Modular Product Programs

Marco Eilers(✉), Peter Müller, and Samuel Hitz

Department of Computer Science, ETH Zurich, Zurich, Switzerland
{marco.eilers,peter.mueller,samuel.hitz}@inf.ethz.ch

Abstract. Many interesting program properties like determinism or information flow security are hyperproperties, that is, they relate multiple executions of the same program. Hyperproperties can be verified using relational logics, but these logics require dedicated tool support and are difficult to automate. Alternatively, constructions such as self-composition represent multiple executions of a program by one product program, thereby reducing hyperproperties of the original program to trace properties of the product. However, existing constructions do not fully support procedure specifications, for instance, to derive the determinism of a caller from the determinism of a callee, making verification non-modular.

We present modular product programs, a novel kind of product program that permits hyperproperties in procedure specifications and, thus, can reason about calls modularly. We demonstrate its expressiveness by applying it to information flow security with advanced features such as declassification and termination-sensitivity. Modular product programs can be verified using off-the-shelf verifiers; we have implemented our approach to secure information flow using the Viper verification infrastructure.

1 Introduction

The past decades have seen significant progress in automated reasoning about program behavior. In the most common scenario, the goal is to prove trace properties of programs such as functional correctness or termination. However, important program properties such as information flow security, injectivity, and determinism cannot be expressed as properties of individual traces; these so-called *hyperproperties* relate different executions of the same program. For example, proving determinism of a program requires showing that any two executions from identical initial states will result in identical final states.

An important attribute of reasoning techniques about programs is *modularity*. A technique is modular if it allows reasoning about parts of a program in isolation, e.g., verifying each procedure separately and using only the *specifications* of other procedures. Modularity is vital for scalability and to verify libraries without knowing all of their clients. Fully modular reasoning about hyperproperties thus requires the ability to formulate *relational* specifications, which relate

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 502–529, 2018.
https://doi.org/10.1007/978-3-319-89884-1_18

different executions of a procedure, and to apply those specifications where the procedure is called. As an example, the statement

$$\texttt{if (x) then \{y:=x\} else \{y:= call f(x)\}}$$

can be proved to be deterministic if f's relational specification guarantees that its result deterministically depends on its input.

Relational program logics [11,27,29] allow directly proving general hyperproperties, however, automating relational logics is difficult and requires building dedicated tools. Alternatively, self-composition [9] and product programs [6,7] reduce a hyperproperty to an ordinary trace property, thus making it possible to use off-the-shelf program verifiers for proving hyperproperties. Both approaches construct a new program that combines the behaviors of multiple runs of the original program. However, by the nature of their construction, neither approach supports modular verification based on relational specifications: Procedure calls in the original program will be duplicated, which means that there is no single program point at which a relational specification can be applied. For the aforementioned example, self-composition yields the following program:

$$\begin{aligned}&\texttt{if (x) then \{y:=x\} else \{y:= call f(x)\};}\\&\texttt{if (x') then \{y':=x'\} else \{y':= call f(x')\}}\end{aligned}$$

Determinism can now be verified by proving the trace property that identical values for x and x' in the initial state imply identical values for y and y' in the final state. However, such a proof cannot make use of a relational specification for procedure f (expressing that f is deterministic). Such a specification relates several executions of f, whereas each call in the self-composition belongs to a single execution. Instead, verification requires a *precise functional specification* of f, which *exactly* determines its result value in terms of the input. Verifying such precise functional specifications increases the verification effort and is at odds with data abstraction (for instance, a collection might not want to promise the exact iteration order); inferring them is beyond the state of the art for most procedures [28]. Existing product programs allow aligning or combining some statements and can thereby lift this requirement in some cases, but this requires manual effort during the construction, depends on the used specifications, and does not solve the problem in general.

In this paper, we present modular product programs, a novel kind of product programs that allows modular reasoning about hyperproperties. Modular product programs enable proving k-safety hyperproperties, i.e., hyperproperties that relate finite prefixes of k execution traces, for arbitrary values of k [12]. We achieve this via a transformation that, unlike existing products, does not duplicate loops or procedure calls, meaning that for any loop or call in the original program, there is exactly one statement in the k-product at which a relational specification can be applied. Like existing product programs, modular products can be reasoned about using off-the-shelf program verifiers.

We demonstrate the expressiveness of modular product programs by applying them to prove secure information flow, a 2-safety hyperproperty. We show

how modular products enable proving traditional non-interference using natural and concise information flow specifications, and how to extend our approach for proving the absence of timing or termination channels, and supporting declassification in an intuitive way.

To summarize, we make the following contributions:

- We introduce modular k-product programs, which enable modular proofs of arbitrary k-safety hyperproperties for sequential programs using off-the-shelf verifiers.
- We demonstrate the usefulness of modular product programs by applying them to secure information flow, with support for declassification and preventing different kinds of side channels.
- We implement our product-based approach for information flow verification in an automated verifier and show that our tool can automatically prove information flow security of challenging examples.

After giving an informal overview of our approach in Sect. 2 and introducing our programming and assertion language in Sect. 3, we formally define modular product programs in Sect. 4. We sketch a soundness proof in Sect. 5. Section 6 demonstrates how to apply modular products for proving secure information flow. We describe and evaluate our implementation in Sect. 7, discuss related work in Sect. 8, and conclude in Sect. 9.

2 Overview

In this section, we will illustrate the core concepts behind modular k-products on an example program. We will first show how modular products are constructed, and subsequently demonstrate how they allow using relational specifications to modularly prove hyperproperties.

2.1 Relational Specifications

Consider the example program in Fig. 1, which counts the number of female entries in a sequence of people. Now assume we want to prove that the program is deterministic, i.e., that its output state is completely determined by its input arguments. This can be expressed as a 2-safety hyperproperty which states that, for two terminating executions of the program with identical inputs, the outputs will be the same. This hyperproperty can be expressed by the *relational* (as opposed to *unary*) specification $\mathsf{main} : \overset{1}{\mathsf{people}} = \overset{2}{\mathsf{people}} \twoheadrightarrow \overset{1}{\mathsf{count}} = \overset{2}{\mathsf{count}}$, where $\overset{i}{x}$ refers to the value of the variable x in the ith execution.

Intuitively, it is possible to prove this specification by giving is_female a precise functional specification like $\mathsf{is\_female} : \mathit{true} \rightsquigarrow \mathsf{res} = 1 - \mathsf{person}\ \mathbf{mod}\ 2$, meaning that is_female can be invoked in any state and that $\mathsf{res} = 1 - \mathsf{person}\ \mathbf{mod}\ 2$ will hold if it returns. From this specification and an appropriate loop invariant, main can be shown to be deterministic. However, this specification

```
procedure main(people)
          returns (count)
{
  i := 0;
  count := 0;
  while (i < |people|) {
    current := people[i];
    f := is_female(current);
    count := count + f;
    i := i + 1;
  }
}
```

```
procedure is_female(person)
          returns (res)
{
  // gender encoded in first bit
  gender := person mod 2;
  if (gender == 0) {
    res := 1;
  }else{
    res := 0;
  }
}
```

Fig. 1. Example program. The parameter people contains a sequence of integers that each encode attributes of a person; the main procedure counts the number of females in this sequence.

is unnecessarily strong. For proving determinism, it is irrelevant what exactly the final value of count is; it is only important that it is uniquely determined by the procedure's inputs. Proving hyperproperties using only unary specifications, however, critically depends on having exact specifications for every value returned by a called procedure, as well as all heap locations modified by it. Not only are such specifications difficult to infer and cumbersome to provide manually; this requirement also fundamentally removes the option of underspecifying program behavior, which is often desirable in practice. Because of these limitations, verification techniques that require precise functional specifications for proving hyperproperties often do not work well in practice, as observed by Terauchi and Aiken for the case of self-composition [28].

Proving determinism of the example program becomes much simpler if we are able to reason about two program executions at once. If both runs start with identical values for people then they will have identical values for people, i, and count when they reach the loop. Since the loop guard only depends on i and people, it will either be true for both executions or false for both. Assuming that is_female behaves deterministically, all three variables will again be equal in both executions at the end of the loop body. This means that the program establishes and preserves the relational loop invariant that people, i, and count have identical values in both executions, from which we can deduce the desired relational postcondition. Our modular product programs enable this modular and intuitive reasoning, as we explain next.

2.2 Modular Product Programs

Like other product programs, our modular k-product programs multiply the state space of the original program by creating k renamed versions of all original variables. However, unlike other product programs, they do *not* duplicate control structures like loops or procedure calls, while still allowing different executions to take different paths through the program.

Modular product programs achieve this as follows: The set of transitions made by the execution of a product is the union of the transitions made by

```
procedure main(p1, p2, people1, people2)
          returns (count1, count2)
{
  if (p1) { i1 := 0; }
  if (p2) { i2 := 0; }
  if (p1) { count1 := 0; }
  if (p2) { count2 := 0; }
  while ((p1 && i1 < |people1|) ||
         (p2 && i2 < |people2|)) {
    l1 := p1 && i1 < |people1|;
    l2 := p2 && i2 < |people2|;
    if (l1) { current1 := people1[i1]; }
    if (l2) { current2 := people2[i2]; }
    if (l1 || l2) {
      t1, t2 := is_female(l1, l2,
                current1, current2);
    }
    if (l1) { f1 := t1; }
    if (l2) { f2 := t2; }
    if (l1) { count1 := count1 + f1; }
    if (l2) { count2 := count2 + f2; }
    if (l1) { i1 := i1 + 1; }
    if (l2) { i2 := i2 + 1; }
  }
}
```

```
procedure is_female(p1, p2,
                    person1,
                    person2)
          returns (res1, res2)
{
  if (p1) {
    gender1 := person1 mod 2;
  }
  if (p2) {
    gender2 := person2 mod 2;
  }
  t1 := p1 && gender1 == 0;
  t2 := p2 && gender2 == 0;
  f1 := p1 && !(gender1 == 0);
  f2 := p2 && !(gender2 == 0);
  if (t1) { res1 := 1; }
  if (t2) { res2 := 1; }
  if (f1) { res1 := 0; }
  if (f2) { res2 := 0; }
}
```

Fig. 2. Modular 2-product of the program in Fig. 1 (slightly simplified). Parameters and local variables have been duplicated, but control flow statements have not. All statements are parameterized by activation variables.

the executions of the original program it represents. This means that if two executions of an if-then-else statement execute different branches, an execution of the product will execute the corresponding versions of *both* branches; however, it will be aware of the fact that each branch is taken by only one of the original executions, and the transformation of the statements *inside* each branch will ensure that the state of the other execution is not modified by executing it.

