Y.: ACPH: system description for CoCo 2017 (2017). http://project-coco.uibk.ac.at/2017/papers/acph.pdf
41. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of Java Bytecode by term rewriting. In: Lynch, C. (ed.) Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010. LIPIcs, vol. 6, Edinburgh, Scottland, UK, 11–13 July 2010, pp. 259–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010). https://doi.org/10.4230/LIPIcs.RTA.2010.259
42. Payet, É.: Loop detection in term rewriting using the eliminating unfoldings. Theor. Comput. Sci. **403**(2–3), 307–327 (2008). https://doi.org/10.1016/j.tcs.2008.05.013
43. van de Pol, J.: Termination of higher-order rewrite systems. Ph.D. thesis, University of Utrecht (1996)
44. Sakai, M., Kusakari, K.: On dependency pair method for proving termination of higher-order rewrite systems. IEICE Trans. Inf. Syst. **E88-D**(3), 583–593 (2005)
45. Sakai, M., Watanabe, Y., Sakabe, T.: An extension of the dependency pair method for proving termination of higher-order rewrite systems. IEICE Trans. Inf. Syst. **E84-D**(8), 1025–1032 (2001)
46. Suzuki, S., Kusakari, K., Blanqui, F.: Argument filterings and usable rules in higher-order rewrite systems. IPSJ Trans. Program. **4**(2), 1–12 (2011)
47. Tait, W.: Intensional interpretation of functionals of finite type. J. Symbolic Logic **32**(2), 187–199 (1967)
48. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
49. Wiki: Termination Problems DataBase (TPDB). http://termination-portal.org/wiki/TPDB
50. Wiki: The International Termination Competition (TermComp) (2018). http://termination-portal.org/wiki/Termination_Competition

Coinduction in Uniform: Foundations for Corecursive Proof Search with Horn Clauses

Henning Basold[1(✉)], Ekaterina Komendantskaya[2(✉)], and Yue Li[2]

[1] CNRS, ENS Lyon, Lyon, France
henning.basold@ens-lyon.fr
[2] Heriot-Watt University, Edinburgh, UK
{ek19,yl55}@hw.ac.uk

Abstract. We establish proof-theoretic, constructive and coalgebraic foundations for proof search in coinductive Horn clause theories. Operational semantics of coinductive Horn clause resolution is cast in terms of *coinductive uniform proofs*; its constructive content is exposed via soundness relative to an intuitionistic first-order logic with recursion controlled by the later modality; and soundness of both proof systems is proven relative to a novel coalgebraic description of complete Herbrand models.

Keywords: Horn clause logic · Coinduction · Uniform proofs · Intuitionistic logic · Coalgebra · Fibrations · Löb modality

1 Introduction

Horn clause logic is a Turing complete and constructive fragment of first-order logic, that plays a central role in verification [22], automated theorem proving [52,53,57] and type inference. Examples of the latter can be traced from the Hindley-Milner type inference algorithm [55,73], to more recent uses of Horn clauses in Haskell type classes [26,51] and in refinement types [28,43]. Its popularity can be attributed to well-understood fixed point semantics and an efficient semi-decidable resolution procedure for automated proof search.

According to the standard fixed point semantics [34,52], given a set P of Horn clauses, the *least Herbrand model* for P is the set of all (finite) ground atomic formulae *inductively entailed* by P. For example, the two clauses below define the set of natural numbers in the least Herbrand model.

$$\kappa_{\mathbf{nat}0} : \mathbf{nat}\,0$$
$$\kappa_{\mathbf{nat}s} : \forall x.\, \mathbf{nat}\,x \to \mathbf{nat}\,(s\,x)$$

This work is supported by the European Research Council (ERC) under the EU's Horizon 2020 programme (CoVeCe, grant agreement No. 678157) and by the EPSRC research grants EP/N014758/1, EP/K031864/1-2.

L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 783–813, 2019.
https://doi.org/10.1007/978-3-030-17184-1_28

Formally, the least Herbrand model for the above two clauses is the set of ground atomic formulae obtained by taking a (forward) closure of the above two clauses. The model for **nat** is given by $\mathcal{N} = \{\mathbf{nat}\,0,\ \mathbf{nat}\,(s\,0),\ \mathbf{nat}\,(s\,(s\,0)), \ldots\}$.

We can also view Horn clauses coinductively. The *greatest complete Herbrand model* for a set P of Horn clauses is the largest set of finite and infinite ground atomic formulae *coinductively entailed* by P. For example, the greatest complete Herbrand model for the above two clauses is the set

$$\mathcal{N}^\infty = \mathcal{N} \cup \{\mathbf{nat}\,(s\,(s\,(\cdots)))\},$$

obtained by taking a backward closure of the above two inference rules on the set of all finite and infinite ground atomic formulae. The *greatest Herbrand model* is the largest set of *finite* ground atomic formulae *coinductively entailed* by P. In our example, it would be given by $\mathcal{N}$ already. Finally, one can also consider the *least complete Hebrand model*, which interprets entailment inductively but over potentially infinite terms. In the case of **nat**, this interpretation does not differ from $\mathcal{N}$. However, finite paths in coinductive structures like transition systems, for example, require such semantics.

The need for coinductive semantics of Horn clauses arises in several scenarios: the Horn clause theory may explicitly define a coinductive data structure or a coinductive relation. However, it may also happen that a Horn clause theory, which is not explicitly intended as coinductive, nevertheless gives rise to infinite inference by resolution and has an interesting coinductive model. This commonly happens in type inference. We will illustrate all these cases by means of examples.

Horn Clause Theories as Coinductive Data Type Declarations. The following clause defines, together with $\kappa_{\mathbf{nat}0}$ and $\kappa_{\mathbf{nat}s}$, the type of streams over natural numbers.

$$\kappa_{\mathbf{stream}} : \forall xy.\, \mathbf{nat}\,x \wedge \mathbf{stream}\,y \rightarrow \mathbf{stream}\,(\mathrm{scons}\,x\,y)$$

This Horn clause does not have a meaningful inductive, i.e. least fixed point, model. The greatest Herbrand model of the clauses is given by

$$\mathcal{S} = \mathcal{N}^\infty \cup \{\mathbf{stream}(\mathrm{scons}\,x_0\,(\mathrm{scons}\,x_1\,\cdots)) \mid \mathbf{nat}\,x_0, \mathbf{nat}\,x_1, \ldots \in \mathcal{N}^\infty\}$$

In trying to prove, for example, the goal $(\mathbf{stream}\,x)$, a goal-directed proof search may try to find a substitution for x that will make $(\mathbf{stream}\,x)$ valid relative to the coinductive model of this set of clauses. This search by resolution may proceed by means of an infinite reduction $\underline{\mathbf{stream}\,x} \overset{\kappa_{\mathbf{stream}}:[\mathrm{scons}\ y\ x'/x]}{\rightsquigarrow} \mathbf{nat}\,y \wedge \mathbf{stream}\,x' \overset{\kappa_{\mathbf{nat}0}:[0/y]}{\rightsquigarrow} \underline{\mathbf{stream}\,x'} \overset{\kappa_{\mathbf{stream}}:[\mathrm{scons}\ y'\ x''/x']}{\rightsquigarrow} \cdots$, thereby generating a stream Z of zeros via composition of the computed substitutions: $Z = (\mathrm{scons}\,0\,x')[\mathrm{scons}\,0\,x''/x'] \cdots$. Above, we annotated each resolution step with the label of the clause it resolves against and the computed substitution. A method to compute an answer for this infinite sequence of reductions was given by Gupta et al. [41] and Simon et al. [69]: the underlined loop gives rise to the

circular unifier $x = \text{scons}\,0\,x$ that corresponds to the infinite term Z. It is proven that, if a loop and a corresponding circular unifier are detected, they provide an answer that is sound relative to the greatest complete Herbrand model of the clauses. This approach is known under the name of CoLP.

Horn Clause Theories in Type Inference. Below clauses give the typing rules of the simply typed λ-calculus, and may be used for type inference or type checking:

$$\kappa_{t1} : \forall x\,\Gamma\,a.\,\mathbf{var}\,x \wedge \mathbf{find}\,\Gamma\,x\,a \;\rightarrow \mathbf{typed}\,\Gamma\,x\,a$$
$$\kappa_{t2} : \forall x\,\Gamma\,a\,m\,b.\,\mathbf{typed}\,[x:a|\Gamma]\,m\,b \rightarrow \mathbf{typed}\,\Gamma\,(\lambda\,x\,m)\,(a \rightarrow b)$$
$$\kappa_{t3} : \forall \Gamma\,a\,m\,n\,b.\,\mathbf{typed}\,\Gamma\,m\,(a \rightarrow b) \wedge \mathbf{typed}\,\Gamma\,n\,a \rightarrow \mathbf{typed}\,\Gamma\,(\text{app}\,m\,n)\,b$$

It is well known that the Y-combinator is not typable in the simply-typed λ-calculus and, in particular, self-application $\lambda x.\,x\,x$ is not typable either. However, by switching off the occurs-check in Prolog or by allowing circular unifiers in CoLP [41,69], we can resolve the goal "$\mathbf{typed}\,[]\,(\lambda\,x\,(\text{app}\,x\,x))\,a$" and would compute the circular substitution: $a = b \rightarrow c, b = b \rightarrow c$ suggesting that an infinite, or circular, type may be able to type this λ-term. A similar trick would provide a typing for the Y-combinator. Thus, a coinductive interpretation of the above Horn clauses yields a theory of infinite types, while an inductive interpretation corresponds to the standard type system of the simply typed λ-calculus.

Horn Clause Theories in Type Class Inference. Haskell type class inference does not require circular unifiers but may require a cyclic resolution inference [37,51]. Consider, for example, the following mutually defined data structures in Haskell.

```
data OddList  a = OCons a (EvenList a)
data EvenList a = Nil | ECons a (OddList a)
```

This type declaration gives rise to the following equality class instance declarations, where we leave the, here irrelevant, body out.

```
instance (Eq a, Eq (EvenList a)) => Eq (OddList a) where
instance (Eq a, Eq (OddList a)) => Eq (EvenList a) where
```

The above two type class instance declarations have the shape of Horn clauses. Since the two declarations mutually refer to each other, an instance inference for, e.g., **Eq** (OddList **Int**) will give rise to an infinite resolution that alternates between the subgoals **Eq** (OddList **Int**) and **Eq** (EvenList **Int**). The solution is to terminate the computation as soon as the cycle is detected [51], and this method has been shown sound relative to the greatest Herbrand models in [36]. We will demonstrate this later in the proof systems proposed in this paper.

The diversity of these coinductive examples in the existing literature shows that there is a practical demand for coinductive methods in Horn clause logic, but it also shows that no unifying proof-theoretic approach exists to allow for a generic use of these methods. This causes several problems.

Problem 1. The existing proof-theoretic coinductive interpretations of cycle and loop detection are unclear, incomplete and not uniform.

Table 1. Examples of greatest (complete) Herbrand models for Horn clauses γ_1, γ_2, γ_3. The signatures are $\{a\}$ for the clause γ_1 and $\{a, f\}$ for the others.

Horn clauses	$\gamma_1 : \forall x.\, p\, x \to p\, x$	$\gamma_2 : \forall x.\, p(f\, x) \to p\, x$	$\gamma_3 : \forall x.\, p\, x \to p(f\, x)$
Greatest Herbrand model:	$\{p\, a\}$	$\{p(a), p(f\, a), p(f(f\, a)), \ldots\}$	$\emptyset$
Greatest complete Herbrand model:	$\{p\, a\}$	$\{p(a), p(f\, a), p(f(f\, a), \ldots, p(f(f \ldots)\}$	$\{p(f(f \ldots)\}$
CoLP substitution for query $p\, a$	id	fails	fails
CoLP substitution for query $p\, x$	id	$x = f\, x$	$x = f\, x$

To see this, consider Table 1, which exemplifies three kinds of circular phenomena in Horn clauses: The clause γ_1 is the easiest case. Its coinductive models are given by the finite set $\{p\, a\}$. On the other extreme is the clause γ_3 that, just like $\kappa_{\mathbf{stream}}$, admits only an infinite formula in its coinductive model. The intermediate case is γ_2, which could be interpreted by an infinite set of finite formulae in its greatest Herbrand model, or may admit an infinite formula in its greatest complete Herbrand model. Examples like γ_1 appear in Haskell type class resolution [51], and examples like γ_2 in its experimental extensions [37]. Cycle detection would only cover computations for γ_1, whereas γ_2, γ_3 require some form of loop detection[1]. However, CoLP's loop detection gives confusing results here. It correctly fails to infer $p\, a$ from γ_3 (no unifier for subgoals $p\, a$ and $p\,(f\, a)$ exists), but incorrectly fails to infer $p\, a$ from γ_2 (also failing to unify $p\, a$ and $p\,(f\, a)$). The latter failure is misleading bearing in mind that $p\, a$ is in fact in the coinductive model of γ_2. Vice versa, if we interpret the CoLP answer $x = f\, x$ as a declaration of an infinite term $(f\, f\, \ldots)$ in the model, then CoLP's answer for γ_3 and $p\, x$ is exactly correct, however the same answer is badly incomplete for the query involving $p\, x$ and γ_2, because γ_2 in fact admits other, finite, formulae in its models. And in some applications, e.g. in Haskell type class inference, a finite formula would be the only acceptable answer for any query to γ_2.

This set of examples shows that loop detection is too coarse a tool to give an operational semantics to a diversity of coinductive models.

Problem 2. Constructive interpretation of coinductive proofs in Horn clause logic is unclear. Horn clause logic is known to be a constructive fragment of FOL. Some applications of Horn clauses rely on this property in a crucial way. For example, inference in Haskell type class resolution is constructive: when a certain formula F is inferred, the Haskell compiler in fact constructs a proof term that inhabits F seen as type. In our earlier example **Eq** (OddList **Int**) of the Haskell type classes, Haskell in fact captures the cycle by a fixpoint term t and proves that t inhabits the type **Eq** (OddList **Int**).

[1] We follow the standard terminology of [74] and say that two formulae F and G form a cycle if $F = G$, and a loop if $F[\theta] = G[\theta]$ for some (possibly circular) unifier θ.