For this purpose, modular product programs use boolean *activation variables* that store, for each execution, the condition under which it is currently active. All activation variables are initially true. For every statement that directly changes the program state, the product performs the state change for all active executions. Control structures update which executions are active (for instance based on the loop condition) and pass this information down (into the branches of a conditional, the body of a loop, or the callee of a procedure call) to the level of atomic statements[1]. This representation avoids duplicating these control structures.

Figure 2 shows the modular 2-product of the program in Fig. 1. Consider first the main procedure. Its parameters have been duplicated, there are now two copies of all variables, one for each execution. This is analogous to self-composition or existing product programs. In addition, the transformed procedure has two boolean parameters p1 and p2; these variables are the initial

[1] The information stored in activation variables is similar to a path condition in symbolic execution, which is also updated every time a branch is taken. However, they differ for loops and calls.

activation variables of the procedure. Since main is the entry point of the program, the initial activation variables can be assumed to be true.

Consider what happens when the product is run with arbitrary input values for people1 and people2. The product will first initialize i1 and i2 to zero, like it does with i in the original program, and analogously for count1 and count2.

The loop in the original program has been transformed to a single loop in the product. Its condition is true if the original loop condition is true for any active execution. This means that the loop will iterate as long as at least one execution of the original program would. Inside the loop body, the fresh activation variables l1 and l2 represent whether the corresponding executions would execute the loop body. That is, for each execution, the respective activation variable will be true if the previous activation variable (p1 or p2, respectively) is true, meaning that this execution actually reaches the loop, and the loop guard is true for that execution. All statements in the loop body are then transformed using these new activation variables. Consequently, the loop will keep iterating while at least one execution executes the loop, but as soon as the loop guard is false for any execution, its activation variable will be false and the loop body will have no effect.

Conceptually, procedure calls are handled very similarly to loops. For the call to is_female in the original program, only a single call is created in the product. This call is executed if at least one activation variable is true, i.e., if at least one execution would perform the call in the original program. In addition to the (duplicated) arguments of the original call, the current activation variables are passed to the called procedure. In the transformed version of is_female, all statements are then made conditional on those activation variables. Therefore, like with loops, a call in the product will be performed if at least one execution would perform it in the original program, but it will have no effect on the state of the executions that are not active when the call is made.

The transformed version of is_female shows how conditionals are handled. We introduce four fresh activation variables t1, t2, f1, and f2, two for each execution. The first pair encodes whether the then-branch should be executed by either of the two executions; the second encodes the same for the else-branch. These activation variables are then used to transform the branches. Consequently, neither branch will have an effect for inactive executions, and exactly one branch has an effect for each active execution.

To summarize, our activation variables ensure that the sequence of state-changing statements executed by each execution is the same in the product and the original program. We achieve this without duplicating control structures or imposing restrictions on the control flow.

2.3 Interpretation of Relational Specifications

Since modular product programs do not duplicate calls, they provide a simple way of interpreting relational procedure specifications: If all executions call a procedure, its relational precondition is required to hold before the call and the relational postcondition afterwards. If a call is performed by some executions but not all, the relational specification are not meaningful, and thus cannot be

required to hold. To encode this intuition, we transform every relational pre- or postcondition $\hat{Q}$ of the original program into an implication $(\bigwedge_{i=1}^{k} \mathsf{p}_i) \Rightarrow \hat{Q}$. In the transformed version, both pre- and postconditions are made conditional on the conjunction of all activation parameters p_i of the procedure. As a result, both will be trivially true if at least one execution is not active at the call site.

In our example, we give is_female the relational specification is_female : $\mathit{true} \twoheadrightarrow \overset{1}{\mathsf{person}} = \overset{2}{\mathsf{person}} \Rightarrow \overset{1}{\mathsf{res}} = \overset{2}{\mathsf{res}}$, which expresses determinism. This specification will be transformed into a unary specification of the product program: is_female : $\mathsf{p1} \wedge \mathsf{p2} \Rightarrow \mathit{true} \rightsquigarrow \mathsf{p1} \wedge \mathsf{p2} \Rightarrow (\mathsf{person1} = \mathsf{person2} \Rightarrow \mathsf{res1} = \mathsf{res2})$.

Assume for the moment that is_female also has a unary precondition person ≥ 0. Such a specification should hold for *every* call, and therefore for every active execution, even if other executions are inactive. Therefore, its interpretation in the product program is $(\mathsf{p1} \Rightarrow \mathsf{person1} \geq 0) \wedge (\mathsf{p2} \Rightarrow \mathsf{person2} \geq 0)$. The translation of other unary assertions is analogous.

Note that it is possible (and useful) to give a procedure both a relational and a unary specification; in the product this is encoded by simply conjoining the transformed versions of the unary and the relational assertions.

2.4 Product Program Verification

We can now prove determinism of our example using the product program. Verifying is_female is simple. For main, we want to prove the transformed specification main : $(\mathsf{p1} \wedge \mathsf{p2} \Rightarrow \mathsf{people1} = \mathsf{people2}) \rightsquigarrow (\mathsf{p1} \wedge \mathsf{p2} \Rightarrow \mathsf{count1} = \mathsf{count2})$. We use the relational loop invariant $\overset{1}{\mathsf{i}} = \overset{2}{\mathsf{i}} \wedge \overset{1}{\mathsf{count}} = \overset{2}{\mathsf{count}} \wedge \overset{1}{\mathsf{people}} = \overset{2}{\mathsf{people}}$, encoded as $\mathsf{p1} \wedge \mathsf{p2} \Rightarrow \mathsf{i1} = \mathsf{i2} \wedge \mathsf{count1} = \mathsf{count2} \wedge \mathsf{people1} = \mathsf{people2}$. The loop invariant holds trivially if either p1 or p2 is false. Otherwise, it ensures $\mathsf{l1} = \mathsf{l2}$ and $\mathsf{current1} = \mathsf{current2}$. Using the specification of is_female, we obtain $\mathsf{t1} = \mathsf{t2}$, which implies that the loop invariant is preserved. The loop invariant implies the postcondition.

3 Preliminaries

We model our setting according to the relational logic by Banerjee, Naumann and Nikouei [5][2] and, like them, use a standard Hoare logic [4] to reason about single program executions. Figure 3 shows the language we use to define modular product programs. x ranges over the set of local integer variable names Var. Note that this language is deterministic; non-determinism can for example be modelled via additional inputs, as is often done for modelling fairness in concurrent programs [16]. Program configurations have the form $\langle s, \sigma \rangle$, where $\sigma \in \Sigma$ maps variable names to values. The value of expression e in state σ is

[2] Our handling of procedure calls is slightly different, but amounts to restricting procedures to work only on local variables not used in the rest of the program (as opposed to having a global state on which all procedures work directly), and only interacting with the rest of the program via explicitly declared return parameters.

$$
\begin{array}{ll}
(\mathit{Programs}) & \mathit{Prog} ::= \texttt{procedure}\ \mathit{main}(\overline{x})\ \texttt{returns}\ (\overline{y})\{s\} :: \mathit{Nil} \mid \mathit{Proc} :: \mathit{Prog} \\
(\mathit{Procedures}) & \mathit{Proc} ::= \texttt{procedure}\ m(\overline{x})\ \texttt{returns}\ (\overline{y})\{s\} \\
(\mathit{Statements}) & s ::= x{:=}e \mid s; s \mid \texttt{if}\ (e)\ \texttt{then}\ \{s\}\ \texttt{else}\ \{s\} \mid \texttt{while}\ (e)\ \texttt{do}\ \{s\} \\
 & \quad \mid \overline{x}{:=}\ \texttt{call}\ m(\overline{e}) \\
(\mathit{Expressions}) & e ::= c \mid x \mid e \oplus e \text{ where } c \in \mathbb{Z} \text{ and } \oplus \in \{+, -, \times, \ldots\} \\
(\mathit{Assertions}) & P ::= P \wedge P \mid P \Rightarrow P \mid \forall x.\, P \mid e \\
(\mathit{RelExpressions}) & \hat{e} ::= c \mid \overset{i}{x} \mid \hat{e} \oplus \hat{e} \\
(\mathit{RelAssertions}) & \hat{P} ::= \hat{P} \wedge \hat{P} \mid \hat{P} \Rightarrow \hat{P} \mid \forall \overset{1}{x}, \ldots, \overset{k}{x}.\, \hat{P} \mid \hat{e} \\
(\mathit{MixAssertions}) & \check{P} ::= P \mid \hat{P} \mid \check{P} \wedge \check{P}
\end{array}
$$

Fig. 3. Language.

denoted as $\sigma(e)$. The small-step transition relation for program configurations has the form $\langle s, \sigma\rangle \to \langle s', \sigma'\rangle$. A hypothesis context Φ maps procedure names to specifications.

The judgment $\Phi \vDash s : P \rightsquigarrow Q$ denotes that statement s, when executed in a state fulfilling the unary assertion P, will not fault, and if the execution terminates, the resulting state will fulfill the unary assertion Q. For an extensive discussion of the language and its operational and axiomatic semantics, see [5].

In addition to standard unary expressions and assertions, we define relational expressions and assertions. They differ from normal expressions and assertions in that they contain parameterized variable references of the form $\overset{i}{x}$ and are evaluated over a tuple of states instead of a single one. A relational expression is k-relational if for all contained variable references $\overset{i}{x}$, $1 \leq i \leq k$, and analogous for relational assertions. The value of a variable reference $\overset{i}{x}$ with $1 \leq i \leq k$ in a tuple of states $(\sigma_1, \ldots, \sigma_k)$ is $\sigma_i(x)$; the evaluation of arbitrary relational expressions and the validity of relational assertions $(\sigma_1, \ldots, \sigma_k) \vDash \hat{P}$ are defined accordingly.

Definition 1. *A k-relational specification $s : \hat{P} \twoheadrightarrow_k \hat{Q}$ holds iff $\hat{P}$ and $\hat{Q}$ are k-relational assertions, and for all $\sigma_1, \ldots, \sigma_k, \sigma'_1, \ldots, \sigma'_k$, if $(\sigma_1, \ldots, \sigma_k) \vDash \hat{P}$ and $\forall i \in \{1, \ldots, k\}.\, \langle s, \sigma_i\rangle \to^* \langle \texttt{skip}, \sigma'_i\rangle$, then $(\sigma'_1, \ldots, \sigma'_k) \vDash \hat{Q}$.*

We write $s : \hat{P} \twoheadrightarrow \hat{Q}$ for the most common case $s : \hat{P} \twoheadrightarrow_2 \hat{Q}$.

4 Modular k-Product Programs

In this section, we define the construction of modular products for arbitrary k. We will subsequently define the transformation of both relational and unary specifications to modular products.

4.1 Product Construction

Assume as given a function $(\textsc{Var}, \mathbb{N}) \to \textsc{Var}$ that renames variables for different executions. We write $e^{(i)}$ for the renaming of expression e for execution i and

require that $\forall x, y, i, j.\, i \neq j \Rightarrow x^{(i)} \neq y^{(j)}$. We write $fresh(x_1, x_2, \ldots)$ to denote that the variable names $x_1, x_2, \ldots$ are fresh names that do not occur in the program and have not yet been used during the transformation. $\mathring{e}$ is used to abbreviate $e^{(1)}, \ldots, e^{(k)}$.

We denote the modular k-product of a statement s that is parameterized by the activation variables $p^{(1)}, \ldots, p^{(k)}$ as $[\![s]\!]_k^{\mathring{p}}$. The product construction for procedures is defined as

$$
\begin{aligned}
& [\![\textbf{procedure}\ m(x_1, \ldots, x_m)\ \textbf{returns}\ (y_1, \ldots, y_n)\{s\}]\!]_k \\
= \ & \textbf{procedure}\ m(p^{(1)}, \ldots, p^{(k)}, args)\ \textbf{returns}\ (rets)\{[\![s]\!]_k^{\mathring{p}}\} \\
& \text{where} \\
& args = {x_1}^{(1)}, \ldots, {x_1}^{(k)}, \ldots, {x_m}^{(1)}, \ldots, {x_m}^{(k)} \\
& rets = {y_1}^{(1)}, \ldots, {y_1}^{(k)}, \ldots, {y_n}^{(1)}, \ldots, {y_n}^{(k)}
\end{aligned}
$$

Figure 4 shows the product construction rules for statements, which generalize the transformation explained in Sect. 2. We write `if` (e) `then` $\{s\}$ as a shorthand for `if` (e) `then` $\{s\}$ `else` $\{$`skip`$\}$, and $\bigodot_{i=1}^{k} s_i$ for the sequential composition of k statements $s_1; \ldots; s_k$.

The core principle behind our encoding is that statements that directly change the state are duplicated for each execution and made conditional under the respective activation variables, whereas control statements are not duplicated and instead manipulate the activation variables to pass activation information to their sub-statements. This enables us to assert or assume relational assertions before and after any statement from the original program. The only state-changing statements in our language, variable assignments, are therefore transformed to a sequence of conditional assignments, one for each execution. Each assignment is executed only if the respective execution is currently active.

Duplicating conditionals would also duplicate the calls and loops in their branches. To avoid that, modular products eliminate top-level conditionals; instead, new activation variables are created and assigned the values of the current activation variables conjoined with the guard for each branch. The branches are then sequentially executed based on their respective activation variables.

A while loop is transformed to a single while loop in the product program that iterates as long as the loop guard is true for *any* active execution. Inside the loop, fresh activation variables indicate whether an execution reaches the loop *and* its loop condition is true. The loop body will then modify the state of an execution only if its activation variable is true. The resulting construct affects the program state in the same way as a self-composition of the original loop would, but the fact that our product contains only a single loop enables us to use relational loop invariants instead of full functional specifications.

For procedure calls, it is crucial that the product contains a single call for every call in the original program, in order to be able to apply relational specifications at the call site. As explained before, initial activation parameters are added to every procedure declaration, and all parameters are duplicated k times.

$$
\begin{aligned}
[\![s_1; s_2]\!]_k^{\mathring{p}} &= [\![s_1]\!]_k^{\mathring{p}}; [\![s_2]\!]_k^{\mathring{p}} \\
[\![\texttt{skip}]\!]_k^{\mathring{p}} &= \texttt{skip} \\
[\![x{:=}e]\!]_k^{\mathring{p}} &= \bigodot_{i=1}^{k} \texttt{if } (p^{(i)}) \texttt{ then } \{x^{(i)}{:=}e^{(i)}\} \\
[\![\texttt{if } (e) \texttt{ then } \{s_1\} \texttt{ else } \{s_2\}]\!]_k^{\mathring{p}} &= \bigodot_{i=1}^{k} ({p_1}^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \\
&\quad \bigodot_{i=1}^{k} ({p_2}^{(i)}{:=}p^{(i)} \wedge \neg e^{(i)}); \\
&\quad [\![s_1]\!]_k^{\mathring{p_1}}; [\![s_2]\!]_k^{\mathring{p_2}} \\
&\quad \text{where} \\
&\quad fresh(\mathring{p_1}) \wedge fresh(\mathring{p_2}) \\
[\![\texttt{while } (e) \texttt{ do } \{s\}]\!]_k^{\mathring{p}} &= \texttt{while } (\textstyle\bigvee_{i=1}^{k} (p^{(i)} \wedge e^{(i)})) \texttt{ do } \{ \\
&\qquad \bigodot_{i=1}^{k} ({p_1}^{(i)}{:=}p^{(i)} \wedge e^{(i)}); \\
&\qquad [\![s]\!]_k^{\mathring{p_1}} \\
&\quad \} \\
&\quad \text{where} \\
&\quad fresh(\mathring{p_1}) \\
[\![x_1, \mathring{\ldots}, x_n{:=} \texttt{ call } m(e_1, \ldots, e_m)]\!]_k^{\mathring{p}} &= \texttt{if } (\textstyle\bigvee_{i=1}^{k} p^{(i)}) \texttt{ then } \{ \\
&\qquad \bigodot_{i=1}^{k} \texttt{if } (p^{(i)}) \texttt{ then } \{\bigodot_{j=1}^{m} ({a_j}^{(i)}{:=}{e_j}^{(i)})\}; \\
&\qquad ts{:=} \texttt{ call } m(p^{(1)}, \ldots, p^{(k)}, as); \\
&\qquad \bigodot_{i=1}^{k} \texttt{if } (p^{(i)}) \texttt{ then } \{\bigodot_{j=1}^{n} (x_j{}^{(i)}{:=}{t_j}^{(i)})\} \\
&\quad \} \\
&\quad \text{where} \\
&\quad fresh(\mathring{a_1}, \ldots, \mathring{a_m}) \wedge fresh(\mathring{t_1}, \ldots, \mathring{t_n}) \\
&\quad as = [{a_1}^{(1)}, \ldots, {a_1}^{(k)}, \ldots, {a_m}^{(1)}, \ldots, {a_m}^{(k)}] \\
&\quad ts = [{t_1}^{(1)}, \ldots, {t_1}^{(k)}, \ldots, {t_n}^{(1)}, \ldots, {t_n}^{(k)}]
\end{aligned}
$$

Fig. 4. Construction rules for statement products.

Procedure calls are therefore transformed such that the values of the current activation variables are passed, and all arguments are passed once for each execution. The return values are stored in temporary variables and subsequently assigned to the actual target variables only for those executions that actually execute the call, so that for all other executions, the target variables are not affected.

The transformation wraps the call in a conditional so that the call is performed only if at least one execution is active. This prevents the transformation from introducing infinite recursion that is not present in the original program.

Note that for an inactive execution i, arbitrary argument values are passed in procedure calls, since the passed variables ${a_j}^{(i)}$ are not initialized. This is unproblematic because these values will not be used by the procedure. It is important to not evaluate ${e_j}^{(i)}$ for inactive executions, since this could lead to false alarms for languages where expression evaluation can fail.

4.2 Transformation of Assertions

We now define how to transform unary and relational assertions for use in a modular product.

Unary assertions such as ordinary procedure preconditions describe state properties that should hold for every single execution. When checking or assuming that a unary assertion holds at a specific point in the program, we need to take into account that it only makes sense to do so for executions that actually reach that program point. We can express this by making the assertion conditional on the activation variable of the respective execution; as a result, any unary assertion is trivially valid for all inactive executions.

A k-relational assertion, on the other hand, describes the relation between the states of all k executions. Checking or assuming a relational assertion at some point is meaningful only if *all* executions actually reach that point. This can be expressed by making relational assertions conditional on the conjunction of all current activation variables. If at least one execution does not reach the assertion, it holds trivially.

We formalize this idea by defining a function α that maps relational assertions $\hat{P}$ to unary assertions P of the product program such that $\alpha(\hat{P}) = \hat{P}[Var^{(1)}/\overset{1}{Var}]\dots[Var^{(k)}/\overset{k}{Var}]$. Assertions can then be transformed for use in a k-product as follows:

- The transformation $\lfloor \hat{P} \rfloor_k^{\mathring{p}}$ of a k-relational assertion $\hat{P}$ with the activation variables $p^{(1)}, \dots, p^{(k)}$ is $(\bigwedge_{i=1}^{k} p^{(i)}) \Rightarrow \alpha(\hat{P})$.
- The transformation $\lfloor P \rfloor_k^{\mathring{p}}$ of a unary assertion P is $\bigwedge_{i=1}^{k} (p^{(i)} \Rightarrow P^{(i)})$.

Importantly, our approach allows using *mixed* assertions and specifications, which represent conjunctions of unary and relational assertions. For example, it is common to combine a unary precondition that ensures that a procedure will not raise an error with a relational postcondition that states that it is deterministic.

A mixed assertion $\check{R}$ of the form $P \wedge \hat{Q}$ means that the unary assertion P holds for every single execution, and if all executions are currently active, the relational assertion $\hat{Q}$ holds as well. The transformation of mixed assertions is straightforward: $\lfloor \check{R} \rfloor_k^{\mathring{p}} = \lfloor P \rfloor_k^{\mathring{p}} \wedge \lfloor \hat{Q} \rfloor_k^{\mathring{p}}$.