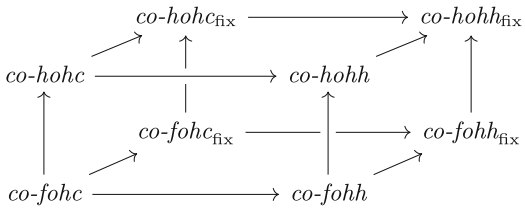


Fig. 1. Cube of logics covered by CUP

Although we know from [36] that these computations are sound relative to greatest Herbrand models of Horn clauses, the results of [36] do not extend to Horn clauses like γ_3 or $\kappa_{\mathbf{stream}}$, or generally to Horn clauses modelled by the greatest *complete* Herbrand models. This shows that there is not just a need for coinductive proofs in Horn clause logic, but *constructive* coinductive proofs.

Problem 3. Incompleteness of circular unification for irregular coinductive data structures. Table 1 already showed some issues with incompleteness of circular unification. A more famous consequence of it is the failure of circular unification to capture irregular terms. This is illustrated by the following Horn clause, which defines the infinite stream of successive natural numbers.

$$\kappa_{\mathbf{from}} : \forall x\, y.\, \mathbf{from}\,(s\,x)\,y \rightarrow \mathbf{from}\,x\,(\mathrm{scons}\,x\,y)$$

The reductions for $\mathbf{from}\,0\,y$ consist only of irregular (non-unifiable) formulae:

$$\mathbf{from}\ 0\ y \overset{\kappa_{\mathbf{from}}:[\mathrm{scons}\ 0\ y'/y]}{\rightsquigarrow} \mathbf{from}\ (s\,0)\ y' \overset{\kappa_{\mathbf{from}}:[\mathrm{scons}\ (s\,0)\ y''/y']}{\rightsquigarrow} \cdots$$

The composition of the computed substitutions would suggest an infinite term as answer: $\mathbf{from}\,0\,(\mathrm{scons}\ 0\ (\mathrm{scons}\ (s\,0)\ \ldots))$. However, circular unification no longer helps to compute this answer, and CoLP fails. Thus, there is a need for more general operational semantics that allows irregular coinductive structures.

A New Theory of Coinductive Proof Search in Horn Clause Logic

In this paper, we aim to give a principled and *general* theory that resolves the three problems above. This theory establishes a *constructive* foundation for coinductive resolution and allows us to give proof-theoretic characterisations of the approaches that have been proposed throughout the literature.

To solve Problem 1, we follow the footsteps of the *uniform proofs* by Miller et al. [53,54], who gave a general proof-theoretic account of resolution in first-order Horn clause logic (*fohc*) and three extensions: first-order hereditary Harrop clauses (*fohh*), higher-order Horn clauses (*hohc*), and higher-order hereditary Harrop clauses (*hohh*). In Sect. 3, we extend uniform proofs with a general coinduction proof principle. The resulting framework is called *coinductive uniform proofs (CUP)*. We show how the coinductive extensions of the four logics of Miller et al., which we name *co-fohc*, *co-fohh*, *co-hohc* and *co-hohh*, give a precise

proof-theoretic characterisation to the different kinds of coinduction described in the literature. For example, coinductive proofs involving the clauses γ_1 and γ_2 belong to *co-fohc* and *co-fohh*, respectively. However, proofs involving clauses like γ_3 or $\kappa_{\mathbf{stream}}$ require in addition fixed point terms to express infinite data. These extentions are denoted by *co-fohc*$_{\text{fix}}$, *co-fohh*$_{\text{fix}}$, *co-hohc*$_{\text{fix}}$ and *co-hohh*$_{\text{fix}}$.

Section 3 shows that this yields the cube in Fig. 1, where the arrows show the increase in logical strength. The invariant search for regular infinite objects done in CoLP is fully described by the logic *co-fohc*$_{\text{fix}}$, including proofs for clauses like γ_3 and $\kappa_{\mathbf{stream}}$. An important consequence is that CUP is complete for γ_1, γ_2, and γ_3, e.g. $p\,a$ is provable from γ_2 in CUP, but not in CoLP.

In tackling Problem 3, we will find that the irregular proofs, such as those for $\kappa_{\mathbf{from}}$, can be given in *co-hohh*$_{\text{fix}}$. The stream of successive numbers can be defined as a higher-order fixed point term $s_{\text{fr}} = \text{fix}\, f.\, \lambda x.\, \text{scons}\, x\, (f\, (s\, x))$, and the proposition $\forall x.\, \mathbf{from}\, x\, (s_{\text{fr}}\, x)$ is provable in *co-hohh*$_{\text{fix}}$. This requires the use of higher-order syntax, fixed point terms and the goals of universal shape, which become available in the syntax of Hereditary Harrop logic.

In order to solve Problem 2 and to expose the constructive nature of the resulting proof systems, we present in Sect. 4 a coinductive extension of first-order intuitionistic logic and its sequent calculus. This extension (**iFOL**$_{\blacktriangleright}$) is based on the so-called later modality (or Löb modality) known from provability logic [16,71], type theory [8,58] and domain theory [20]. However, our way of using the later modality to control recursion in first-order proofs is new and builds on [13,14]. In the same section we also show that CUP is sound relative to **iFOL**$_{\blacktriangleright}$, which gives us a handle on the constructive content of CUP. This yields, among other consequences, a constructive interpretation of CoLP proofs.

Section 5 is dedicated to showing soundness of both coinductive proof systems relative to *complete Herbrand models* [52]. The construction of these models is carried out by using coalgebras and category theory. This frees us from having to use topological methods and will simplify future extensions of the theory to, e.g., encompass typed logic programming. It also makes it possible to give original and constructive proofs of soundness for both CUP and **iFOL**$_{\blacktriangleright}$ in Sect. 5. We finish the paper with discussion of related and future work.

Originality of the Contribution

The results of this paper give a comprehensive characterisation of coinductive Horn clause theories from the point of view of proof search (by expressing coinductive proof search and resolution as coinductive uniform proofs), constructive proof theory (via a translation into an intuitionistic sequent calculus), and coalgebraic semantics (via coinductive Herbrand models and constructive soundness results). Several of the presented results have never appeared before: the coinductive extension of uniform proofs; characterisation of coinductive properties of Horn clause theories in higher-order logic with and without fixed point operators; coalgebraic and fibrational view on complete Herbrand models; and soundness of an intuitionistic logic with later modality relative to complete Herbrand models.

2 Preliminaries: Terms and Formulae

In this section, we set up notation and terminology for the rest of the paper. Most of it is standard, and blends together the notation used in [53] and [11].

Definition 1. We define the sets $\mathbb{T}$ of *types* and $\mathbb{P}$ of *proposition types* by the following grammars, where ι and o are the *base type* and *base proposition type*.

$$\mathbb{T} \ni \sigma, \tau ::= \iota \mid \sigma \to \tau \qquad \mathbb{P} \ni \rho ::= o \mid \sigma \to \rho, \quad \sigma \in \mathbb{T}$$

We adapt the usual convention that $\to$ binds to the right.

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.\, M : \sigma \to \tau} \qquad \frac{\Gamma, x : \tau \vdash M : \tau}{\Gamma \vdash \mathrm{fix}\, x.\, M : \tau}$$

Fig. 2. Well-formed terms

$$\frac{(p : \tau_1 \to \cdots \to \tau_n \to o) \in \Pi \qquad \Gamma \vdash M_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash M_n : \tau_n}{\Gamma \Vdash p\ M_1 \cdots M_n}$$

$$\frac{}{\Gamma \Vdash \top} \qquad \frac{\Gamma \Vdash \varphi \qquad \Gamma \Vdash \psi \qquad \square \in \{\wedge, \vee, \to\}}{\Gamma \Vdash \varphi \square \psi} \qquad \frac{\Gamma, x : \tau \Vdash \varphi}{\Gamma \Vdash \forall x : \tau.\, \varphi} \qquad \frac{\Gamma, x : \tau \Vdash \varphi}{\Gamma \Vdash \exists x : \tau.\, \varphi}$$

Fig. 3. Well-formed formulae

Definition 2. A *term signature* Σ is a set of pairs $c : \tau$, where $\tau \in \mathbb{T}$, and a *predicate signature* is a set Π of pairs $p : \rho$ with $\rho \in \mathbb{P}$. The elements in Σ and Π are called *term symbols* and *predicate symbols*, respectively. Given term and predicate signatures Σ and Π, we refer to the pair (Σ, Π) as *signature*. Let Var be a countable set of variables, the elements of which we denote by $x, y, \ldots$ We call a finite list Γ of pairs $x : \tau$ of variables and types a *context*. The set Λ_Σ of *(well-typed) terms* over Σ is the collection of all M with $\Gamma \vdash M : \tau$ for some context Γ and type $\tau \in \mathbb{T}$, where $\Gamma \vdash M : \tau$ is defined inductively in Fig. 2. A term is called *closed* if $\vdash M : \tau$, otherwise it is called *open*. Finally, we let Λ_Σ^- denote the set of all terms M that do not involve fix.

Definition 3. Let (Σ, Π) be a signature. We say that φ is a *(first-order) formula* in context Γ, if $\Gamma \Vdash \varphi$ is inductively derivable from the rules in Fig. 3.

Definition 4. The *reduction relation* $\longrightarrow$ on terms in Λ_Σ is given as the compatible closure (reduction under applications and binders) of β- and fix-reduction:

$$(\lambda x.\, M)N \longrightarrow M\,[N/x] \qquad \mathrm{fix}\, x.\, M \longrightarrow M\,[\mathrm{fix}\, x.\, M/x]$$

We denote the reflexive, transitive closure of $\longrightarrow$ by $\longrightarrow\!\!\!\!\!\rightarrow$. Two terms M and N are called *convertible*, if $M \equiv N$, where $\equiv$ is the equivalence closure of $\longrightarrow$. Conversion of terms extends to formulae in the obvious way: if $M_k \equiv M'_k$ for $k = 1, \ldots, n$, then $p\, M_1 \cdots M_n \equiv p\, M'_1 \cdots\, M'_n$.

We will use in the following that the above calculus features subject reduction and confluence, cf. [61]: if $\Gamma \vdash M : \tau$ and $M \equiv N$, then $\Gamma \vdash N : \tau$; and $M \equiv N$ iff there is a term P, such that $M \longrightarrow\!\!\!\!\!\rightarrow P$ and $N \longrightarrow\!\!\!\!\!\rightarrow P$.

The *order* of a type $\tau \in \mathbb{T}$ is given as usual by $\mathrm{ord}(\iota) = 0$ and $\mathrm{ord}(\sigma \to \tau) = \max\{\mathrm{ord}(\sigma) + 1, \mathrm{ord}(\tau)\}$. If $\mathrm{ord}(\tau) \leq 1$, then the arity of τ is given by $\mathrm{ar}(\iota) = 0$ and $\mathrm{ar}(\iota \to \tau) = \mathrm{ar}(\tau) + 1$. A signature Σ is called *first-order*, if for all $f : \tau \in \Sigma$ we have $\mathrm{ord}(\tau) \leq 1$. We let the arity of f then be $\mathrm{ar}(\tau)$ and denote it by $\mathrm{ar}(f)$.

Definition 5. The set of *guarded base terms* over a first-order signature Σ is given by the following type-driven rules.

$$\frac{x : \tau \in \Gamma \qquad \mathrm{ord}(\tau) \leq 1}{\Gamma \vdash_g x : \tau} \qquad \frac{f : \tau \in \Sigma}{\Gamma \vdash_g f : \tau} \qquad \frac{\Gamma \vdash_g M : \sigma \to \tau \qquad \Gamma \vdash_g N : \sigma}{\Gamma \vdash_g M\, N : \tau}$$

$$\frac{f : \sigma \in \Sigma \qquad \mathrm{ord}(\tau) \leq 1 \qquad \Gamma, x : \tau, y_1 : \iota, \ldots, y_{\mathrm{ar}(\tau)} : \iota \vdash_g M_i : \iota \qquad 1 \leq i \leq \mathrm{ar}(f)}{\Gamma \vdash_g \mathrm{fix}\, x.\, \lambda \vec{y}.\, f\, \vec{M} : \tau}$$

General *guarded terms* are terms M, such that all fix-subterms are guarded base terms, which means that they are generated by the following grammar.

$$G ::= M \text{ (with } \vdash_g M : \tau \text{ for some type } \tau) \mid c \in \Sigma \mid x \in \mathrm{Var} \mid G\, G \mid \lambda x.G$$

Finally, M is a *first-order* term over Σ with $\Gamma \vdash M : \tau$ if $\mathrm{ord}(\tau) \leq 1$ and the types of all variables occurring in Γ are of order 0. We denote the set of guarded first-order terms M with $\Gamma \vdash M : \iota$ by $\Lambda^{G,1}_\Sigma(\Gamma)$ and the set of guarded terms in Γ by $\Lambda^{G}_\Sigma(\Gamma)$. If Γ is empty, we just write $\Lambda^{G,1}_\Sigma$ and Λ^{G}_Σ, respectively.

Note that an important aspect of guarded terms is that no free variable occurs under a fix-operator. *Guarded base terms* should be seen as specific fixed point terms that we will be able to unfold into potentially infinite trees. *Guarded terms* close guarded base terms under operations of the simply typed λ-calculus.

Example 6. Let us provide a few examples that illustrate (first-order) guarded terms. We use the first-order signature $\Sigma = \{\mathrm{scons} \colon \iota \to \iota \to \iota, s \colon \iota \to \iota, 0 : \iota\}$.

1. Let $s_{\mathrm{fr}} = \mathrm{fix}\, f.\, \lambda x.\, \mathrm{scons}\ x\ (f\ \ (s\ \ x))$ be the function that computes the streams of numerals starting at the given argument. It is easy to show that $\vdash_g s_{\mathrm{fr}} : \iota \to \iota$ and so $s_{\mathrm{fr}}\ 0 \in \Lambda^{G,1}_\Sigma$.