4.3 Heap-Manipulating Programs

The approach outlined so far can easily be extended to programs that work on a mutable heap, assuming that object references are opaque, i.e., they cannot be inspected or used in arithmetic. In order to create a distinct state space for each execution represented in the product, allocation statements are duplicated and made conditional like assignments, and therefore create a different object for each active execution. The renaming of a field dereference $e.f$ is then defined as $e^{(i)}.f$. As a result, the heap of a k-product will consist of k partitions that do not contain references to each other, and execution i will only ever interact with objects from its partition of the heap.

The verification of modular products of heap-manipulating programs does not depend on any specific way of achieving framing. Our implementation is based on implicit dynamic frames [25], but other approaches are feasible as well, provided that procedures can be specified in such a way that the caller knows the heap stays unmodified for all executions whose activation variables are false.

Since the handling of the heap is largely orthogonal to our main technique, we will not go into further detail here, but we do support heap-manipulating programs in our implementation.

5 Soundness and Completeness

A product construction is sound if an execution of a k-product mirrors k separate executions of the original program such that properties proved about the product entail hyperproperties of the original program. In this section, we sketch a soundness proof of our k-product construction in the presence of only unary procedure specifications. We also sketch a proof for relational specifications for the case $k = 2$, making use of the relational logic presented by Banerjee et al. [5]. Finally, we informally discuss the completeness of modular products.

5.1 Soundness with Unary Specifications

A modular k-product must soundly encode k executions of the original program. That is, if an encoded unary specification holds for a product program then the original specification holds for the original program.

We define a relation $\sigma \simeq_i \sigma'$ that denotes that σ contains a renamed version of all variables in σ', i.e., $\forall v \in dom(\sigma') : \sigma(v^{(i)}) = \sigma'(v)$. Without the index i, $\simeq$ denotes the same but without renaming, and is used to express equality modulo newly introduced activation variables.

Theorem 1. *Assume that for all procedures m in a hypothesis context Φ we have that $m : S \rightsquigarrow T \in dom(\Phi)$ if and only if $m : \lfloor S \rfloor_k^{\mathring{p}} \rightsquigarrow \lfloor T \rfloor_k^{\mathring{p}} \in dom(\Phi')$. Then $\Phi' \vDash [\![s]\!]_k^{\mathring{p}} : \lfloor P \rfloor_k^{\mathring{p}} \rightsquigarrow \lfloor Q \rfloor_k^{\mathring{p}}$ implies that $\Phi \vDash s : P \rightsquigarrow Q$.*

Proof (Sketch). We sketch a proof based on the operational semantics of our language. We show that the execution of the product program with exactly one active execution corresponds to a single execution of the original program.

Assume that $\Phi' \vDash [\![s]\!]_k^{\mathring{p}} : \lfloor P \rfloor_k^{\mathring{p}} \rightsquigarrow \lfloor Q \rfloor_k^{\mathring{p}}$, and that $\sigma \vDash \lfloor P \rfloor_k^{\mathring{p}}$. If $[\![s]\!]_k^{\mathring{p}}$ does not diverge when executed from σ we have that $\langle [\![s]\!]_k^{\mathring{p}}, \sigma \rangle \rightarrow^* \langle \texttt{skip}, \sigma' \rangle$ and $\sigma' \vDash \lfloor Q \rfloor_k^{\mathring{p}}$. We now prove that a run of the product with all but one execution being inactive reflects the states that occur in a run of the original program. Assume that $\sigma \vDash p^{(1)} \wedge \bigwedge_{i=2}^{k} (\neg p^{(i)})$ and $\langle s, \sigma_1 \rangle \rightarrow^* \langle \texttt{skip}, \sigma_1' \rangle$ and initially $\sigma \simeq_1 \sigma_1$, which implies $\sigma_1 \vDash P$. We prove by induction on the derivation of $\langle s, \sigma_1 \rangle \rightarrow^* \langle \texttt{skip}, \sigma_1' \rangle$ that $\langle [\![s]\!]_k^{\mathring{p}}, \sigma \rangle \rightarrow^* \langle \texttt{skip}, \sigma' \rangle$ and $\sigma' \simeq_1 \sigma_1'$, meaning that the product execution terminates, and subsequently by induction on the derivation of $\langle [\![s]\!]_k^{\mathring{p}}, \sigma \rangle \rightarrow^* \langle \texttt{skip}, \sigma' \rangle$ that $\sigma' \simeq_1 \sigma_1'$, from which we can derive that $\sigma_1' \vDash Q$. □

5.2 Soundness for Relational Specifications

The main advantage of modular product programs over other kinds of product programs is that it allows reasoning about procedure calls in terms of relational specifications. We therefore need to show the soundness of our approach in the presence of procedures with such specifications. In particular, we must establish that if a transformed relational specification holds for a modular product then the original relational specification will hold for a set of k executions of the original program.

Our proof sketch is phrased in terms of *biprograms* as introduced by Banerjee et al. [5]. Biprogram executions correspond to two partly aligned executions of their two underlying programs. A biprogram ss can have the form $(s_1|s_2)$ or $\|s\|$; the former represents the two executions of s_1 and s_2, whereas the latter represents an aligned execution of s by both executions, which enables using relational specifications for procedure calls[3]. We denote the small-step transition relation between biprogram configurations as $\langle ss, \sigma_1|\sigma_2\rangle \Rrightarrow^* \langle ss', \sigma_1'|\sigma_2'\rangle$. We make use of a relation $\sigma \approxeq \sigma_1|\sigma_2$ that denotes that σ contains renamed versions of all variables in both σ_1 and σ_2 with the same values.

Biprograms do not allow mixed procedure specifications, meaning that a procedure can either have only a unary specification, or it can have only a relational specification, in which case it can only be invoked by both executions simultaneously. As mentioned before, our approach does not have this limitation, but we can artificially enforce it for the purposes of the soundness proof.

We can now state our theorem. Since biprograms represent the execution of two programs, we formulate soundness for $k = 2$ here.

Theorem 2. *Assume that hypothesis context Φ maps procedure names to relational specifications if all calls to the procedure in s can be aligned from any pair of states satisfying $\hat{P}$, and to unary specifications otherwise. Assume further that hypothesis context Φ' maps the same procedure names to their transformed specifications. Finally, assume that $\Phi' \vdash [\![s]\!]_2^{\mathring{p}} : \lfloor \hat{P} \rfloor_2^{\mathring{p}} \rightsquigarrow \lfloor \hat{Q} \rfloor_2^{\mathring{p}}$ and $(\sigma_1, \sigma_2) \vDash \hat{P}$. If $\langle s, \sigma_1\rangle \to^* \langle \texttt{skip}, \sigma_1'\rangle$ and $\langle s, \sigma_2\rangle \to^* \langle \texttt{skip}, \sigma_2'\rangle$, then $(\sigma_1', \sigma_2') \vDash \hat{Q}$.*

Proof (Sketch). The proof follows the same basic outline as the one for Theorem 1 but reasons about the operational semantics of biprograms representing two executions of s.

Assume that $\Phi' \vdash [\![s]\!]_2^{\mathring{p}} : \lfloor \hat{P} \rfloor_2^{\mathring{p}} \rightsquigarrow \lfloor \hat{Q} \rfloor_2^{\mathring{p}}$ and $\sigma \vDash \lfloor \hat{P} \rfloor_2^{\mathring{p}}$. If $[\![s]\!]_2^{\mathring{p}}$ does not diverge when executed from σ we get that $\langle [\![s]\!]_2^{\mathring{p}}, \sigma\rangle \to^* \langle \texttt{skip}, \sigma'\rangle$ and $\sigma' \vDash \lfloor \hat{Q} \rfloor_2^{\mathring{p}}$. Assume that initially $\sigma \approxeq \sigma_1|\sigma_2$, which implies that $(\sigma_1, \sigma_2) \vDash \hat{P}$. We prove by induction on the derivation of $\langle [\![s]\!]_2^{\mathring{p}}, \sigma\rangle \to^* \langle \texttt{skip}, \sigma'\rangle$ that (1) if $\sigma \vDash p^{(1)} \wedge p^{(2)}$, then there exists ss that represents two executions of s s.t. $\langle ss, \sigma_1|\sigma_2\rangle \Rrightarrow^* \langle \|\texttt{skip}\|, \sigma_1'|\sigma_2'\rangle$ and $\sigma' \approxeq \sigma_1'|\sigma_2'$; (2) if $\sigma \vDash p^{(1)} \wedge \neg p^{(2)}$, then $\langle s, \sigma_1\rangle \to^* \langle \texttt{skip}, \sigma_1'\rangle$ and $\sigma' \approxeq \sigma_1'|\sigma_2$; (3) if $\sigma \vDash \neg p^{(1)} \wedge p^{(2)}$, then $\langle s, \sigma_2\rangle \to^* \langle \texttt{skip}, \sigma_2'\rangle$ and $\sigma' \approxeq \sigma_1|\sigma_2'$; (4) if $\sigma \vDash \neg p^{(1)} \wedge \neg p^{(2)}$, then $\sigma \simeq \sigma'$. From the first point and semantic consistency

[3] We modified the original notation to avoid clashes with our own concepts introduced earlier.

of the relational logic, we can conclude that $(\sigma_1', \sigma_2') \vDash \hat{Q}$. Finally, we prove that $\langle [\![s]\!]_2^{\mathring{p}}, \sigma\rangle \to^* \langle \texttt{skip}, \sigma'\rangle$ by showing that non-termination of the product implies the non-termination of at least one of the two original program runs. If the condition of a loop in the product remains true forever, the loop condition of at least one encoded execution must be true after every iteration. We show that (1) this is not due to an interaction of multiple executions, since the condition for every execution will remain false if it becomes false once, and (2) since the encoded states of active executions progress as they do in the original program, the condition of a single execution in the product remains true forever only if it does in the original program. A similar argument shows that the product cannot diverge because of infinite recursive calls. □

5.3 Completeness

We believe modular product programs to be complete, meaning that any hyperproperty of multiple executions of a program can be proved about its modular product program. Since the product faithfully models the executions of the original program, the completeness of modular products is potentially limited only by the underlying verification logic and the assertion language, but not by the product construction itself.

6 Modular Verification of Secure Information Flow

In this section, we demonstrate the expressiveness of modular product programs by showing how they can be used to verify an important hyperproperty, information flow security. We first concentrate on secure information flow in the classical sense [9], and later demonstrate how the ability to check relational assertions at any point in the program can be exploited to prove advanced properties like the absence of timing and termination channels, and to encode declassification.