2. For the same signature Σ we also have $x : \iota \vdash_g x : \iota$. Thus $x \in \Lambda^{G,1}_{\Sigma}(x : \iota)$ and $s\ x \in \Lambda^{G,1}_{\Sigma}(x : \iota)$.
3. We have $x : \iota \to \iota \vdash_g x\ 0 : \iota$, but $(x\ 0) \notin \Lambda^{G,1}_{\Sigma}(x : \iota \to \iota)$.

The purpose of guarded terms is that these are productive, that is, we can reduce them to a term that either has a function symbol at the root or is just a variable. In other words, guarded terms have head normal forms: We say that a term M is in *head normal form*, if $M = f\ \vec{N}$ for some $f \in \Sigma$ or if $M = x$ for some variable x. The following lemma is a technical result that is needed to show in Lemma 8 that all guarded terms have a head normal form.

Lemma 7. *Let M and N be guarded base terms with $\Gamma, x : \sigma \vdash_g M : \tau$ and $\Gamma \vdash_g N : \sigma$. Then $M\,[N/x]$ is a guarded base term with $\Gamma \vdash_g M\,[N/x] : \tau$.*

Lemma 8. *If M is a first-order guarded term with $M \in \Lambda^{G,1}_{\Sigma}(\Gamma)$, then M reduces to a unique head normal form. This means that either (i) there is a unique $f \in \Sigma$ and terms $N_1, \ldots, N_{\mathrm{ar}(f)}$ with $\Gamma \vdash_g N_k : \iota$ and $M \longrightarrow\!\!\!\!\!\rightarrow f\ \vec{N}$, and for all L if $M \longrightarrow\!\!\!\!\!\rightarrow f\ \vec{L}$, then $\vec{N} \equiv \vec{L}$; or (ii) $M \longrightarrow\!\!\!\!\!\rightarrow x$ for some $x : \iota \in \Gamma$.*

We end this section by introducing the notion of an atom and refinements thereof. This will enable us to define the different logics and thereby to analyse the strength of coinduction hypotheses, which we promised in the introduction.

Definition 9. A formula φ of the shape $\top$ or $p\ M_1 \cdots M_n$ is an *atom* and a

- *first-order atom*, if p and all the terms M_i are first-order;
- *guarded atom*, if all terms M_i are guarded; and
- *simple atom*, if all terms M_i are non-recursive, that is, are in Λ^{-}_{Σ}.

First-order, guarded and simple atoms are denoted by At_1, At^g_ω and At^s_ω. We denote conjunctions of these predicates by $\mathrm{At}^g_1 = \mathrm{At}_1 \cap \mathrm{At}^g_\omega$ and $\mathrm{At}^s_1 = \mathrm{At}_1 \cap \mathrm{At}^s_\omega$.

Note that the restriction for At^g_ω only applies to fixed point terms. Hence, any formula that contains terms without fix is already in At^g_ω and $\mathrm{At}^g_\omega \cap \mathrm{At}^s_\omega = \mathrm{At}^s_\omega$. Since these notions are rather subtle, we give a few examples

Example 10. We list three examples of first-order atoms.

1. For $x : \iota$ we have $\mathbf{stream}\ x \in \mathrm{At}_1$, but there are also "garbage" formulae like "$\mathbf{stream}\ (\mathrm{fix}\, x.\, x)$" in At_1. Examples of atoms that are not first-order are $p\ M$, where $p : (\iota \to \iota) \to o$ or $x : \iota \to \iota \vdash M : \tau$.
2. Our running example "$\mathbf{from}\ 0\ (s_{\mathrm{fr}}\ 0)$" is a first-order guarded atom in At^g_1.
3. The formulae in At^s_1 may not contain recursion and higher-order features. However, the atoms of Horn clauses in a logic program fit in here.

3 Coinductive Uniform Proofs

This section introduces the eight logics of the coinductive uniform proof framework announced and motivated in the introduction. The major difference of uniform proofs with, say, a sequent calculus is the "uniformity" property, which means that the choice of the application of each proof rule is deterministic and all proofs are in normal form (cut free). This subsumes the operational semantics of resolution, in which the proof search is always goal directed. Hence, the main challenge, that we set out to solve in this section, is to extend the uniform proof framework with coinduction, while preserving this valuable operational property.

We begin by introducing the different goal formulae and definite clauses that determine the logics that were presented in the cube for coinductive uniform proofs in the introduction. These clauses and formulae correspond directly to those of the original work on uniform proofs [53] with the only difference being that we need to distinguish atoms with and without fixed point terms. The general idea is that goal formulae (G-formulae) occur on the right of a sequent, thus are the *goal* to be proved. Definite clauses (D-formulae), on the other hand, are selected from the context as assumptions. This will become clear once we introduce the proof system for coinductive uniform proofs.

Definition 11. Let D_i be generated by the following grammar with $i \in \{1, \omega\}$.

$$D_i ::= \mathrm{At}_i^s \mid G \to D \mid D \wedge D \mid \forall x : \tau.\, D$$

Table 2. D- and G-formulae for coinductive uniform proofs.

	Definite Clauses	Goals
co-fohc	D_1	$G ::= \mathrm{At}_1^s \mid G \wedge G \mid G \vee G \mid \exists x : \tau.\, G$
co-hohc	D_ω	$G ::= \mathrm{At}_\omega^s \mid G \wedge G \mid G \vee G \mid \exists x : \tau.\, G$
co-fohh	D_1	$G ::= \mathrm{At}_1^s \mid G \wedge G \mid G \vee G \mid \exists x : \tau.\, G \mid D \to G \mid \forall x : \tau.\, G$
co-hohh	D_ω	$G ::= \mathrm{At}_\omega^s \mid G \wedge G \mid G \vee G \mid \exists x : \tau.\, G \mid D \to G \mid \forall x : \tau.\, G$

The sets of definite clauses (D-formulae) and goals (G-formulae) of the four logics *co-fohc*, *co-fohh*, *co-hohc*, *co-hohh* are the well-formed formulae of the corresponding shapes defined in Table 2. For the variations *co-fohh*$_{\mathrm{fix}}$ etc. of these logics with fixed point terms, we replace upper index "s" with "g" everywhere in Table 2. A D-formula of the shape $\forall \vec{x}.\, A_1 \wedge \cdots \wedge A_n \to A_0$ is called *H-formula* or *Horn clause* if $A_k \in \mathrm{At}_1^s$, and *H^g-formula* if $A_k \in \mathrm{At}_1^g$. Finally, a *logic program* (or *program*) P is a set of H-formulae. Note that any set of D-formulae in *fohc* can be transformed into an intuitionistically equivalent set of H-formulae [53].

We are now ready to introduce the coinductive uniform proofs. Such proofs are composed of two parts: an outer coinduction that has to be at the root of a proof tree, and the usual the usual uniform proofs by Miller et al. [54]. The latter are restated in Fig. 4. Of special notice is the rule DECIDE that mimics the operational behaviour of resolution in logic programming, by choosing a clause D from the given program to resolve against. The coinduction is started by the rule CO-FIX in Fig. 5. Our proof system mimics the typical recursion with a guard condition found in coinductive programs and proofs [5,8,19,31,40]. This guardedness condition is formalised by applying the guarding modality $\langle \_ \rangle$ on the formula being proven by coinduction and the proof rules that allow us to distribute the guard over certain logical connectives, see Fig. 5. The guarding modality may be discharged only if the guarded goal was resolved against a clause in the initial program or any hypothesis, except for the coinduction hypotheses. This is reflected in the rule DECIDE$\langle\rangle$, where we may only pick a clause from P, and is in contrast to the rule DECIDE, in which we can pick *any* hypothesis. The proof may only terminate with the INITIAL step if the goal is no longer guarded.

Note that the CO-FIX rule introduces a goal as a new hypothesis. Hence, we have to require that this goal is also a definite clause. Since coinduction hypotheses play such an important role, they deserve a separate definition.

Definition 12. Given a language L from Table 2, a formula φ is a *coinduction goal* of L if φ simultaneously is a D- and a G-formula of L.

Note that the coinduction goals of *co-fohc* and *co-fohh* can be transformed into equivalent H- or H^g-formulae, since any coinduction goal is a D-formula.

Let us now formally introduce the coinductive uniform proof system.

$$\frac{\Sigma; P; \Delta \overset{D}{\Longrightarrow} A \quad D \in P \cup \Delta}{\Sigma; P; \Delta \Longrightarrow A}\ \text{DECIDE} \qquad \frac{A \equiv A'}{\Sigma; P; \Delta \overset{A'}{\Longrightarrow} A}\ \text{INITIAL} \qquad \frac{}{\Sigma; P; \Delta \Longrightarrow \top}\ \top R$$

$$\frac{\Sigma; P; \Delta \overset{D}{\Longrightarrow} A \qquad \Sigma; P; \Delta \Longrightarrow G}{\Sigma; P; \Delta \overset{G \to D}{\Longrightarrow} A}\ \to L \qquad \frac{\Sigma; P, D; \Delta \Longrightarrow G}{\Sigma; P; \Delta \Longrightarrow D \to G}\ \to R$$

$$\frac{\Sigma; P; \Delta \overset{D_x}{\Longrightarrow} A \qquad x \in \{1,2\}}{\Sigma; P; \Delta \overset{D_1 \wedge D_2}{\Longrightarrow} A}\ \wedge L \qquad \frac{\Sigma; P; \Delta \Longrightarrow G_1 \qquad \Sigma; P; \Delta \Longrightarrow G_2}{\Sigma; P; \Delta \Longrightarrow G_1 \wedge G_2}\ \wedge R$$

$$\frac{\Sigma; P; \Delta \overset{D[N/x]}{\Longrightarrow} A \qquad \varnothing \vdash_g N : \tau}{\Sigma; P; \Delta \overset{\forall x.\, D}{\Longrightarrow} A}\ \forall L \qquad \frac{c : \tau, \Sigma; P; \Delta \Longrightarrow G\,[c/x] \qquad c : \tau \notin \Sigma}{\Sigma; P; \Delta \Longrightarrow \forall x : \tau.\, G}\ \forall R$$

$$\frac{\Sigma; P; \Delta \Longrightarrow G\,[N/x] \qquad \varnothing \vdash_g N : \tau}{\Sigma; P; \Delta \Longrightarrow \exists x : \tau.\, G}\ \exists R \qquad \frac{\Sigma; P; \Delta \Longrightarrow G_x \qquad x \in \{1,2\}}{\Sigma; P; \Delta \Longrightarrow G_1 \vee G_2}\ \vee R$$

Fig. 4. Uniform proof rules

$$\frac{\Sigma; P; \varphi \Longrightarrow \langle\varphi\rangle}{\Sigma; P \curvearrowright \varphi}\ \text{CO-FIX}$$

$$\frac{\Sigma; P; \Delta \overset{D}{\Rightarrow} A \quad D \in P}{\Sigma; P; \Delta \Longrightarrow \langle A\rangle}\ \text{DECIDE}\langle\rangle \qquad \frac{c : \tau, \Sigma; P; \Delta \Longrightarrow \langle\varphi\,[c/x]\rangle \quad c : \tau \notin \Sigma}{\Sigma; P; \Delta \Longrightarrow \langle\forall x : \tau.\, \varphi\rangle}\ \forall R\langle\rangle$$

$$\frac{\Sigma; P; \Delta \Longrightarrow \langle\varphi_1\rangle \qquad \Sigma; P; \Delta \Longrightarrow \langle\varphi_2\rangle}{\Sigma; P; \Delta \Longrightarrow \langle\varphi_1 \wedge \varphi_2\rangle}\ \wedge R\langle\rangle \qquad \frac{\Sigma; P; \Delta, \varphi_1 \Longrightarrow \langle\varphi_2\rangle}{\Sigma; P; \Delta \Longrightarrow \langle\varphi_1 \to \varphi_2\rangle}\ \to R\langle\rangle$$

Fig. 5. Coinductive uniform proof rules

Definition 13. Let P and Δ be finite sets of, respectively, definite clauses and coinduction goals, over the signature Σ, and suppose that G is a goal and φ is a coinduction goal. A *sequent* is either a *uniform provability sequent* of the form $\Sigma; P; \Delta \Longrightarrow G$ or $\Sigma; P; \Delta \overset{D}{\Rightarrow} A$ as defined in Fig. 4, or it is a *coinductive uniform provability sequent* of the form $\Sigma; P \curvearrowright \varphi$ as defined in Fig. 5. Let L be a language from Table 2. We say that φ is *coinductively provable* in L, if P is a set of D-formulae in L, φ is a coinduction goal in L and $\Sigma; P \curvearrowright \varphi$ holds.

The logics we have introduced impose different syntactic restrictions on D- and G-formulae, and will therefore admit coinduction goals of different strength. This ability to explicitly use stronger coinduction hypotheses within a goal-directed search was missing in CoLP, for example. And it allows us to account for different coinductive properties of Horn clauses as described in the introduction. We finish this section by illustrating this strengthening.

The first example is one for the logic *co-fohc*, in which we illustrate the framework on the problem of type class resolution.

Example 14. Let us restate the Haskell type class inference problem discussed in the introduction in terms of Horn clauses:

$$\begin{aligned} \kappa_{\mathrm{i}} &: \mathbf{eq}\ \mathrm{i} \\ \kappa_{\mathrm{odd}} &: \forall x.\, \mathbf{eq}\ x \wedge \mathbf{eq}\ (\mathrm{even}\ x) \to \mathbf{eq}\ (\mathrm{odd}\ x) \\ \kappa_{\mathrm{even}} &: \forall x.\, \mathbf{eq}\ x \wedge \mathbf{eq}\ (\mathrm{odd}\ x) \ \to \mathbf{eq}\ (\mathrm{even}\ x) \end{aligned}$$

To prove **eq** (odd i) for this set of Horn clauses, it is sufficient to use this formula directly as coinduction hypothesis, as shown in Fig. 6. Note that this formula is indeed a coinduction goal of *co-fohc*, hence we find ourselves in the simplest scenario of coinductive proof search. In Table 1, γ_1 is a representative for this kind of coinductive proofs with simplest atomic goals.