6.1 Non-interference

Secure information flow, i.e., the property that secret information is not leaked to the public outputs of a program, can be expressed as a relational 2-safety property of a program called *non-interference*. Non-interference states that, if a program is run twice, with the public (often called *low*) inputs being equal in both runs but the secret (or *high*) inputs possibly being different, the public outputs of the program must be equal in both runs [8]. This property guarantees that the high inputs do not influence the low outputs.

We can formalize non-interference as follows:

Definition 2. *A statement s that operates on a set of variables $X = \{x_1, \ldots, x_n\}$, of which some subset $X_l \subseteq X$ is low, satisfies non-interference iff for all σ_1, σ_2 and σ_1', σ_2', if $\forall x \in X_l.\, \sigma_1(x) = \sigma_2(x)$ and $\langle s, \sigma_1\rangle \to^* \langle \texttt{skip}, \sigma_1'\rangle$ and $\langle s, \sigma_2\rangle \to^* \langle \texttt{skip}, \sigma_2'\rangle$ then $\forall x \in X_l.\sigma_1'(x) = \sigma_2'(x)$.*

Since our definition of non-interference describes a hyperproperty, we can verify it using modular product programs:

Theorem 3. *A statement s that operates on a set of variables $X = \{x_1, \ldots, x_n\}$, of which some subset $X_l \subseteq X$ is low, satisfies non-interference under a unary precondition P if $\Phi \vdash [\![s]\!]_2^{\mathring{p}} : \lfloor P \rfloor_2^{\mathring{p}} \wedge (\forall x \in X_l.\, x^{(1)} = x^{(2)}) \rightsquigarrow \forall x \in X_l.\, x^{(1)} = x^{(2)}$*

Proof (Sketch). Since non-interference can be expressed using a 2-relational specification, the theorem follows directly from Theorem 2. □

For non-deterministic programs whose behavior can be modelled by adding input parameters representing the non-deterministic choices, those parameters can be considered low if the choice is not influenced in any way by secret data.

An expanded notion of secure information flow considers observable *events* in addition to regular program outputs [17]. An event is a statement that has an effect that is visible to an outside observer, but may not necessarily affect the program state. The most important examples of events are output operations like printing a string to the console or sending a message over a network. Programs that cause events can be considered information flow secure only if the sequence of produced events is not influenced by high data. One way to verify this using our approach is to track the sequence of produced events in a ghost variable and verify that its value never depends on high data. This approach requires substantial amounts of additional specifications.

Modular product programs offer an alternative approach for preventing leaks via events, since they allow formulating assertions about the relation between the activation variables of different executions. In particular, if a given event has the precondition that all activation variables are equal when the event statement is reached then this event will either be executed by both executions or be skipped by both executions. As a result, the sequence of events produced by a program will be equal in all executions.

6.2 Information Flow Specifications

The relational specifications required for modularly proving non-interference with the previously described approach have a specific pattern: they can contain functional specifications meant to be valid for both executions (e.g., to make sure both executions run without errors), they may require that some information is low, which is equivalent to the two renamings of the same expression being equal, and, in addition, they may assert that the control flow at a specific program point is low.

We therefore introduce modular *information flow specifications*, which can express all properties required for proving secure information flow but are transparent w.r.t. the encoding or the verification methodology, i.e., they allow expressing that a given operation or value must not be secret without knowledge of the encoding of this fact into an assertion about two different program executions. We define information flow specifications as follows:

$$(SIFAssertions)\ \tilde{P} ::= \tilde{P} \wedge \tilde{P} \mid e \mid low(e) \mid lowEvent \mid \tilde{P} \Rightarrow \tilde{P} \mid \forall x.\, \tilde{P}$$

low(e) and *lowEvent* may be used on the left side of an implication only if the right side has the same form. *low*(e) specifies that the value of the expression e is not influenced by high data. Note that e can be any expression and is not limited to variable references; this reflects the fact that our approach can label secrecy in a more fine-grained way than, e.g., a type system. One can, for example, declare to be public whether a number is odd while keeping its value secret.

$$\begin{aligned}
\lceil e \rceil^{\mathring{p}} &= (p^{(1)} \Rightarrow e^{(1)}) \wedge (p^{(2)} \Rightarrow e^{(2)}) \\
\lceil low(e) \rceil^{\mathring{p}} &= (p^{(1)} \wedge p^{(2)} \Rightarrow e^{(1)} = e^{(2)}) \\
\lceil lowEvent \rceil^{\mathring{p}} &= p^{(1)} = p^{(2)} \\
\lceil \tilde{P}_1 \wedge \tilde{P}_2 \rceil^{\mathring{p}} &= \lceil \tilde{P}_1 \rceil^{\mathring{p}} \wedge \lceil \tilde{P}_2 \rceil^{\mathring{p}} \\
\lceil \tilde{P}_1 \Rightarrow \tilde{P}_2 \rceil^{\mathring{p}} &= \lceil \tilde{P}_1 \rceil^{\mathring{p}} \Rightarrow \lceil \tilde{P}_2 \rceil^{\mathring{p}} \\
\lceil \forall x.\, \tilde{P} \rceil^{\mathring{p}} &= \forall x^{(1)}, x^{(2)}.\, x^{(1)} = x^{(2)} \Rightarrow \lceil \tilde{P} \rceil^{\mathring{p}}
\end{aligned}$$

Fig. 5. Translation of information flow specifications.

lowEvent specifies that high data must not influence if and how often the current program point is reached by an execution, which is a sufficient precondition of any statement that causes an observable event. In particular, if a procedure outputs an expression e, the precondition *lowEvent* $\wedge$ *low*(e) guarantees that no high information will be leaked via this procedure.

Information flow specifications can express complex properties. $e_1 \Rightarrow low(e_2)$, for example, expresses that if e_1 is true, e_2 must not depend on high data; $e_1 \Rightarrow$ *lowEvent* says the same about the current control flow. A possible use case for these assertions is the precondition of a library function that prints e_2 to a low-observable channel if e_1 is true, and to a secure channel otherwise.

The encoding $\lceil \tilde{P} \rceil^{\mathring{p}}$ of an information flow assertion $\tilde{P}$ under the activation variables $p^{(1)}$ and $p^{(2)}$ is defined in Fig. 5. Note that high-ness of some expression is not modelled by its renamings being definitely unequal, but by leaving underspecified whether they are equal or not, meaning that high-ness is simply the absence of the knowledge of low-ness. As a result, it is never necessary to specify explicitly that an expression is high. This approach (which is also used in self-composition) is analogous to the way type systems encode security levels, where low is typically a subtype of high. For the example in Fig. 1, a possible, very precise information flow specification could say that the results of main are low if the first bit of all entries in people is low. We can write this as main : $low(|\mathsf{people}|) \wedge \forall i \in \{0, \ldots, |\mathsf{people}| - 1\}.\, low(\mathsf{people[i]}\ \mathbf{mod}\ 2) \rightsquigarrow low(\mathsf{count})$. In the product, this will be translated to main : $\mathsf{p1} \wedge \mathsf{p2} \Rightarrow |\mathsf{people1}| = |\mathsf{people2}| \wedge \forall i \in \{0, \ldots, |\mathsf{people1}| - 1\}.\, (\mathsf{people1[i]}\ \mathbf{mod}\ 2) = (\mathsf{people2[i]}\ \mathbf{mod}\ 2) \twoheadrightarrow \mathsf{count1} = \mathsf{count2}$.

In this scenario, the loop in main could have the simple invariant $low(\mathsf{i}) \wedge low(\mathsf{count})$, and the procedure is_female could have the contract is_female : $true \rightsquigarrow (low(\mathsf{person}\ \mathbf{mod}\ 2) \Rightarrow low(\mathsf{res}))$. This contract follows a useful pattern

```
procedure check(password, input)
          returns (result)
{
  result := |password| == |input|;
  i := 0;
  while (i < min(|password|, |input|)) {
     result := result && password[i] == input[i];
     i := i + 1;
  }
}
```

Fig. 6. Password check example: leaking secret data is desired.

where, instead of requiring an input to be low and promising that an output will be low for all calls, the output is decribed as *conditionally* low based on the level of the input, which is more permissive for callers.

The example shows that the information relevant for proving secure information flow can be expressed concisely, without requiring any knowledge about the methodology used for verification. Modular product programs therefore enable the verification of the information flow security of main based solely on modular, relational specifications, and without depending on functional specifications.

6.3 Secure Information Flow with Arbitrary Security Lattices

The definition of secure information flow used in Definition 2 is a special case in which there are exactly two possible classifications of data, high and low. In the more general case, classifications come from an arbitrary lattice $\langle \mathcal{L}, \sqsubseteq \rangle$ of security levels s.t. for some $l_1, l_2 \in \mathcal{L}$, information from an input with level l_1 may influence an output with level l_2 only if $l_1 \sqsubseteq l_2$. Instead of the specification $low(e)$, information flow assertions can therefore have the form $levelBelow(e, l)$, meaning that the security level of expression e is at most l.

It is well-known that techniques for verifying information flow security with two levels can conceptually be used to verify programs with arbitrary finite security lattices [23] by splitting the verification task into $|\mathcal{L}|$ different verification tasks, one for each element of $\mathcal{L}$. Instead, we propose to combine all these verification tasks into a single task by using a symbolic value for l, i.e., declaring an unconstrained global constant representing l. Specifications can then be translated as follows:

$$levelBelow(e, l') \triangleq\ l' \sqsubseteq l \Rightarrow e^{(1)} = e^{(2)}$$

Since no information about l is known, verification will only succeed if all assertions can be proven for all possible values of l, which is equivalent to proving them separately for each possible value of l.

6.4 Declassification

In practice, non-interference is too strong a property for many use cases. Often, some leakage of secret data is required for a program to work correctly. Consider

```
procedure main(h: Int)
{
  while (h != 0) {
    h := h - 1;
  }
}
```

```
procedure main(h: Int)
{
  i := 0;
  while (i < h) {
    i := i + 1
  }
  print(0)
}
```

Fig. 7. Programs with a termination channel (left), and a timing channel (right). In both cases, h is high.

the case of a password check (see Fig. 6): A secret internal password is compared to a non-secret user input. While the password itself must not be leaked, the information whether the user input matches the password should influence the public outcome of the program, which is forbidden by non-interference.