It was pointed out in [37] that Haskell's type class inference can also give rise to irregular corecursion. Such cases may require the more general coinduction

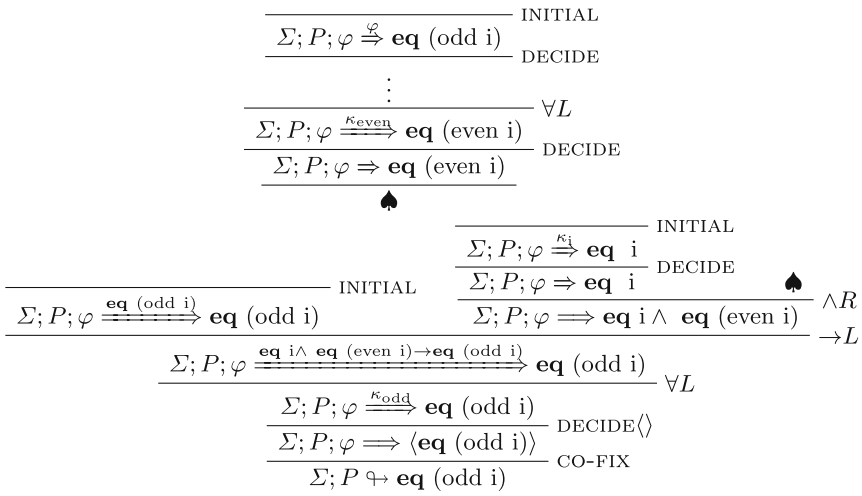


Fig. 6. The *co-fohc* proof for Horn clauses arising from Haskell Type class examples. φ abbreviates the coinduction hypothesis **eq** (odd i). Note its use in the branch ♠.

hypothesis (e.g. universal and/or implicative) of *co-fohh* or *co-hohh*. The below set of Horn clauses is a simplified representation of a problem given in [37]:

$$\begin{aligned} \kappa_{\text{i}} &: \mathbf{eq}\ \text{i} \\ \kappa_s &: \forall x.\, (\mathbf{eq}\ x) \wedge \mathbf{eq}\ (s\ (g\ x)) \rightarrow \mathbf{eq}\ (s\ x) \\ \kappa_g &: \forall x.\, \mathbf{eq}\ x \rightarrow \mathbf{eq}\ (g\ x) \end{aligned}$$

Trying to prove **eq** (s i) by using **eq** (s i) directly as a coinduction hypothesis is deemed to fail, as the coinductive proof search is irregular and this coinduction hypothesis would not be applicable in any guarded context. But it is possible to prove **eq** (s i) as a corollary of another theorem: $\forall x.\, (\mathbf{eq}\ x) \rightarrow \mathbf{eq}\ (s\ x)$. Using this formula as coinduction hypothesis leads to a successful proof, which we omit here. From this more general goal, we can derive the original goal by instantiating the quantifier with i and eliminating the implication with κ_{i}. This second derivation is sound with respect to the models, as we show in Theorem 34.

We encounter γ_2 from Table 1 in a similar situation: To prove $p\,a$, we first have to prove $\forall x.\, p\ x$ in *co-fohh*, and then obtain $p\ a$ as a corollary by appealing to Theorem 34. The next example shows that we can cover all cases in Table 1 by providing a proof in *co-hohh*$_{\text{fix}}$ that involves irregular recursive terms.

Example 15. Recall the clause $\forall x\ y.\, \mathbf{from}\ (s\ x)\ y \rightarrow \mathbf{from}\ x\ (\text{scons}\ x\ y)$ that we named $\kappa_{\mathbf{from}}$ in the introduction. Proving $\exists y.\, \mathbf{from}\ 0\ y$ is again not possible directly. Instead, we can use the term $s_{\text{fr}} = \text{fix}\ f.\, \lambda x.\, \text{scons}\ x\ (f\ (s\ x))$ from Example 6 and prove $\forall x.\, \mathbf{from}\ x\ (s_{\text{fr}}\ x)$ coinductively, as shown in Fig. 7. This formula gives a coinduction hypothesis of sufficient generality. Note that the correct coinduction hypothesis now requires the fixed point definition of an

infinite stream of successive numbers and universal quantification in the goal. Hence the need for the richer language of *co-hohh*$_{\text{fix}}$. From this more general goal we can derive our initial goal $\exists\, y.\mathbf{from}\ 0\ y$ by instantiating y with $s_{\text{fr}}\ 0$.

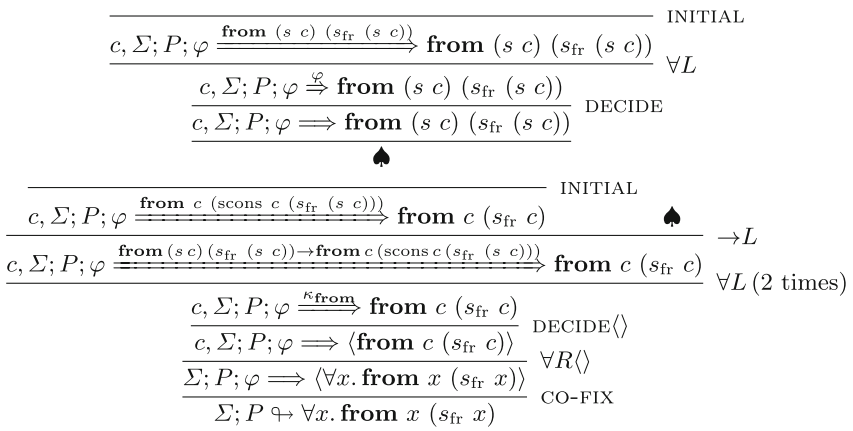


Fig. 7. The *co-hohh*$_{\text{fix}}$ proof for $\varphi = \forall x.\,\mathbf{from}\ x\ (s_{\text{fr}}\ x)$. Note that the last step of the leftmost branch involves $\mathbf{from}\ c\ (\text{scons}\ c\ (s_{\text{fr}}\ (s\ c))) \equiv \mathbf{from}\ c\ (s_{\text{fr}}\ c)$.

There are examples of coinductive proofs that require a fixed point definition of an infinite stream, but do not require the syntax of higher-order terms or hereditary Harrop formulae. Such proofs can be performed in the *co-fohc*$_{\text{fix}}$ logic. A good example is a proof that the stream of zeros satisfies the Horn clause theory defining the predicate **stream** in the introduction. The goal $(\mathbf{stream}\ s_0)$, with $s_0 = \text{fix}\, x.\,\text{scons}\ 0\ x$ can be proven directly by coinduction. Similarly, one can type self-application with the infinite type $a = \text{fix}\, t.\, t \to b$ for some given type b. The proof for $\mathbf{typed}\ [x : a]\ (\text{app}\ x\ x)\ b$ is then in *co-fohc*$_{\text{fix}}$. Finally, the clause γ_3 is also in this group. More generally, circular unifiers obtained from CoLP's [41] loop detection yield immediately guarded fixed point terms, and thus CoLP corresponds to coinductive proofs in the logic *co-fohc*$_{\text{fix}}$. A general discussion of Horn clause theories that describe infinite objects was given in [48], where the above logic programs were identified as being productive.

4 Coinductive Uniform Proofs and Intuitionistic Logic

In the last section, we introduced the framework of coinductive uniform proofs, which gives an operational account to proofs for coinductively interpreted logic programs. Having this framework at hand, we need to position it in the existing ecosystem of logical systems. The goal of this section is to prove that coinductive uniform proofs are in fact constructive. We show this by first introducing an extension of intuitionistic first-order logic that allows us to deal with recursive

$$\frac{\Gamma \Vdash \Delta \qquad \varphi \in \Delta}{\Gamma \mid \Delta \vdash \varphi}\ \textbf{(Proj)} \qquad \frac{\Gamma \mid \Delta \vdash \varphi' \qquad \varphi \equiv \varphi'}{\Gamma \mid \Delta \vdash \varphi}\ \textbf{(Conv)} \qquad \frac{\Gamma \Vdash \Delta}{\Gamma \mid \Delta \vdash \top}\ (\top\textbf{-I})$$

$$\frac{\Gamma \mid \Delta \vdash \varphi \qquad \Gamma \mid \Delta \vdash \psi}{\Gamma \mid \Delta \vdash \varphi \wedge \psi}\ (\wedge\textbf{-I}) \qquad \frac{\Gamma \mid \Delta \vdash \varphi_1 \wedge \varphi_2 \qquad i \in \{1,2\}}{\Gamma \mid \Delta \vdash \varphi_i}\ (\wedge_i\textbf{-E})$$

$$\frac{\Gamma \mid \Delta \vdash \varphi_i \qquad \Gamma \Vdash \varphi_j \qquad j \neq i}{\Gamma \mid \Delta \vdash \varphi_1 \vee \varphi_2}\ (\vee_i\textbf{-I}) \qquad \frac{\Gamma \mid \Delta, \varphi_1 \vdash \psi \qquad \Gamma \mid \Delta, \varphi_2 \vdash \psi}{\Gamma \mid \Delta, \varphi_1 \vee \varphi_2 \vdash \psi}\ (\vee\textbf{-E})$$

$$\frac{\Gamma \mid \Delta, \varphi \vdash \psi}{\Gamma \mid \Delta \vdash \varphi \to \psi}\ (\to\textbf{-I}) \qquad \frac{\Gamma \mid \Delta \vdash \varphi \to \psi \qquad \Gamma \mid \Delta \vdash \varphi}{\Gamma \mid \Delta \vdash \psi}\ (\to\textbf{-E})$$

$$\frac{\Gamma, x : \tau \mid \Delta \vdash \varphi \qquad x \notin \Gamma}{\Gamma \mid \Delta \vdash \forall x : \tau.\, \varphi}\ (\forall\textbf{-I}) \qquad \frac{\Gamma \mid \Delta \vdash \forall x : \tau.\, \varphi \qquad M : \tau \in \Lambda^G_\Sigma(\Gamma)}{\Gamma \mid \Delta \vdash \varphi\,[M/x]}\ (\forall\textbf{-E})$$

$$\frac{M : \tau \in \Lambda^G_\Sigma(\Gamma) \qquad \Gamma \mid \Delta \vdash \varphi\,[M/x]}{\Gamma \mid \Delta \vdash \exists x : \tau.\, \varphi}\ (\exists\textbf{-I}) \qquad \frac{\Gamma \Vdash \psi \qquad \Gamma, x : \tau \mid \Delta, \varphi \vdash \psi \qquad x \notin \Gamma}{\Gamma \mid \Delta, \exists x : \tau.\, \varphi \vdash \psi}\ (\exists\textbf{-E})$$

Fig. 8. Intuitionistic rules for standard connectives

proofs for coinductive predicates. Afterwards, we show that coinductive uniform proofs are sound relative to this logic by means of a proof tree translation. The model-theoretic soundness proofs for both logics will be provided in Sect. 5.

We begin by introducing an extension of intuitionistic first-order logic with the so-called *later modality*, written $\blacktriangleright$. This modality is the essential ingredient that allows us to equip proofs with a controlled form of recursion. The later modality stems originally from provability logic, which characterises transitive, well-founded Kripke frames [30,72], and thus allows one to carry out induction without an explicit induction scheme [16]. Later, the later modality was picked up by the type-theoretic community to control recursion in coinductive programming [8,9,21,56,58], mostly with the intent to replace syntactic guardedness checks for coinductive definitions by type-based checks of well-definedness.

Formally, the logic $\mathbf{iFOL}_{\blacktriangleright}$ is given by the following definition.

Definition 16. The formulae of $\mathbf{iFOL}_{\blacktriangleright}$ are given by Definition 3 and the rule:

$$\frac{\Gamma \Vdash \varphi}{\Gamma \Vdash \blacktriangleright \varphi}$$

Conversion extends to these formulae in the obvious way. Let φ be a formula and Δ a sequence of formulae in $\mathbf{iFOL}_{\blacktriangleright}$. We say φ is *provable in context* Γ *under the assumptions* Δ in $\mathbf{iFOL}_{\blacktriangleright}$, if $\Gamma \mid \Delta \vdash \varphi$ holds. The *provability relation* $\vdash$ is thereby given inductively by the rules in Figs. 8 and 9.

$$\frac{\Gamma \mid \Delta \vdash \varphi}{\Gamma \mid \Delta \vdash \blacktriangleright \varphi}\ \textbf{(Next)} \qquad \frac{\Gamma \mid \Delta \vdash \blacktriangleright(\varphi \to \psi)}{\Gamma \mid \Delta \vdash \blacktriangleright \varphi \to \blacktriangleright \psi}\ \textbf{(Mon)} \qquad \frac{\Gamma \mid \Delta, \blacktriangleright \varphi \vdash \varphi}{\Gamma \mid \Delta \vdash \varphi}\ \textbf{(Löb)}$$

Fig. 9. Rules for the later modality

The rules in Fig. 8 are the usual rules for intuitionistic first-order logic and should come at no surprise. More interesting are the rules in Fig. 9, where the rule **(Löb)** introduces recursion into the proof system. Furthermore, the rule **(Mon)** allows us to to distribute the later modality over implication, and consequently over conjunction and universal quantification. This is essential in the translation in Theorem 18 below. Finally, the rule **(Next)** gives us the possibility to proceed without any recursion, if necessary.

Note that so far it is not possible to use the assumption $\blacktriangleright\varphi$ introduced in the **(Löb)**-rule. The idea is that the formulae of a logic program provide us the obligations that we have to prove, possibly by recursion, in order to prove a coinductive predicate. This is cast in the following definition.

Definition 17. Given an H^g-formula φ of the shape $\forall\vec{x}.\,(A_1 \wedge \cdots \wedge A_n) \to \psi$, we define its *guarding* $\overline{\varphi}$ to be $\forall\vec{x}.\,(\blacktriangleright A_1 \wedge \cdots \wedge \blacktriangleright A_n) \to \psi$. For a logic program P, we define its guarding $\overline{P}$ by guarding each formula in P.

The translation given in Definition 17 of a logic program into formulae that admit recursion corresponds unfolding a coinductive predicate, cf. [14]. We show now how to transform a coinductive uniform proof tree into a proof tree in $\mathbf{iFOL}_{\blacktriangleright}$, such that the recursion and guarding mechanisms in both logics match up.