To incorporate this intention, the relevant part of the secret information can be *declassified* [24], e.g., via a declassification statement `declassify` e that declares an arbitrary expression e to be low. With modular products, declassification can be encoded via a simple assumption stating that, if the declassification is executed in both executions, the expression is equal in both executions:

$$[\![\texttt{declassify}\ e]\!]_2^{\mathring{p}} = \texttt{assume}\ (p^{(1)} \wedge p^{(2)}) \Rightarrow e^{(1)} = e^{(2)}$$

Introducing an assumption of this form is sound if the information flow specifications from Sect. 6.2 are used to specify the program. Since high-ness is encoded as the absence of the knowledge that an expression is equal in both executions, not by the knowledge that they are different, there is no danger that assuming equality will contradict current knowledge and thereby cause unsoundness. As in the information flow specifications, the declassified expression can be arbitrarily complex, so that it is for example possible to declassify the sign of an integer while keeping all other information about it secret.

The example in Fig. 6 becomes valid if we add `declassify result` at the end of the procedure, or if we declassify a more complex expression by adding `declassify` *equal* (password, input) at some earlier point. The latter would arguably be safer because it specifies exactly the information that is intended to be leaked, and would therefore prevent accidentally leaking more if the implementation of the checking loop was faulty.

This kind of declassification has the following interesting properties: First, it is *imperative*, meaning that the declassified information may be leaked (e.g., via a **print** statement) after the execution of the declassification statement, but not before. Second, it is *semantic*, meaning that the declassification affects the value of the declassified expression as opposed to, e.g., syntactically the declassified variable. As a result, it will be allowed to leak any expression whose value contains the same (or a part of the) secret information which was declassified, e.g., the expression $f(e)$ if f is a deterministic function and e has been declassified.

6.5 Preventing Termination Channels

In Definition 2, we have considered only terminating program executions. In practice, however, termination is a possible side-channel that can leak secret information to an outside observer. Figure 7 (left) shows an example of a program that verifies under the methodology presented so far, but leaks information about the secret input h to an observer: If h is initially negative, the program will enter an endless loop. Anyone who can observe the termination behavior of the program can therefore conclude if h was negative or not.

To prevent leaking information via a termination side channel, it is necessary to verify that the termination of a program depends only on public data. We will show that modular product programs are expressive enough to encode and check this property. We will focus on preventing non-termination caused by infinite loops here; preventing infinite recursion works analogously. In particular, we want to prove that if a loop iterates forever in one execution, any other execution with the same low inputs will also reach this loop and iterate forever. More precisely, this means that

(A) if a loop does not terminate, then whether or not an execution reaches that loop must not depend on high data.
(B) whether a loop that is reached by both executions terminates must not depend on high data.

We propose to verify these properties by requiring additional specifications that state, for every loop, an exact condition under which it terminates. This condition may neither over- nor underapproximate the termination behavior; the loop must terminate if and only if the condition is true. For Fig. 7 (left) the condition is $h \geq 0$. We also require a ranking function for the cases when the termination condition is true. We can then prove the following:

(a) If the termination condition of a loop evaluates to false, then any two executions with identical low inputs either both reach the loop or both do not reach the loop (i.e., reaching the loop is a low event). This guarantees property (A) above.
(b) For loops executed by both executions, the loop's termination condition is low. This guarantees property (B) under the assumption that the termination condition is exact.
(c) The termination condition is sound, i.e., every loop terminates if its termination condition is true. We prove this by showing that if the termination condition is true, we can prove the termination of the loop using the supplied ranking function.
(d) The termination condition is complete, i.e., every loop terminates only if its termination condition is true. We prove this by showing that if the condition is false, the loop condition will always remain true. This check, along with the previous proof obligation, ensures that the termination condition is exact.
(e) Every statement in a loop body terminates if the loop's termination condition is true, i.e., the loop's termination condition implies the termination conditions of all statements in its body.

```
term(w, c) = cond:=e_c;
             assert ¬e_c ⇒ lowEvent;          // checks (a)
             assert low(e_c);                 // checks (b)
             assert e_c ⇒ e_r ≥ 0;            // checks (c)
             assert c ⇒ e_c;                  // checks (e)
             while (e)
              invariant ¬cond ⇒ e             // checks (d)
              do {
                 if (cond) then {rank:=e_r};
                 term(s, cond);
                 if (cond) then {             // checks (c)
                     assert 0 ≤ e_r ∧ e_r < rank
                 }
             }
```

Fig. 8. Program instrumentation for termination leak prevention. We abbreviate `while` (e) `terminates`(e_c, e_r) `do` $\{s\}$ as w.

We introduce an annotated while loop `while` (e) `terminates`(e_c, e_r) `do` $\{s\}$, where e_c is the exact termination condition and e_r is the ranking function, i.e., an integer expression whose value decreases with every loop iteration but never becomes negative if the termination condition is true. Based on these annotations, we present a program instrumentation *term* (s, c) that inserts the checks outlined above for every while loop in s. c is the termination condition of the outside scope, i.e., for the instrumentation of a nested loop, it is the termination condition e_c of the outer loop. The instrumentation is defined for annotated while loops in Fig. 8; for all other statements, it does not make any changes except instrumenting all substatements. The instrumentation uses information flow assertions as defined in Sect. 6.2. Again, we make use of the fact that modular products allow checking relational assertions at arbitrary program points and formulating assertions about the control flow.

We now prove that if an instrumented statement verifies under some 2-relational precondition then any two runs from a pair of states fulfilling that precondition will either both terminate or both loop forever.

Theorem 4. *If* $s' = term(s, false)$*, and* $[\![s']\!]_2^{\mathring{P}}$ *verifies under some precondition* $P = \lfloor \hat{P} \rfloor_2^{\mathring{P}}$*, and for some* $\sigma_1, \sigma_2, \sigma_1'$*,* $(\sigma_1, \sigma_2) \vDash \hat{P}$ *and* $\langle s, \sigma_1 \rangle \rightarrow^* \langle \texttt{skip}, \sigma_1' \rangle$*, then there exists some* σ_2' *s.t.* $\langle s, \sigma_2 \rangle \rightarrow^* \langle \texttt{skip}, \sigma_2' \rangle$.

Proof (Sketch). We first establish that our instrumentation ensures that each statement terminates (1) if and (2) only if its termination condition is true, (1) by showing equivalence to a standard termination proof, and (2) by a contradiction if a loop which should not terminate does. Since the execution from σ_1 terminates, by the second condition, its termination condition must have been true before the loop. We case split on whether the other execution also reaches the loop or not. If it does then the termination condition before the loop is identical in both executions, so by the first condition, the other execution also

terminates. If it does not then the loop is not executed at all by the other execution, and therefore cannot cause non-termination. □

6.6 Preventing Timing Channels

A program has a *timing channel* if high input data influences the program's execution time, meaning that an attacker who can observe the time the program executes can gain information about those secrets. Timing channels can occur in combination with observable events; the time at which an event occurs may depend on a secret even if the overall execution time of a program does not.

Consider the example in Fig. 7 (right). Assuming main receives a positive secret h, both the **print** statement and the end of the program execution will be reached later for larger values of h.

Using modular product programs, we can verify the absence of timing side channels by adding ghost state to the program that tracks the time passed since the program has started; this could, for example, be achieved via a simple step counting mechanism, or by tracking the sequence of previously executed bytecode statements. This ghost state is updated separately for both executions. We can then assert anywhere in the program that the passed time does not depend on high data in the same way we do for program variables. In particular, we can enforce that the passed time is equal whenever an observable event occurs, and we can enable users to write relational specifications that compare the time passed in both executions of a loop or a procedure.

7 Implementation and Evaluation

We have implemented our approach for secure information flow in the Viper verification infrastructure [22] and applied it to a number of example programs from the literature. Both the implementation and examples are available at http://viper.ethz.ch/modularproducts/.

7.1 Implementation in Viper

Our implementation supports a version of the Viper language that adds the following features:

1. The assertions *low*(*e*) and *lowEvent* for information flow specifications
2. A **declassify** statement
3. Variations of the existing method declarations and while loops that include the termination annotations shown in Sect. 6.5

The implementation transforms a program in this extended language into a modular 2-product in the original language, which can then be verified by the (unmodified) Viper back-end verifiers. All specifications are provided as information flow specifications (see Sect. 6.2) such that users require no knowledge

about the transformation or the methodology behind information flow verification. Error messages are automatically translated back to the original program.

Declassification is implemented as described in Sect. 6.4. Our implementation optionally verifies the absence of timing channels; the metric chosen for tracking execution time is simple step-counting. Viper uses implicit dynamic frames [25] to reason about heap-manipulating programs; our implementation uses quantified permissions [21] to support unbounded heap data structures.

For languages with opaque object references, secure information flow can require that pointers are low, i.e., equal up to a consistent renaming of addresses. Therefore, our approach to duplicating the heap state space in the implementation differs from that described in Sect. 4.3: Instead of duplicating objects, our implementation creates a single new statement for every new in the original program, but duplicates the fields each object has. As a result, if both executions execute the same new statement, the newly created object will be considered low afterwards (but the values of its fields might still be high).

7.2 Qualitative Evaluation

We have evaluated our implementation by verifying a number of examples in the extended Viper language. The examples are listed in Table 1 and include all code snippets shown in this paper as well as a number of examples from the literature [2,3,6,13,14,17,18,23,26,28]. They combine complex language features like mutable state on the heap, arrays and procedure calls, as well as timing and termination channels, declassification, and non-trivial information flows (e.g., flows whose legality depends on semantic information not available in a standard information flow type system). We manually added pre- and postconditions as well as loop invariants; for those that have forbidden flows and therefore should not verify, we also added a legal version that declassifies the leaked information. Our implementation returns the correct result for all examples.

In all cases but one, our approach allows us to express all information flow related assertions, i.e., procedure specifications and loop invariants, purely as relational specifications in terms of *low*-assertions (see Table 1). For all these examples, we completely avoid the need to specify the functional behavior of the program. Unlike the original product program paper [6], we also do not inline any procedure calls; verification is completely modular.