Theorem 18. *If P is a logic program over a first-order signature Σ and the sequent $\Sigma; P \rightarrowtail \varphi$ is provable in co-hohh$_{\mathrm{fix}}$, then $\overline{P} \vdash \varphi$ is provable in* $\mathbf{iFOL}_{\blacktriangleright}$.

To prove this theorem, one uses that each coinductive uniform proof tree starts with an initial tree that has an application of the CO-FIX-rule at the root and that eliminates the guard by using the rules in Fig. 5. At the leaves of this tree, one finds proof trees that proceed only by means of the rules in Fig. 4. The initial tree is then translated into a proof tree in $\mathbf{iFOL}_{\blacktriangleright}$ that starts with an application of the **(Löb)**-rule, which corresponds to the CO-FIX-rule, and that simultaneously transforms the coinduction hypothesis and applies introduction rules for conjunctions etc. This ensures that we can match the coinduction hypothesis with the guarded formulae of the program P.

The results of this section show that it is irrelevant whether the guarding modality is used on the right (CUP-style) or on the left ($\mathbf{iFOL}_{\blacktriangleright}$-style), as the former can be translated into the latter. However, CUP uses the guarding on the right to preserve proof uniformity, whereas $\mathbf{iFOL}_{\blacktriangleright}$ extends a general sequent calculus. Thus, to obtain the reverse translation, we would have to have an admissible cut rule in CUP. The main ingredient to such a cut rule is the ability to prove several coinductive statements simultaneously. This is possible in CUP by proving the conjunction of these statements. Unfortunately, we cannot eliminate such a conjunction into one of its components, since this would require non-deterministic guessing in the proof construction, which in turn breaks uniformity. Thus, we leave a solution of this problem for future work.

5 Herbrand Models and Soundness

In Sect. 4 we showed that coinductive uniform proofs are sound relative to the intuitionistic logic $\mathbf{iFOL}_{\blacktriangleright}$. This gives us a handle on the constructive nature of coinductive uniform proofs. Since $\mathbf{iFOL}_{\blacktriangleright}$ is a non-standard logic, we still need to provide semantics for that logic. We do this by interpreting in Sect. 5.4 the formulae of $\mathbf{iFOL}_{\blacktriangleright}$ over the well-known (complete) Herbrand models and prove the soundness of the accompanying proof system with respect to these models. Although we obtain soundness of coinductive uniform proofs over Herbrand models from this, this proof is indirect and does not give a lot of information about the models captured by the different calculi *co-fohc* etc. For this reason, we will give in Sect. 5.3 a direct soundness proof for coinductive uniform proofs. We also obtain coinduction invariants from this proof for each of the calculi, which allows us to describe their proof strength.

5.1 Coinductive Herbrand Models and Semantics of Terms

Before we come to the soundness proofs, we introduce in this section (complete) Herbrand models by using the terminology of final coalgebras. We then utilise this description to give operational and denotational semantics to guarded terms. These semantics show that guarded terms allow the description and computation of potentially infinite trees.

The coalgebraic approach has been proven very successful both in logic and programming [1,75,76]. We will only require very little category theoretical vocabulary and assume that the reader is familiar with the category **Set** of sets and functions, and functors, see for example [12,25,50]. The terminology of algebras and coalgebras [4,47,64,65] is given by the following definition.

Definition 19. A *coalgebra* for a functor $F\colon \mathbf{Set} \to \mathbf{Set}$ is a map $c\colon X \to FX$. Given coalgebras $d : Y \to FY$ and $c\colon X \to FX$, we say that a map $h : Y \to X$ is a *homomorphism* $d \to c$ if $Fh \circ d = c \circ h$. We call a coalgebra $c\colon X \to FX$ *final*, if for every coalgebra d there is a unique homomorphism $h\colon d \to c$. We will refer to h as the *coinductive extension* of d.

The idea of (complete) Herbrand models is that a set of Horn clauses determines for each predicate symbol a set of potentially infinite terms. Such terms are (potentially infinite) trees, whose nodes are labelled by function symbols and whose branching is given by the arity of these function symbols. To be able to deal with open terms, we will allow such trees to have leaves labelled by variables. Such trees are a final coalgebra for a functor determined by the signature.

Definition 20. Let Σ be first-order signature. The *extension* of a first-order signature Σ is a (polynomial) functor [38] $[\![\Sigma]\!] : \mathbf{Set} \to \mathbf{Set}$ given by

$$[\![\Sigma]\!](X) = \coprod_{f \in \Sigma} X^{\mathrm{ar}(f)},$$

where $\mathrm{ar}\colon \Sigma \to \mathbb{N}$ is defined in Sect. 2 and X^n is the n-fold product of X. We define for a set V a functor $[\![\Sigma]\!]+V\colon \mathbf{Set} \to \mathbf{Set}$ by $([\![\Sigma]\!]+V)(X) = [\![\Sigma]\!](X)+V$, where $+$ is the coproduct (disjoint union) in **Set**.

To make sense of the following definition, we note that we can view Π as a signature and we thus obtain its extension $[\![\Pi]\!]$. Moreover, we note that the final coalgebra of $[\![\Sigma]\!] + V$ exists because $[\![\Sigma]\!]$ is a polynomial functor.

Definition 21. Let Σ be a first-order signature. The *coterms* over Σ are the final coalgebra $\mathrm{root}_V \colon \Sigma^\infty(V) \to [\![\Sigma]\!](\Sigma^\infty(V)) + V$. For brevity, we denote the coterms with no variables, i.e. $\Sigma^\infty(\emptyset)$, by $\mathrm{root} \colon \Sigma^\infty \to [\![\Sigma]\!](\Sigma^\infty)$, and call it the *(complete) Herbrand universe* and its elements *ground* coterms. Finally, we let the *(complete) Herbrand base* $\mathcal{B}^\infty$ be the set $[\![\Pi]\!](\Sigma^\infty)$.

The construction $\Sigma^\infty(V)$ gives rise to a functor $\Sigma^\infty \colon \mathbf{Set} \to \mathbf{Set}$, called the *free completely iterative monad* [5]. If there is no ambiguity, we will drop the injections κ_i when describing elements of $\Sigma^\infty(V)$. Note that $\Sigma^\infty(V)$ is final with property that for every $s \in \Sigma^\infty(V)$ either there are $f \in \Sigma$ and $\vec{t} \in (\Sigma^\infty(V))^{\mathrm{ar}(f)}$ with $\mathrm{root}_V(s) = f(\vec{t})$, or there is $x \in V$ with $\mathrm{root}_V(s) = x$. Finality allows us to specify unique maps into $\Sigma^\infty(V)$ by giving a coalgebra $X \to [\![\Sigma]\!](X) + V$. In particular, one can define for each $\theta \colon V \to \Sigma^\infty$ the substitution $t[\theta]$ of variables in the coterm t by θ as the coinductive extension of the following coalgebra.

$$\Sigma^\infty(V) \xrightarrow{\mathrm{root}_V} [\![\Sigma]\!](\Sigma^\infty(V)) + V \xrightarrow{[\mathrm{id},\mathrm{root}\circ\theta]} [\![\Sigma]\!](\Sigma^\infty(V))$$

Now that we have set up the basic terminology of coalgebras, we can give semantics to guarded terms from Definition 5. The idea is that guarded terms guarantee that we can always compute with them so far that we find a function symbol in head position, see Lemma 8. This function symbol determines then the label and branching of a node in the tree generated by a guarded term. If the computation reaches a constant or a variable, then we stop creating the tree at the present branch. This idea is captured by the following lemma.

Lemma 22. *There is a map* $[\![-]\!]_1 \colon \Lambda_\Sigma^{G,1}(\Gamma) \to \Sigma^\infty(\Gamma)$ *that is unique with*

1. *if* $M \equiv N$, *then* $[\![M]\!]_1 = [\![N]\!]_1$, *and*
2. *for all* M, *if* $M \longrightarrow\!\!\!\!\!\rightarrow f\,\vec{N}$ *then* $\mathrm{root}_\Gamma([\![M]\!]_1) = f\big(\overrightarrow{[\![N]\!]_1}\big)$, *and if* $M \longrightarrow\!\!\!\!\!\rightarrow x$ *then* $\mathrm{root}_\Gamma([\![M]\!]_1) = x$.

Proof (sketch). By Lemma 8, we can define a coalgebra on the quotient of guarded terms by convertibility $c \colon \Lambda_\Sigma^{G,1}(\Gamma)/\!\equiv \;\to [\![\Sigma]\!]\Big(\Lambda_\Sigma^{G,1}(\Gamma)/\!\equiv\Big) + \Gamma$ with $c[M] = f[\vec{N}]$ if $M \longrightarrow\!\!\!\!\!\rightarrow f\,\vec{N}$ and $c[M] = x$ if $M \longrightarrow\!\!\!\!\!\rightarrow x$. This yields a homomorphism $h \colon \Lambda_\Sigma^{G,1}(\Gamma)/\!\equiv \;\to \Sigma^\infty(\Gamma)$ and we can define $[\![-]\!]_1 = h \circ [-]$. The rest follows from uniqueness of h.

5.2 Interpretation of Basic Intuitionistic First-Order Formulae

In this section, we give an interpretation of the formulae in Definition 3, in which we restrict ourselves to guarded terms. This interpretation will be relative to models in the complete Herbrand universe. Since we later extend these models to Kripke models to be able to handle the later modality, we formulate these models already now in the language of fibrations [17,46].

Definition 23. Let $p\colon \mathbf{E} \to \mathbf{B}$ be a functor. Given an object $I \in \mathbf{B}$, the *fibre* $\mathbf{E}_I$ above I is the category of objects $A \in \mathbf{E}$ with $p(A) = I$ and morphisms $f\colon A \to B$ with $p(f) = \mathrm{id}_I$. The functor p is a *(split) fibration* if for every morphism $u\colon I \to J$ in $\mathbf{B}$ there is functor $u^*\colon \mathbf{E}_J \to \mathbf{E}_I$, such that $\mathrm{id}_I^* = \mathrm{Id}_{\mathbf{E}_I}$ and $(v \circ u)^* = u^* \circ v^*$. We call u^* the *reindexing along* u.

To give an interpretation of formulae, consider the following category **Pred**.

$$\mathbf{Pred} = \begin{cases} \text{objects}: & (X, P) \text{ with } X \in \mathbf{Set} \text{ and } P \subseteq X \\ \text{morphisms}: & f : (X, P) \to (Y, Q) \text{ is a map } f : X \to Y \text{ with } f(P) \subseteq Q \end{cases}$$

The functor $\mathbb{P}\colon \mathbf{Pred} \to \mathbf{Set}$ with $\mathbb{P}(X, P) = X$ and $\mathbb{P}(f) = f$ is a split fibration, see [46], where the reindexing functor for $f\colon X \to Y$ is given by taking preimages: $f^*(Q) = f^{-1}(Q)$. Note that each fibre $\mathbf{Pred}_X$ is isomorphic to the complete lattice of predicates over X ordered by set inclusion. Thus, we refer to this fibration as the *predicate fibration*.

Let us now expose the logical structure of the predicate fibration. This will allow us to conveniently interpret first-order formulae over this fibration, but it comes at the cost of having to introduce a good amount of category theoretical language. However, doing so will pay off in Sect. 5.4, where we will construct another fibration out of the predicate fibration. We can then use category theoretical results to show that this new fibration admits the same logical structure and allows the interpretation of the later modality.

The first notion we need is that of fibred products, coproducts and exponents, which will allow us to interpret conjunction, disjunction and implication.

Definition 24. A fibration $p\colon \mathbf{E} \to \mathbf{B}$ has *fibred finite products* $(\mathbf{1}, \times)$, if each fibre $\mathbf{E}_I$ has finite products $(\mathbf{1}_I, \times_I)$ and these are preserved by reindexing: for all $f\colon I \to J$, we have $f^*(\mathbf{1}_J) = \mathbf{1}_I$ and $f^*(A \times_J B) = f^*(A) \times_I f^*(B)$. Fibred finite coproducts and exponents are defined analogously.

The fibration $\mathbb{P}$ is a so-called first-order fibration, which allows us to interpret first-order logic, see [46, Def. 4.2.1].

Definition 25. A fibration $p\colon \mathbf{E} \to \mathbf{B}$ is a *first-order fibration* if[2]

- $\mathbf{B}$ has finite products and the fibres of p are preorders;
- p has fibred finite products $(\top, \wedge)$ and coproducts $(\bot, \vee)$ that distribute;
- p has fibred exponents $\to$; and
- p has existential and universal quantifiers $\exists_{I,J} \dashv \pi_{I,J}^* \dashv \forall_{I,J}$ for all projections $\pi_{I,J}\colon I \times J \to I$.

A *first-order* λ*-fibration* is a first-order fibration with Cartesian closed base $\mathbf{B}$.

[2] Technically, the quantifiers should also fulfil the Beck-Chevalley and Frobenius conditions, and the fibration should admit equality. Since these are fulfilled in all our models and we do not need equality, we will not discuss them here.

The fibration $\mathbb{P}\colon \mathbf{Pred} \to \mathbf{Set}$ is a first-order λ-fibration, as all its fibres are posets and **Set** is Cartesian closed; $\mathbb{P}$ has fibred finite products $(\top, \cap)$, given by $\top_X = X$ and intersection; fibred distributive coproducts $(\emptyset, \cup)$; fibred exponents $\Rightarrow$, given by $(P \Rightarrow Q) = \{\vec{t} \mid \text{if } \vec{t} \in P, \text{ then } \vec{t} \in Q\}$; and universal and existential quantifiers given for $P \in \mathbf{Pred}_{X \times Y}$ by

$$\forall_{X,Y} P = \{x \in X \mid \forall y \in Y.\, (x, y) \in P\} \quad \exists_{X,Y} P = \{x \in X \mid \exists y \in Y.\, (x, y) \in P\}.$$

The purpose of first-order fibrations is to capture the essentials of first-order logic, while the λ-part takes care of higher-order features of the term language. In the following, we interpret types, contexts, guarded terms and formulae in the fibration $\mathbb{P}\colon \mathbf{Pred} \to \mathbf{Set}$: We define for types τ and context Γ sets $[\![\tau]\!]$ and $[\![\Gamma]\!]$; for guarded terms M with $\Gamma \vdash M : \tau$ we define a map $[\![M]\!]\colon [\![\Gamma]\!] \to [\![\tau]\!]$ in **Set**; and for a formula $\Gamma \Vdash \varphi$ we give a predicate $[\![\varphi]\!] \in \mathbf{Pred}_{[\![\Gamma]\!]}$.