The only exception is an example that, depending on a high input, executes different loops with identical behavior, and for which we need to prove that the execution time is low. In this case we have to provide invariants for both loops that exactly specify their execution time in order to prove that the overall execution time after the conditional is low. Nevertheless, the specification of the procedure containing the loop is again expressed with a relational specification using only *low*. For all other examples, unary specifications were only needed to verify the absence of runtime errors (e.g., out-of-bounds array accesses), which Viper verifies by default. Consequently, a verified program cannot leak low data through such errors, which is typically not guaranteed by type systems or static analyses.

File	Event	Heap	Array	Decl.	Term.	Time	Call	LOC	Ann/SF/NI/TM/F	T_{VCG}	T_{SE}
antopolous1 [2]						x		25	7/3/3/0/2	0.78	1.10
antopolous2 [2]				x		x		61	14/0/14/0/0	0.72	0.91
banerjee [3]		x		x			x	76	17/11/6/0/0	1.02	0.61
constanzo [13]	x		x					22	7/2/5/0/0	0.67	0.28
darvas [14]		x		x				33	12/8/4/0/0	0.67	0.35
example			x				x	31	7/1/6/0/0	0.73	0.59
example_decl			x	x				19	5/2/3/0/0	0.72	0.77
example_term				x	x			31	8/4/2/2/0	0.77	0.43
example_time	x			x		x	x	32	9/0/9/0/0	0.70	0.38
joana_1_tl [17]	x			x			x	28	1/0/1/0/0	0.62	0.23
joana_2_bl [17]	x						x	18	2/0/2/0/0	0.63	0.25
joana_2_t [17]	x							15	1/0/1/0/0	0.62	0.20
joana_3_bl [17]	x			x	x		x	47	5/1/2/2/0	0.77	0.47
joana_3_br [17]	x			x	x		x	43	8/0/2/6/0	0.83	0.60
joana_3_tl [17]	x				x		x	33	8/2/2/4/0	0.75	0.53
joana_3_tr [17]	x			x	x		x	35	8/4/2/2/0	0.76	0.51
joana_13_l [17]							x	12	1/0/1/0/0	0.62	0.24
kusters [18]		x					x	29	9/6/3/0/0	0.64	0.44
naumann [23]		x	x					20	6/3/6/0/0	0.81	0.88
product [6]		x	x				x	65	30/21/21/0/0	5.47	15.73
smith [26]			x	x				43	12/6/8/0/0	0.87	0.89
terauchi1 [28]								14	2/0/2/0/0	0.62	0.26
terauchi2 [28]				x			x	21	4/0/4/0/0	0.63	0.30
terauchi3 [28]								24	5/1/4/0/0	0.66	0.40

Table 1. Evaluated examples. We show the used language features, lines of code including specifications, overall lines used for specifications (Ann), unary specifications for safety (SF), relational specifications for non-interference (NI), specifications for termination (TM), and functional specifications required for non-interference (F). Note that some lines contain specifications belonging to multiple categories. Columns T_{SE} and T_{VCG} show the running times of the verifiers for the SE backend and the VCG backend, respectively, in seconds.

7.3 Performance

For all but one example, the runtime (averaged over 10 runs on a Lenovo ThinkPad T450s running Ubuntu) with both the Symbolic Execution (SE) and the Verification Condition Generation (VCG) verifiers is under or around one second (see Table 1). The one exception, which makes extensive use of unbounded heap data structures, takes ca. five seconds when verified using VCG, and 15 in the SE verifier. This is likely a result of inefficiencies in our encoding: The created product has a high number of branching statements, and some properties have to be proved more than once, two issues which have a much larger performance impact for SE than for VCG. We believe that it is feasible to remove much of this overhead by optimizing the encoding; we leave this as future work.

8 Related Work

The notion of k-safety hyperproperties was originally introduced by Clarkson and Schneider [12]. Here, we focus on statically proving hyperproperties for imperative and object-oriented programs; much more work exists for testing or monitoring hyperproperties like secure information flow at runtime, or for reasoning about hyperproperties in different programming paradigms.

Relational logics such as Relational Hoare Logic [11], Relational Separation Logic [29] and others [1,10] allow reasoning directly about relational properties of two different program executions. Unlike our approach, they usually allow reasoning about the executions of two *different* programs; as a result, they do not give special support for two executions of the same program calling the same procedure with a relational specification. Recently, Banerjee et al. [5] introduced biprograms, which allow explicitly expressing alignment between executions and using relational specifications to reason about aligned calls; however, this approach requires that procedures with relational specifications are always called by both executions, which is for instance not the case if a call occurs under a high guard in secure information flow verification. We handle such cases by interpreting relational specifications as trivially true; one can then still resort to functional specifications to complete the proof. Their work also does not allow mixed specifications, which are easily supported in our product programs. Relational program logics are generally difficult to automate. Recent work by Sousa and Dillig [27] presents a logic that can be applied automatically by an algorithm that implicitly constructs different product programs that align *some* identical statements, but does not fully support relational specifications. Moreover, their approach requires dedicated tool support, whereas our modular product programs can be verified using off-the-shelf tools.

The approach of reducing hyperproperties to ordinary trace properties was introduced by self-composition [9]. While self-composition is theoretically complete, it does not allow modular reasoning with relational specifications. The resulting problem of having to fully specify program behavior was pointed out by Terauchi and Aiken [28]; since then, there have been a number of different attempts to solve this problem by allowing (parts of) programs to execute in lock-step. Terauchi and Aiken [28] did this for secure information flow by relying on information from a type system; other similar approaches exist [23].

Product programs [6,7] allow different interleavings of program executions. The initial product program approach [6] would in principle allow the use of relational specifications for procedure calls, but only under the restriction that both program executions always follow the same control flow. The generalized approach [7] allows combining different programs and arbitrary numbers of executions. This product construction is non-deterministic and usually interactive. In some (but not all) cases, programmers can manually construct product programs that avoid duplicated calls and loops and thereby allow using relational specifications. However, whether this is possible depends on the used specification, meaning that the product construction and verification are intertwined and a new product has to be constructed when specifications change. In contrast, our

new product construction is fully deterministic and automatic, allows arbitrary control flows while still being able to use relational specifications for all loops and calls, and therefore avoids the issue of requiring full functional specifications.

Considerable work has been invested into proving specific hyperproperties like secure information flow. One popular approach is the use of type systems [26]; while those are modular and offer good performance, they overapproximate possible program behaviors and are therefore less precise than approaches using logics. In particular, they require labeling any single value as either high or low, and do not allow distinctions like the one we made for the example in Fig. 1, where only the first bits of a sequence of integers were low. In addition, type systems typically struggle to prevent information leaks via side channels like termination or program aborts. There have been attempts to create type systems that handle some of these limitations (e.g. [15]).

Static analyses [2,17] enable fully automatic reasoning. They are typically not modular and, similarly to type systems, need to abstract semantic information, which can lead to false positives. They strike a trade-off different from our solution, which requires specifications, but enables precise, modular reasoning.

A number of logic-based approaches to proving specific hyperproperties exist. As an example, Darvas et al. use dynamic logic for proving non-interference [14]; this approach offers some automation, but requires user interaction for most realistic programs. Leino et al. [19] verify determinism up to equivalence using self-composition, which suffers from the drawbacks explained above.

Different kinds of declassification have been studied extensively, Sabelfeld and Sands [24] provide a good overview. Li and Zdancewic [20] introduce downgrading policies that describe which information can be declassified and, similar to our approach, can do so for arbitrary expressions.

9 Conclusion and Future Work

We have presented modular product programs, a novel form of product programs that enable modular reasoning about k-safety hyperproperties using relational specifications with off-the-shelf verifiers. We showed that modular products are expressive enough to handle advanced aspects of secure information flow verification. They can prove the absence of termination and timing side channels and encode declassification. Our implementation shows that our technique works in practice on a number of challenging examples from the literature, and exhibits good performance even without optimizations.

For future work, we plan to infer relational properties by using standard program analysis techniques on the products. We also plan to generalize our technique to prove probabilistic secure information flow for concurrent program by combining our encoding with ideas from concurrent separation logic. Finally, we plan to optimize our encoding to further improve performance.

Acknowledgements. We would like to thank Toby Murray and David Naumann for various helpful discussions. We are grateful to the anonymous reviewes for their valuable comments. We also gratefully acknowledge support from the Zurich Information Security and Privacy Center (ZISC).

References

1. Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Strub, P.: A relational logic for higher-order programs. PACMPL **1**(ICFP), 21:1–21:29 (2017)
2. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 362–375 (2017)
3. Banerjee, A., Naumann, D.A., Secure information flow and pointer confinement in a java-like language. In: 15th IEEE Computer Security Foundations Workshop (CSFW-15 2002), 24–26 June 2002, Cape Breton, Nova Scotia, Canada, p. 253 (2002)
4. Banerjee, A., Naumann, D.A.: A logical analysis of framing for specifications with pure method calls. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 3–20. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_1
5. Banerjee, A., Naumann, D.A., Nikouei, M.: Relational logic with framing and hypotheses. In: 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, Chennai, India, 13–15 December 2016, pp. 11:1–11:16 (2016)
6. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
7. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: asymmetric product programs for relational program verification. In: Artemov, S., Nerode, A. (eds.) LFCS 2013. LNCS, vol. 7734, pp. 29–43. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35722-0_3
8. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28–30 June 2004, Pacific Grove, CA, USA, pp. 100–114 (2004)
9. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. **21**(6), 1207–1252 (2011)
10. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, 21–23 January 2009, pp. 90–101 (2009)
11. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, 14–16 January 2004, pp. 14–25 (2004)
12. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
13. Costanzo, D., Shao, Z.: A separation logic for enforcing declarative information flow control policies. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 179–198. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_10
14. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32004-3_20