The semantics of types and contexts are given inductively in the Cartesian closed category **Set**, where the base type ι is interpreted as coterms, as follows.

$$[\![\iota]\!] = \Sigma^\infty \qquad [\![\emptyset]\!] = \mathbf{1}$$
$$[\![\tau \to \sigma]\!] = [\![\sigma]\!]^{[\![\tau]\!]} \qquad [\![\Gamma, x : \tau]\!] = [\![\Gamma]\!] \times [\![\tau]\!]$$

We note that a coterm $t \in \Sigma^\infty(V)$ can be seen as a map $(\Sigma^\infty)^V \to \Sigma^\infty$ by applying a substitution in $(\Sigma^\infty)^V$ to $t\colon \sigma \mapsto t[\sigma]$. In particular, the semantics of a guarded first-order term $M \in \Lambda^{G,1}_\Sigma(\Gamma)$ is equivalently a map $[\![M]\!]_1 : [\![\Gamma]\!] \to \Sigma^\infty$. We can now extend this map inductively to $[\![M]\!]\colon [\![\Gamma]\!] \to [\![\tau]\!]$ for all guarded terms $M \in \Lambda^G_\Sigma(\Gamma)$ with $\Gamma \vdash M : \tau$ by

$$\begin{aligned}
[\![M]\!](\gamma)(\vec{t}) &= [\![M\ \vec{x}]\!]_1([\vec{x} \mapsto \vec{t}]) \qquad \vdash_g M : \tau \text{ with } \mathrm{ar}(\tau) = |\vec{t}| = |\vec{x}| \\
[\![c]\!](\gamma)(\vec{t}) &= c\ \vec{t} \\
[\![x]\!](\gamma) &= \gamma(x) \\
[\![M\ N]\!](\gamma) &= [\![M]\!](\gamma)([\![N]\!](\gamma)) \\
[\![\lambda x.\, M]\!](\gamma)(t) &= [\![M]\!](\gamma[x \mapsto t])
\end{aligned}$$

Lemma 26. *The mapping $[\![-]\!]$ is a well-defined function from guarded terms to functions, such that $\Gamma \vdash M : \tau$ implies $[\![M]\!]\colon [\![\Gamma]\!] \to [\![\tau]\!]$.*

Since $\mathbb{P}\colon \mathbf{Pred} \to \mathbf{Set}$ is a first-order fibration, we can interpret inductively all logical connectives of the formulae from Definition 3 in this fibration. The only case that is missing is the base case of predicate symbols. Their interpretation will be given over a Herbrand model that is constructed as the largest fixed point of an operator over all predicate interpretations in the Herbrand base. Both the operator and the fixed point are the subjects of the following definition.

Definition 27. We let the set of *interpretations* $\mathcal{I}$ be the powerset $\mathcal{P}(\mathcal{B}^\infty)$ of the complete Herbrand base. For $I \in \mathcal{I}$ and $p \in \Pi$, we denote by $I|_p$ the interpretation of p in I (the fibre of I above p)

$$I|_p = \left\{ \vec{t} \in (\Sigma^\infty)^{\mathrm{ar}(p)} \;\middle|\; p(\vec{t}) \in I \right\}.$$

Given a set P of H^g-formulae, we define a monotone map $\Phi_P \colon \mathcal{I} \to \mathcal{I}$ by

$$\Phi_P(I) = \{[\![\psi]\!]_1[\theta] \mid (\forall \vec{x}.\, \textstyle\bigwedge_{k=1}^{n} \varphi_k \to \psi) \in P, \theta \colon |\vec{x}| \to \Sigma^\infty, \forall k.\, [\![\varphi_k]\!]_1[\theta] \in I\},$$

where $[\![-]\!]_1[\theta]$ is the extension of semantics and substitution from coterms to the Herbrand base by functoriality of $[\![\Pi]\!]$. The *(complete) Herbrand model* $\mathcal{M}_P$ of P is the largest fixed point of Φ_P, which exists because $\mathcal{I}$ is a complete lattice.

Given a formula φ with $\Gamma \Vdash \varphi$ that contains only guarded terms, we define the semantics of φ in **Pred** from an interpretation $I \in \mathcal{I}$ inductively as follows.

$$\begin{aligned}
[\![\Gamma \Vdash p\,\vec{M}]\!]_I &= \left([\![\vec{M}]\!]\right)^*(I|_p) \\
[\![\Gamma \Vdash \top]\!]_I &= \top_{[\![\Gamma]\!]} \\
[\![\Gamma \Vdash \varphi \,\square\, \psi]\!]_I &= [\![\Gamma \Vdash \varphi]\!]_I \,\square\, [\![\Gamma \Vdash \psi]\!]_I && \square \in \{\wedge, \vee, \to\} \\
[\![\Gamma \Vdash Qx : \tau.\, \varphi]\!]_I &= Q_{[\![\Gamma]\!],[\![\tau]\!]}\, [\![\Gamma, x : \tau \Vdash \varphi]\!]_I && Q \in \{\forall, \exists\}
\end{aligned}$$

Lemma 28. *The mapping $[\![-]\!]_I$ is a well-defined function from formulae to predicates, such that $\Gamma \Vdash \varphi$ implies $[\![\varphi]\!]_I \subseteq [\![\Gamma]\!]$ or, equivalently, $[\![\varphi]\!]_I \in \mathbf{Pred}_{[\![\Gamma]\!]}$.*

This concludes the semantics of types, terms and formulae. We now turn to show that coinductive uniform proofs are sound for this interpretation.

5.3 Soundness of Coinductive Uniform Proofs for Herbrand Models

In this section, we give a direct proof of soundness for the coinductive uniform proof system from Sect. 3. Later, we will obtain another soundness result by combining the proof translation from Theorem 18 with the soundness of **iFOL**$_\blacktriangleright$ (Theorems 39 and 42). The purpose of giving a direct soundness proof for uniform proofs is that it allows the extraction of a coinduction invariant, see Lemma 32.

The main idea is as follows. Given a formula φ and a uniform proof π for $\Sigma; P \Rightarrow \varphi$, we construct an interpretation $I \in \mathcal{I}$ that validates φ, i.e. $[\![\varphi]\!]_I = \top$, and that is contained in the complete Herbrand model $\mathcal{M}_P$. Combining these two facts, we obtain that $[\![\varphi]\!]_{\mathcal{M}_P} = \top$, and thus the soundness of uniform proofs.

To show that the constructed interpretation I is contained in $\mathcal{M}_P$, we use the usual coinduction proof principle, as it is given in the following definition.

Definition 29. An *invariant for* $K \in \mathcal{I}$ is a set $I \in \mathcal{I}$, such that $K \subseteq I$ and I is a Φ_P-invariant, that is, $I \subseteq \Phi_P(I)$. If K has an invariant, then $K \subseteq \mathcal{M}_P$.

Thus, our goal is now to construct an interpretation together with an invariant. This invariant will essentially collect and iterate all the substitutions that appear in a proof. For this we need the ability to compose substitutions of coterms, which we derive from the monad [5] $(\Sigma^\infty, \eta, \mu)$ with $\mu \colon \Sigma^\infty \Sigma^\infty \Rightarrow \Sigma^\infty$.

Definition 30. A *(Kleisli-)substitution* θ from V to W, written $\theta \colon V \multimap W$, is map $V \to \Sigma^\infty(W)$. Composition of $\theta \colon V \multimap W$ and $\delta \colon U \multimap V$ is given by

$$\theta \circledcirc \delta = U \xrightarrow{\delta} \Sigma^\infty(V) \xrightarrow{\Sigma^\infty(\theta)} \Sigma^\infty(\Sigma^\infty(W)) \xrightarrow{\mu_W} \Sigma^\infty(W).$$

The notions in the following definition will allow us to easily organise and iterate the substitutions that occur in a uniform proof.

Definition 31. Let S be a set with $S = \{1, \dots, n\}$ for some $n \in \mathbb{N}$. We call the set S^* of lists over S the set of *substitution identifiers*. Suppose that we have substitutions $\theta_0 \colon V \nrightarrow \emptyset$ and $\theta_k \colon V \nrightarrow V$ for each $k \in S$. Then we can define a map $\Theta \colon S^* \to (\Sigma^\infty)^V$, which turns each substitution identifier into a substitution, by iteration from the right:

$$\Theta(\varepsilon) = \theta_0 \quad \text{and} \quad \Theta(w : k) = \Theta(w) \circledcirc \theta_k$$

After introducing these notations, we can give the outline of the soundness proof for uniform proofs relative to the complete Herbrand model. Given an H^g-formula $\forall \vec{x}.\, \varphi$, we note that a uniform proof π for $\Sigma; P \Rightarrow \forall \vec{x}.\, \varphi$ starts with

$$\dfrac{\dfrac{\vec{c} : \iota, \Sigma; P; \Delta \Longrightarrow \langle \varphi[\vec{c}/\vec{x}] \rangle \qquad \vec{c} : \iota \notin \Sigma}{\Sigma; P; \forall \vec{x}.\, \varphi \Longrightarrow \langle \forall \vec{x}.\, \varphi \rangle}\, \forall R\langle\rangle}{\Sigma; P \Rightarrow \forall \vec{x}.\, \varphi}\, \text{CO-FIX}$$

where the eigenvariables in $\vec{c}$ are all distinct. Let Σ^c be the signature $\vec{c} : \iota, \Sigma$ and C the set of variables in $\vec{c}$. Suppose the following is a valid subtree of π.

$$\dfrac{\dfrac{\Sigma^c; P; \Delta \overset{\varphi[\vec{N}/\vec{x}]}{\Longrightarrow} A}{\Sigma^c; P; \Delta \overset{\forall \vec{x}.\, \varphi \in \Delta}{\Longrightarrow} A}\, \forall L}{\Sigma^c; P; \Delta \Longrightarrow A}\, \text{DECIDE}$$

This proof tree gives rise to a substitution $\delta \colon C \nrightarrow C$ by $\delta(c) = [\![N_c]\!]$, which we call an *agent* of π. We let $D \subseteq \mathrm{At}_1^g$ be the set of atoms that are proven in π:

$$D = \{A \mid \Sigma^c; P; \Delta \Longrightarrow \langle A \rangle \text{ or } \Sigma^c; P; \Delta \Longrightarrow A \text{ appears in } \pi\}$$

From the agents and atoms in π we extract an invariant for the goal formula.

Lemma 32. *Suppose that φ is an H^g-formula of the form $\forall \vec{x}.\, A_1 \wedge \dots \wedge A_n \to A_0$ and that there is a proof π for $\Sigma; P \Rightarrow \varphi$. Let D be the proven atoms in π and $\theta_0, \dots, \theta_s$ be the agents of π. Define $A_k^c = A_k[\vec{c}/\vec{x}]$ and suppose further that I_1 is an invariant for $\{A_k^c[\Theta(\varepsilon)] \mid 1 \leq k \leq n\}$. If we put*

$$I_2 = \bigcup_{w \in S^*} D\,[\Theta\,(w)]$$

then $I_1 \cup I_2$ is an invariant for $A_0^c[\Theta(\varepsilon)]$.

Once we have Lemma 32 the following soundness theorem is easily proven.

Theorem 33. *If φ is an H^g-formula and $\Sigma; P \Rightarrow \varphi$, then $[\![\varphi]\!]_{\mathcal{M}_P} = \top$.*

Finally, we show that extending logic programs with coinductively proven lemmas is sound. This follows easily by coinduction.

Theorem 34. *Let φ be an H^g-formula of the shape $\forall\vec{x}.\,\psi_1 \to \psi_2$, such that, for all substitutions θ if $[\![\psi_1]\!]_1[\theta] \in \mathcal{M}_{P,\varphi}$, then $[\![\psi_1]\!]_1[\theta] \in \mathcal{M}_P$. Then $\Sigma; P \upharpoonright\!\!\rightarrow \varphi$ implies $\mathcal{M}_{P\cup\{\varphi\}} = \mathcal{M}_P$, that is, $P \cup \{\varphi\}$ is a conservative extension of P with respect to the Herbrand model.*

As a corollary we obtain that, if there is a proof for $\Sigma; P \upharpoonright\!\!\rightarrow \varphi$, then a proof for $\Sigma; P, \varphi \upharpoonright\!\!\rightarrow \psi$ is sound with respect to $\mathcal{M}_P$. Indeed, by Theorem 34 we have that $\mathcal{M}_P = \mathcal{M}_{P\cup\varphi}$ and by Theorem 33 that $\Sigma; P, \varphi \upharpoonright\!\!\rightarrow \psi$ is sound with respect to $\mathcal{M}_{P\cup\{\varphi\}}$. Thus, the proof of $\Sigma; P, \varphi \upharpoonright\!\!\rightarrow \psi$ is also sound with respect to $\mathcal{M}_P$. We use this property implicitly in our running examples, and refer the reader to [15,49] for proofs, further examples and discussion.

5.4 Soundness of iFOL$_\blacktriangleright$ over Herbrand Models

In this section, we demonstrate how the logic **iFOL$_\blacktriangleright$** can be interpreted over Herbrand models. Recall that we obtained a fixed point model from the monotone map Φ_P on interpretations. In what follows, it is crucial that we construct the greatest fixed point of Φ_P by iteration, c.f. [6,32,77]: Let **Ord** be the class of all ordinals equipped with their (well-founded) order. We denote by $\mathbf{Ord}^{\mathrm{op}}$ the class of ordinals with their reversed order and define a monotone function $\overleftarrow{\Phi_P}\colon \mathbf{Ord}^{\mathrm{op}} \to \mathcal{I}$, where we write the argument ordinal in the subscript, by

$$\big(\overleftarrow{\Phi_P}\big)_\alpha = \bigcap\nolimits_{\beta<\alpha} \Phi_P\big(\overleftarrow{\Phi_P}_\beta\big).$$

Note that this definition is well-defined because $<$ is well-founded and because Φ_P is monotone, see [14]. Since $\mathcal{I}$ is a complete lattice, there is an ordinal α such that $\overleftarrow{\Phi_P}_\alpha = \Phi_P\big(\overleftarrow{\Phi_P}_\alpha\big)$, at which point $\overleftarrow{\Phi_P}_\alpha$ is the largest fixed point $\mathcal{M}_P$ of Φ_P. In what follows, we will utilise this construction to give semantics to **iFOL$_\blacktriangleright$**.

The fibration $\mathbb{P}\colon \mathbf{Pred} \to \mathbf{Set}$ gives rise to another fibration as follows. We let $\overline{\mathbf{Pred}}$ be the category of functors (monotone maps) with fixed predicate domain:

$$\overline{\mathbf{Pred}} = \begin{cases} \text{objects:} & u\colon \mathbf{Ord}^{\mathrm{op}} \to \mathbf{Pred}, \text{such that } \mathbb{P}\circ u \text{ is constant} \\ \text{morphisms:} & u \to v \text{ are natural transformations } f\colon u \Rightarrow v, \\ & \text{such that } \mathbb{P}f\colon \mathbb{P}\circ u \Rightarrow \mathbb{P}\circ v \text{ is the identity} \end{cases}$$

The fibration $\overline{\mathbb{P}}\colon \mathbf{Pred} \to \mathbf{Set}$ is defined by evaluation at any ordinal (here 0), i.e. by $\overline{\mathbb{P}}(u) = \mathbb{P}(u(0))$ and $\overline{\mathbb{P}}(f) = (\mathbb{P}f)_0$, and reindexing along $f\colon X \to Y$ by applying the reindexing of $\mathbb{P}$ point-wise, i.e. by $f^{\#}(u)_\alpha = f^*(u_\alpha)$.

Note that there is a (full) embedding $K\colon \mathbf{Pred} \to \overline{\mathbf{Pred}}$ that is given by $K(X,P) = (X,\overline{P})$ with $\overline{P}_\alpha = P$. One can show [14] that $\overline{\mathbb{P}}$ is again a first-order fibration and that it models the later modality, as in the following theorem.

Theorem 35. *The fibration $\overline{\mathbb{P}}$ is a first-order fibration. If necessary, we denote the first-order connectives by $\dot\top$, $\dot\wedge$ etc. to distinguish them from those in* **Pred**. *Otherwise, we drop the dots. Finite (co)products and quantifiers are given point-wise, while for $X \in \mathbf{Set}$ and $u, v \in \overline{\mathbf{Pred}}_X$ exponents are given by*

$$(v \mathrel{\dot\Rightarrow} u)_\alpha = \bigcap\nolimits_{\beta\le\alpha} (v_\beta \Rightarrow u_\beta).$$

There is a fibred functor $\blacktriangleright\colon \overline{\mathbf{Pred}} \to \overline{\mathbf{Pred}}$ *with* $\overline{\pi} \circ \blacktriangleright = \overline{\pi}$ *given on objects by*

$$(\blacktriangleright u)_\alpha = \bigcap\nolimits_{\beta<\alpha} u_\beta$$

and a natural transformation $\mathrm{next}\colon \mathrm{Id} \Rightarrow \blacktriangleright$ *from the identity functor to* $\blacktriangleright$. *The functor* $\blacktriangleright$ *preserves reindexing, products, exponents and universal quantification:* $\blacktriangleright(f^\# u) = f^\#(\blacktriangleright u)$, $\blacktriangleright(u \wedge v) = \blacktriangleright u \wedge \blacktriangleright v$, $\blacktriangleright(u^v) \to (\blacktriangleright u)^{\blacktriangleright v}$, $\blacktriangleright\big(\overline{\forall}_n u\big) = \overline{\forall}_n(\blacktriangleright u)$. *Finally, for all* $X \in \mathbf{Set}$ *and* $u \in \overline{\mathbf{Pred}}_X$, *there is* $\mathrm{löb}\colon (\blacktriangleright u \dot{\Rightarrow} u) \to u$ *in* $\overline{\mathbf{Pred}}_X$.

Using the above theorem, we can extend the interpretation of formulae to $\mathbf{iFOL}_\blacktriangleright$ as follows. Let $u\colon \mathbf{Ord}^{\mathrm{op}} \to \mathcal{I}$ be a descending sequence of interpretations. As before, we define the restriction of u to a predicate symbol $p \in \Pi$ by $\big(u|_p\big)_\alpha = u_\alpha|_p = \big\{\, \vec{t} \mid p\big(\vec{t}\big) \in u_\alpha \big\}$. The semantics of formulae in $\mathbf{iFOL}_\blacktriangleright$ as objects in $\overline{\mathbf{Pred}}$ is given by the following iterative definition.

$$\begin{aligned}
[\![\Gamma \Vdash p\,\vec{M}]\!]_u &= \Big(\overrightarrow{[\![M]\!]}\Big)^{\#}\big(u|_p\big) \\
[\![\Gamma \Vdash \top]\!]_u &= \dot{\top}_{[\![\Gamma]\!]} \\
[\![\Gamma \Vdash \varphi \,\square\, \psi]\!]_u &= [\![\Gamma \Vdash \varphi]\!]_u \,\square\, [\![\Gamma \Vdash \psi]\!]_u && \square \in \{\wedge, \vee, \to\} \\
[\![\Gamma \Vdash Qx : \tau.\, \varphi]\!]_u &= Q_{[\![\Gamma]\!],[\![\tau]\!]}\, [\![\Gamma, x : \tau \Vdash \varphi]\!]_u && Q \in \{\forall, \exists\} \\
[\![\Gamma \Vdash \blacktriangleright \varphi]\!]_u &= \blacktriangleright [\![\Gamma \Vdash \varphi]\!]_u
\end{aligned}$$

The following lemma is the analogue of Lemma 28 for the interpretation of formulae without the later modality.

Lemma 36. *The mapping* $[\![-]\!]_u$ *is a well-defined map from formulae in* $\mathbf{iFOL}_\blacktriangleright$ *to sequences of predicates, such that* $\Gamma \Vdash \varphi$ *implies* $[\![\varphi]\!]_u \in \overline{\mathbf{Pred}}_{[\![\Gamma]\!]}$.

Lemma 37. *All rules of* $\mathbf{iFOL}_\blacktriangleright$ *are sound with respect to the interpretation* $[\![-]\!]_u$ *of formulae in* $\overline{\mathbf{Pred}}$, *that is, if* $\Gamma \mid \Delta \vdash \varphi$, *then* $\big(\bigwedge_{\psi\in\Delta} [\![\psi]\!]_u \dot{\Rightarrow} [\![\varphi]\!]_u\big) = \dot{\top}$. *In particular,* $\Gamma \vdash \varphi$ *implies* $[\![\varphi]\!]_u = \dot{\top}$.

The following lemma shows that the guarding of a set of formulae is valid in the chain model that they generate.

Lemma 38. *If* φ *is an* H*-formula in* P, *then* $[\![\overline{\varphi}]\!]_{\overleftarrow{\Phi_P}} = \dot{\top}$.

Combining this with soundness from Lemma 37, we obtain that provability in $\mathbf{iFOL}_\blacktriangleright$ relative to a logic program P is sound for the model of P.

Theorem 39. *For all logic programs* P, *if* $\Gamma \mid \overline{P} \vdash \varphi$ *then* $[\![\varphi]\!]_{\overleftarrow{\Phi_P}} = \dot{\top}$.

The final result of this section is to show that the descending chain model, which we used to interpret formulae of $\mathbf{iFOL}_\blacktriangleright$, is sound and complete for the fixed point model, which we used to interpret the formulae of coinductive uniform proofs. This will be proved in Theorem 42 below. The easiest way to prove this result is by establishing a functor $\overline{\mathbf{Pred}} \to \mathbf{Pred}$ that maps the chain $\overleftarrow{\Phi_P}$ to the model $\mathcal{M}_P$, and that preserves and reflects truth of first-order formulae (Proposition 41). We will phrase the preservation of truth of first-order formulae by a functor by appealing to the following notion of fibrations maps, cf. [46, Def. 4.3.1].

Definition 40. Let $p\colon \mathbf{E} \to \mathbf{B}$ and $q\colon \mathbf{D} \to \mathbf{A}$ be fibrations. A *fibration map* $p \to q$ is a pair $(F\colon \mathbf{E} \to \mathbf{D}, G\colon \mathbf{B} \to \mathbf{A})$ of functors, s.t. $q \circ F = G \circ p$ and F preserves Cartesian morphisms: if $f\colon X \to Y$ in $\mathbf{E}$ is Cartesian over $p(f)$, then $F(f)$ is Cartesian over $G(p(f))$. (F, G) is a map of *first-order (λ-)fibrations*, if p and q are first-order (λ-)fibrations, and F and G preserve this structure.

Let us now construct a first-order λ-fibration map $\overline{\mathbf{Pred}} \to \mathbf{Pred}$. We note that since every fibre of the predicate fibration is a complete lattice, for every chain $u \in \overline{\mathbf{Pred}}_X$ there exists an ordinal α at which u stabilises. This means that there is a limit $\lim u$ of u in $\mathbf{Pred}_X$, which is the largest subset of X, such that $\forall \alpha.\ \lim u \subseteq u_\alpha$. This allows us to define a map $L\colon \overline{\mathbf{Pred}} \to \mathbf{Pred}$ by

$$L(X, u) = (X, \lim u)$$
$$L(f\colon (X, u) \to (Y, v)) = f.$$

In the following proposition, we show that L gives us the ability to express first-order properties of limits equivalently through their approximating chains. This, in turn, provides soundness and completeness for the interpretation of the logic $\mathbf{iFOL}_\blacktriangleright$ over descending chains with respect to the largest Herbrand model.

Proposition 41. *$L\colon \overline{\mathbf{Pred}} \to \mathbf{Pred}$, as defined above, is a map of first-order fibrations. Furthermore, L is right-adjoint to the embedding $K\colon \mathbf{Pred} \to \overline{\mathbf{Pred}}$. Finally, for each $p \in \Pi$ and $u \in \overline{\mathbf{Pred}}_{\mathcal{B}^\infty}$, we have $L\big(u|_p\big) = L(u)|_p$.*

We get from Proposition 41 soundness and completeness of $\overleftarrow{\Phi_P}$ for Herbrand models. More precisely, if φ is a formula of plain first-order logic ($\blacktriangleright$-free), then its interpretation in the coinductive Herbrand model is true if and only if its interpretation over the chain approximation of the Herbrand model is true.

Theorem 42. *If φ is $\blacktriangleright$-free (Definition 3) then $[\![\varphi]\!]_{\overleftarrow{\Phi_P}} = \dot{\top}$ if and only if $[\![\varphi]\!]_{\mathcal{M}_P} = \top$.*

Proof (sketch). First, one shows for all $\blacktriangleright$-free formulae φ that $L([\![\varphi]\!]_{\overleftarrow{\Phi_P}}) = [\![\varphi]\!]_{\mathcal{M}_P}$ by induction on φ and using Proposition 41. Using this identity and $K \dashv L$, the result is then obtained from the following adjoint correspondence.

$$\frac{\dot{\top} = K(\top) \longrightarrow [\![\varphi]\!]_{\overleftarrow{\Phi_P}} \quad \text{in } \overline{\mathbf{Pred}}}{\top \longrightarrow L\big([\![\varphi]\!]_{\overleftarrow{\Phi_P}}\big) = [\![\varphi]\!]_{\mathcal{M}_P} \quad \text{in } \mathbf{Pred}} \qquad \square$$

6 Conclusion, Related Work and the Future

In this paper, we provided a comprehensive theory of resolution in coinductive Horn-clause theories and coinductive logic programs. This theory comprises of a uniform proof system that features a form of guarded recursion and that provides

operational semantics for proofs of coinductive predicates. Further, we showed how to translate proofs in this system into proofs for an extension of intuitionistic FOL with guarded recursion, and we provided sound semantics for both proof systems in terms of coinductive Herbrand models. The Herbrand models and semantics were thereby presented in a modern style that utilises coalgebras and fibrations to provide a conceptual view on the semantics.

Related Work. It may be surprising that automated *proof search for coinductive predicates* in first-order logic does not have a coherent and comprehensive theory, even after three decades [3,60], despite all the attention that it received as programming [2,29,42,44] and proof [33,35,39,40,45,59,64–67] method. The work that comes close to algorithmic proof search is the system CIRC [63], but it cannot handle general coinductive predicates and corecursive programming. Inductive and coinductive data types are also being added to SMT solvers [24,62]. However, both CIRC and SMT solving are inherently based on classical logic and are therefore not suited to situations where proof objects are relevant, like programming, type class inference or (dependent) type theory. Moreover, the proposed solutions, just like those in [41,69] can only deal with regular data, while our approach also works for irregular data, as we saw in the **from**-example.

This paper subsumes Haskell type class inference [37,51] and exposes that the inference presented in those papers corresponds to coinductive proofs in *co-fohc* and *co-hohh*. Given that the proof systems proposed in this paper are constructive and that uniform proofs provide proofs (type inhabitants) in normal form, we could give a propositions-as-types interpretation to all eight coinductive uniform proof systems. This was done for *co-fohc* and *co-hohh* in [37], but we leave the remaining cube from the introduction for future work.

Future Work. There are several directions that we wish to pursue in the future. First, we know that CUP is incomplete for the presented models, as it is intuitionistic and it lacks an admissible cut rule. The first can be solved by moving to Kripke/Beth-models, as done by Clouston and Goré [30] for the propositional part of $\mathbf{iFOL}_{\blacktriangleright}$. However, the admissible cut rule is more delicate. To obtain such a rule one has to be able to prove several propositions simultaneously by coinduction, as discussed at the end of Sect. 4. In general, completeness of recursive proof systems depends largely on the theory they are applied to, see [70] and [18]. However, techniques from cyclic proof systems [27,68] may help. We also aim to extend our ideas to other situations like higher-order Horn clauses [28,43] and interactive proof assistants [7,10,23,31], typed logic programming, and logic programming that mix inductive and coinductive predicates.

Acknowledgements. We would like to thank Damien Pous and the anonymous reviewers for their valuable feedback.

References

1. Abbott, M., Altenkirch, T., Ghani, N.: Containers: constructing strictly positive types. TCS **342**(1), 3–27 (2005). https://doi.org/10.1016/j.tcs.2005.06.002
2. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: programming infinite structures by observations. In: POPL 2013, pp. 27–38 (2013). https://doi.org/10.1145/2429069.2429075
3. Aczel, P.: Non-well-founded sets. Center for the Study of Language and Information, Stanford University (1988)
4. Aczel, P.: Algebras and coalgebras. In: Backhouse, R., Crole, R., Gibbons, J. (eds.) Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. LNCS, vol. 2297, pp. 79–88. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47797-7_3
5. Aczel, P., Adámek, J., Milius, S., Velebil, J.: Infinite trees and completely iterative theories: a coalgebraic view. TCS **300**(1–3), 1–45 (2003). https://doi.org/10.1016/S0304-3975(02)00728-4
6. Adámek, J.: On final coalgebras of continuous functors. Theor. Comput. Sci. **294**(1/2), 3–29 (2003). https://doi.org/10.1016/S0304-3975(01)00240-7
7. P.L. group on Agda: Agda Documentation. Technical report, Chalmers and Gothenburg University (2015). http://wiki.portal.chalmers.se/agda/, version 2.4.2.5
8. Appel, A.W., Melliès, P.A., Richards, C.D., Vouillon, J.: A very modal model of a modern, major, general type system. In: POPL, pp. 109–122. ACM (2007). https://doi.org/10.1145/1190216.1190235
9. Atkey, R., McBride, C.: Productive coprogramming with guarded recursion. In: ICFP, pp. 197–208. ACM (2013). https://doi.org/10.1145/2500365.2500597
10. Baelde, D., et al.: Abella: a system for reasoning about relational specifications. J. Formaliz. Reason. **7**(2), 1–89 (2014). https://doi.org/10.6092/issn.1972-5787/4650
11. Barendregt, H., Dekkers, W., Statman, R.: Lambda Calculus with Types. Cambridge University Press, Cambridge (2013)
12. Barr, M., Wells, C.: Category Theory for Computing Science. Prentice Hall International Series in Computer Science, 2nd edn. Prentice Hall, Upper Saddle River (1995). http://www.tac.mta.ca/tac/reprints/articles/22/tr22abs.html
13. Basold, H.: Mixed inductive-coinductive reasoning: types, programs and logic. Ph.D. thesis, Radboud University Nijmegen (2018). http://hdl.handle.net/2066/190323
14. Basold, H.: Breaking the Loop: Recursive Proofs for Coinductive Predicates in Fibrations. ArXiv e-prints, February 2018. https://arxiv.org/abs/1802.07143
15. Basold, H., Komendantskaya, E., Li, Y.: Coinduction in uniform: foundations for corecursive proof search with horn clauses. Extended version of this paper. CoRR abs/1811.07644 (2018). http://arxiv.org/abs/1811.07644
16. Beklemishev, L.D.: Parameter free induction and provably total computable functions. TCS **224**(1–2), 13–33 (1999). https://doi.org/10.1016/S0304-3975(98)00305-3
17. Bénabou, J.: Fibered categories and the foundations of naive category theory. J. Symb. Logic **50**(1), 10–37 (1985). https://doi.org/10.2307/2273784
18. Berardi, S., Tatsuta, M.: Classical system of Martin-Löf's inductive definitions is not equivalent to cyclic proof system. In: Esparza, J., Murawski, A.S. (eds.) FoSSaCS 2017. LNCS, vol. 10203, pp. 301–317. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54458-7_18

19. Birkedal, L., Møgelberg, R.E.: Intensional type theory with guarded recursive types qua fixed points on universes. In: LICS, pp. 213–222. IEEE Computer Society (2013). https://doi.org/10.1109/LICS.2013.27
20. Birkedal, L., Møgelberg, R.E., Schwinghammer, J., Støvring, K.: First steps in synthetic guarded domain theory: step-indexing in the topos of trees. In: Proceedings of LICS 2011, pp. 55–64. IEEE Computer Society (2011). https://doi.org/10.1109/LICS.2011.16
21. Bizjak, A., Grathwohl, H.B., Clouston, R., Møgelberg, R.E., Birkedal, L.: Guarded dependent type theory with coinductive types. In: Jacobs, B., Löding, C. (eds.) FoSSaCS 2016. LNCS, vol. 9634, pp. 20–35. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49630-5_2. https://arxiv.org/abs/1601.01586
22. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
23. Blanchette, J.C., Meier, F., Popescu, A., Traytel, D.: Foundational nonuniform (co)datatypes for Higher-Order Logic. In: LICS 2017, pp. 1–12. IEEE Computer Society (2017). https://doi.org/10.1109/LICS.2017.8005071
24. Blanchette, J.C., Peltier, N., Robillard, S.: Superposition with datatypes and codatatypes. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 370–387. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_25
25. Borceux, F.: Handbook of Categorical Algebra. Basic Category Theory, vol. 1. Cambridge University Press, Cambridge (2008)
26. Bottu, G., Karachalias, G., Schrijvers, T., Oliveira, B.C.D.S., Wadler, P.: Quantified class constraints. In: Haskell Symposium, pp. 148–161. ACM (2017). https://doi.org/10.1145/3122955.3122967
27. Brotherston, J., Simpson, A.: Sequent calculi for induction and infinite descent. J. Log. Comput. **21**(6), 1177–1216 (2011). https://doi.org/10.1093/logcom/exq052
28. Burn, T.C., Ong, C.L., Ramsay, S.J.: Higher-order constrained horn clauses for verification. PACMPL **2**(POPL), 11:1–11:28 (2018). https://doi.org/10.1145/3158099
29. Capretta, V.: General Recursion via Coinductive Types. Log. Methods Comput. Sci. **1**(2), July 2005. https://doi.org/10.2168/LMCS-1(2:1)2005
30. Clouston, R., Goré, R.: Sequent calculus in the topos of trees. In: Pitts, A. (ed.) FoSSaCS 2015. LNCS, vol. 9034, pp. 133–147. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46678-0_9
31. Coquand, T.: Infinite objects in type theory. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 62–78. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58085-9_72
32. Cousot, P., Cousot, R.: Constructive versions of Tarski's fixed point theorems. Pac. J. Math. **82**(1), 43–57 (1979). http://projecteuclid.org/euclid.pjm/1102785059
33. Dax, C., Hofmann, M., Lange, M.: A proof system for the linear time μ-calculus. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 273–284. Springer, Heidelberg (2006). https://doi.org/10.1007/11944836_26
34. van Emden, M., Kowalski, R.: The semantics of predicate logic as a programming language. J. Assoc. Comput. Mach. **23**, 733–742 (1976). https://doi.org/10.1145/321978.321991
35. Endrullis, J., Hansen, H.H., Hendriks, D., Polonsky, A., Silva, A.: A coinductive framework for infinitary rewriting and equational reasoning. In: RTA 2015, pp. 143–159 (2015). https://doi.org/10.4230/LIPIcs.RTA.2015.143

36. Farka, F., Komendantskaya, E., Hammond, K.: Coinductive soundness of corecursive type class resolution. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 311–327. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_18
37. Fu, P., Komendantskaya, E., Schrijvers, T., Pond, A.: Proof relevant corecursive resolution. In: Kiselyov, O., King, A. (eds.) FLOPS 2016. LNCS, vol. 9613, pp. 126–143. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29604-3_9
38. Gambino, N., Kock, J.: Polynomial functors and polynomial monads. Math. Proc. Cambridge Phil. Soc. **154**(1), 153–192 (2013). https://doi.org/10.1017/S0305004112000394
39. Giesl, J., et al.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reason. **58**(1), 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y
40. Giménez, E.: Structural recursive definitions in type theory. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 397–408. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055070
41. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive logic programming and its applications. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 27–44. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74610-2_4
42. Hagino, T.: A typed lambda calculus with categorical type constructors. In: Pitt, D.H., Poigné, A., Rydeheard, D.E. (eds.) Category Theory and Computer Science. LNCS, vol. 283, pp. 140–157. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18508-9_24
43. Hashimoto, K., Unno, H.: Refinement type inference via horn constraint optimization. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 199–216. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9_12
44. Howard, B.T.: Inductive, coinductive, and pointed types. In: Harper, R., Wexelblat, R.L. (eds.) Proceedings of ICFP 1996, pp. 102–109. ACM (1996). https://doi.org/10.1145/232627.232640
45. Hur, C.K., Neis, G., Dreyer, D., Vafeiadis, V.: The power of parameterization in coinductive proof. In: Proceedings of POPL 2013, pp. 193–206. ACM (2013). https://doi.org/10.1145/2429069.2429093
46. Jacobs, B.: Categorical Logic and Type Theory. Studies in Logic and the Foundations of Mathematics, vol. 141. North Holland, Amsterdam (1999)
47. Jacobs, B.: Introduction to Coalgebra: Towards Mathematics of States and Observation. Cambridge Tracts in Theoretical Computer Science, vol. 59. Cambridge University Press, Cambridge (2016). https://doi.org/10.1017/CBO9781316823187. http://www.cs.ru.nl/B.Jacobs/CLG/JacobsCoalgebraIntro.pdf
48. Komendantskaya, E., Li, Y.: Productive corecursion in logic programming. J. TPLP (ICLP 2017 post-proc.) **17**(5–6), 906–923 (2017). https://doi.org/10.1017/S147106841700028X
49. Komendantskaya, E., Li, Y.: Towards coinductive theory exploration in horn clause logic: Position paper. In: Kahsai, T., Vidal, G. (eds.) Proceedings 5th Workshop on Horn Clauses for Verification and Synthesis, HCVS 2018, Oxford, UK, 13th July 2018, vol. 278, pp. 27–33 (2018). https://doi.org/10.4204/EPTCS.278.5
50. Lambek, J., Scott, P.J.: Introduction to Higher-Order Categorical Logic. Cambridge University Press, Cambridge (1988)
51. Lämmel, R., Peyton Jones, S.L.: Scrap your boilerplate with class: extensible generic functions. In: ICFP 2005, pp. 204–215. ACM (2005). https://doi.org/10.1145/1086365.1086391

52. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987). https://doi.org/10.1007/978-3-642-83189-8
53. Miller, D., Nadathur, G.: Programming with Higher-order logic. Cambridge University Press, Cambridge (2012)
54. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform proofs as a foundation for logic programming. Ann. Pure Appl. Logic **51**(1–2), 125–157 (1991). https://doi.org/10.1016/0168-0072(91)90068-W
55. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978). https://doi.org/10.1016/0022-0000(78)90014-4
56. Møgelberg, R.E.: A type theory for productive coprogramming via guarded recursion. In: CSL-LICS, pp. 71:1–71:10. ACM (2014). https://doi.org/10.1145/2603088.2603132
57. Nadathur, G., Mitchell, D.J.: System description: Teyjus—a compiler and abstract machine based implementation of λProlog. CADE-16. LNCS (LNAI), vol. 1632, pp. 287–291. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48660-7_25
58. Nakano, H.: A modality for recursion. In: LICS, pp. 255–266. IEEE Computer Society (2000). https://doi.org/10.1109/LICS.2000.855774
59. Niwinski, D., Walukiewicz, I.: Games for the μ-Calculus. TCS **163**(1&2), 99–116 (1996). https://doi.org/10.1016/0304-3975(95)00136-0
60. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981). https://doi.org/10.1007/BFb0017309
61. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. **5**(3), 223–255 (1977). https://doi.org/10.1016/0304-3975(77)90044-5
62. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 80–98. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_5
63. Roşu, G., Lucanu, D.: Circular coinduction: a proof theoretical foundation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 127–144. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03741-2_10
64. Rutten, J.: Universal coalgebra: a theory of systems. TCS **249**(1), 3–80 (2000). https://doi.org/10.1016/S0304-3975(00)00056-6
65. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press, New York (2011)
66. Santocanale, L.: A calculus of circular proofs and its categorical semantics. In: Nielsen, M., Engberg, U. (eds.) FoSSaCS 2002. LNCS, vol. 2303, pp. 357–371. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45931-6_25
67. Santocanale, L.: μ-bicomplete categories and parity games. RAIRO - ITA **36**(2), 195–227 (2002). https://doi.org/10.1051/ita:2002010
68. Shamkanov, D.S.: Circular proofs for the Gödel-Löb provability logic. Math. Notes **96**(3), 575–585 (2014). https://doi.org/10.1134/S0001434614090326
69. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-logic programming: extending logic programming with coinduction. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 472–483. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73420-8_42
70. Simpson, A.: Cyclic arithmetic is equivalent to Peano arithmetic. In: Esparza, J., Murawski, A.S. (eds.) FoSSaCS 2017. LNCS, vol. 10203, pp. 283–300. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54458-7_17
71. Smoryński, C.: Self-Reference and Modal Logic. Universitext. Springer, New York (1985). https://doi.org/10.1007/978-1-4613-8601-8

72. Solovay, R.M.: Provability interpretations of modal logic. Israel J. Math. **25**(3), 287–304 (1976). https://doi.org/10.1007/BF02757006
73. Sulzmann, M., Stuckey, P.J.: HM(X) type inference is CLP(X) solving. J. Funct. Program. **18**(2), 251–283 (2008). https://doi.org/10.1017/S0956796807006569
74. Terese: Term Rewriting Systems. Cambridge University Press, Cambridge (2003)
75. Turner, D.A.: Elementary strong functional programming. In: Hartel, P.H., Plasmeijer, R. (eds.) FPLE 1995. LNCS, vol. 1022, pp. 1–13. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60675-0_35
76. van den Berg, B., de Marchi, F.: Non-well-founded trees in categories. Ann. Pure Appl. Logic **146**(1), 40–59 (2007). https://doi.org/10.1016/j.apal.2006.12.001
77. Worrell, J.: On the final sequence of a finitary set functor. Theor. Comput. Sci. **338**(1–3), 184–199 (2005). https://doi.org/10.1016/j.tcs.2004.12.009

Author Index