15. Deng, Z., Smith, G.: Lenient array operations for practical secure information flow. In: 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28–30 June 2004, Pacific Grove, CA, USA, p. 115 (2004)
16. Francez, N.: Fairness. Springer-Verlag, New York Inc., New York (1986). https://doi.org/10.1007/978-1-4612-4886-6
17. Giffhorn, D., Snelting, G.: A new algorithm for low-deterministic security. Int. J. Inf. Sec. **14**(3), 263–287 (2015)
18. Küsters, R., Truderung, T., Beckert, B., Bruns, D., Kirsten, M., Mohr, M.: A hybrid approach for proving noninterference of java programs. In: IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13–17 July 2015, pp. 305–319 (2015)
19. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 307–321. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78739-6_24
20. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, 12–14 January 2005, pp. 158–170 (2005)
21. Müller, P., Schwerhoff, M., Summers, A.J.: Automatic verification of iterated separating conjunctions using symbolic execution. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part I. LNCS, vol. 9779, pp. 405–425. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_22
22. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
23. Naumann, D.A.: From coupling relations to mated invariants for checking information flow. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 279–296. Springer, Heidelberg (2006). https://doi.org/10.1007/11863908_18
24. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: 18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20–22 June 2005, Aix-en-Provence, France, pp. 255–269 (2005)
25. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (2012)
26. Smith, G.: Principles of secure information flow analysis. In: Christodorescu, M., Jha, S., Maughan, D., Song, D., Wang, C. (eds.) Malware Detection. ADIS, vol. 27, pp. 291–307. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-44599-1_13
27. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, 13–17 June 2016, pp. 57–69 (2016)
28. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005). https://doi.org/10.1007/11547662_24
29. Yang, H.: Relational separation logic. Theor. Comput. Sci. **375**(1–3), 308–334 (2007)

Program Verification

A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification

Armaël Guéneau[1], Arthur Charguéraud[1,2], and François Pottier[1(✉)]

[1] Inria, Paris, France
francois.pottier@inria.fr
[2] Université de Strasbourg, CNRS, ICube UMR 7357, Strasbourg, France

Abstract. We present a framework for simultaneously verifying the functional correctness and the worst-case asymptotic time complexity of higher-order imperative programs. We build on top of Separation Logic with Time Credits, embedded in an interactive proof assistant. We formalize the O notation, which is key to enabling modular specifications and proofs. We cover the subtleties of the multivariate case, where the complexity of a program fragment depends on multiple parameters. We propose a way of integrating complexity bounds into specifications, present lemmas and tactics that support a natural reasoning style, and illustrate their use with a collection of examples.

1 Introduction

A program or program component whose functional correctness has been verified might nevertheless still contain complexity bugs: that is, its performance, in some scenarios, could be much poorer than expected.

Indeed, many program verification tools only guarantee partial correctness, that is, do not even guarantee termination, so a verified program could run forever. Some program verification tools do enforce termination, but usually do not allow establishing an explicit complexity bound. Tools for automatic complexity inference can produce complexity bounds, but usually have limited expressive power.

In practice, many complexity bugs are revealed by testing. Some have also been detected during ordinary program verification, as shown by Filliâtre and Letouzey [14], who find a violation of the balancing invariant in a widely-distributed implementation of binary search trees. Nevertheless, none of these techniques can guarantee, with a high degree of assurance, the absence of complexity bugs in software.

To illustrate the issue, consider the binary search implementation in Fig. 1. Virtually every modern software verification tool allows proving that this OCaml

This research was partly supported by the French National Research Agency (ANR) under the grant ANR-15-CE25-0008.

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 533–560, 2018.
https://doi.org/10.1007/978-3-319-89884-1_19

code (or analogous code, expressed in another programming language) satisfies the specification of a binary search and terminates on all valid inputs. This code might even pass a lightweight testing process, as some search queries will be answered very quickly, even if the array is very large. Yet, a more thorough testing process would reveal a serious issue: a search for a value that is stored in the second half of the range $[i, j)$ takes linear time. It would be embarrassing if such faulty code was deployed, as it would aggravate benevolent users and possibly allow malicious users to mount denial-of-service attacks.

```
(* Requires t to be a sorted array of integers.
   Returns k such that i <= k < j and t.(k) = v
   or -1 if there is no such k. *)
let rec bsearch t v i j =
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if v = t.(k) then k
    else if v < t.(k) then bsearch t v i k
    else bsearch t v (i+1) j
```

Fig. 1. A flawed binary search. This code is provably correct and terminating, yet exhibits linear (instead of logarithmic) time complexity for some input parameters.

As illustrated above, complexity bugs can affect execution time, but could also concern space (including heap space, stack space, and disk space) or other resources, such as the network, energy, and so on. In this paper, for simplicity, we focus on execution time only. That said, much of our work is independent of which resource is considered. We expect that our techniques could be adapted to verify asymptotic bounds on the use of other non-renewable resources, such as the network.

We work with a simple model of program execution, where certain operations, such as calling a function or entering a loop body, cost one unit of time, and every other operation costs nothing. Although this model is very remote from physical running time, it is independent of the compiler, operating system, and hardware [18,24] and still allows establishing asymptotic time complexity bounds, and therefore, detecting complexity bugs—situations where a program is asymptotically slower than it should be.

In prior work [11], the second and third authors present a method for verifying that a program satisfies a specification that includes an explicit bound on the program's worst-case, amortized time complexity. They use Separation Logic with Time Credits, a simple extension of Separation Logic [23] where the assertion \$1 represents a permission to perform one step of computation, and is consumed when exercised. The assertion $\$n$ is a separating conjunction of n such time credits. Separation Logic with Time Credits is implemented in the second author's interactive verification framework, CFML [9,10], which is embedded in the Coq proof assistant.

Using CFML, the second and third authors verify the correctness and time complexity of an OCaml implementation of the Union-Find data structure [11]. However, their specifications involve *concrete* cost functions: for instance, the precondition of the function *find* indicates that calling *find* requires and consumes $\$(2\alpha(n) + 4)$, where n is the current number of elements in the data structure, and where α denotes an inverse of Ackermann's function. We would prefer the specification to give the *asymptotic* complexity bound $O(\alpha(n))$, which means that, for *some* function $f \in O(\alpha(n))$, calling *find* requires and consumes $\$f(n)$. This is the purpose of this paper.

We argue that the use of asymptotic bounds, such as $O(\alpha(n))$, is necessary for (verified or unverified) complexity analysis to be applicable at scale. At a superficial level, it reduces clutter in specifications and proofs: $O(mn)$ is more compact and readable than $3mn + 2n \log n + 5n + 3m + 2$. At a deeper level, it is crucial for stating modular specifications, which hide the details of a particular implementation. Exposing the fact that *find* costs $2\alpha(n) + 4$ is undesirable: if a tiny modification of the Union-Find module changes this cost to $2\alpha(n) + 5$, then all direct and indirect clients of the Union-Find module must be updated, which is intolerable. Furthermore, sometimes, the constant factors are unknown anyway. Applying the Master Theorem [12] to a recurrence equation only yields an order of growth, not a concrete bound. Finally, for most practical purposes, no critical information is lost when concrete bounds such as $2\alpha(n) + 4$ are replaced with asymptotic bounds such as $O(\alpha(n))$. Indeed, the number of computation steps that take place at the source level is related to physical time only up to a hardware- and compiler-dependent constant factor. The use of asymptotic complexity in the analysis of algorithms, initially advocated by Hopcroft and by Tarjan, has been widely successful and is nowadays standard practice.

One must be aware of several limitations of our approach. First, it is not a worst-case execution time (WCET) analysis: it does not yield bounds on actual physical execution time. Second, it is not fully automated. We place emphasis on expressiveness, as opposed to automation. Our vision is that verifying the functional correctness *and* time complexity of a program, at the same time, should not involve much more effort than verifying correctness alone. Third, we control only the growth of the cost as the parameters grow large. A loop that counts up from 0 to 2^{60} has complexity $O(1)$, even though it typically won't terminate in a lifetime. Although this is admittedly a potential problem, traditional program verification falls prey to analogous pitfalls: for instance, a program that attempts to allocate and initialize an array of size (say) 2^{48} can be proved correct, even though, on contemporary desktop hardware, it will typically fail by lack of memory. We believe that there is value in our approach in spite of these limitations.

Reasoning and working with asymptotic complexity bounds is not as simple as one might hope. As demonstrated by several examples in Sect. 2, typical paper proofs using the O notation rely on informal reasoning principles which can easily be abused to prove a contradiction. Of course, using a proof assistant steers us

clear of this danger, but implies that our proofs cannot be quite as simple and perhaps cannot have quite the same structure as their paper counterparts.

A key issue that we run against is the handling of existential quantifiers. According to what was said earlier, the specification of a sorting algorithm, say *mergesort*, should be, roughly: "there exists a cost function $f \in O(\lambda n.n \log n)$ such that *mergesort* is content with $\$f(n)$, where n is the length of the input list." Therefore, the very first step in a naïve proof of *mergesort* must be to exhibit a witness for f, that is, a concrete cost function. An appropriate witness might be $\lambda n.2n \log n$, or $\lambda n.n \log n + 3$, who knows? This information is not available up front, at the very *beginning* of the proof; it becomes available only *during* the proof, as we examine the code of *mergesort*, step by step. It is not reasonable to expect the human user to guess such a witness. Instead, it seems desirable to *delay* the production of the witness and to *gradually* construct a cost expression as the proof progresses. In the case of a nonrecursive function, such as *insertionsort*, the cost expression, once fully synthesized, yields the desired witness. In the case of a recursive function, such as *mergesort*, the cost expression yields the body of a recurrence equation, whose solution is the desired witness.

We make the following contributions:

1. We formalize O as a binary *domination* relation between functions of type $A \to \mathbb{Z}$, where the type A is chosen by the user. Functions of several variables are covered by instantiating A with a product type. We contend that, in order to define what it means for $a \in A$ to "grow large", or "tend towards infinity", the type A must be equipped with a filter [6], that is, a quantifier $\mathbb{U}a.P$. (Eberl [13] does so as well.) We propose a library of lemmas and tactics that can prove nonnegativeness, monotonicity, and domination assertions (Sect. 3).
2. We propose a standard style of writing specifications, in the setting of the CFML program verification framework, so that they integrate asymptotic time complexity claims (Sect. 4). We define a predicate, `specO`, which imposes this style and incorporates a few important technical decisions, such as the fact that every cost function must be nonnegative and nondecreasing.
3. We propose a methodology, supported by a collection of Coq tactics, to prove such specifications (Sect. 5). Our tactics, which heavily rely on Coq metavariables, help gradually synthesize cost expressions for straight-line code and conditionals, and help construct the recurrence equations involved in the analysis of recursive functions, while delaying their resolution.
4. We present several classic examples of complexity analyses (Sect. 6), including: a simple loop in $O(n.2^n)$, nested loops in $O(n^3)$ and $O(nm)$, binary search in $O(\log n)$, and Union-Find in $O(\alpha(n))$.

Our code can be found online in the form of two standalone Coq libraries and a self-contained archive [16].

2 Challenges in Reasoning with the *O* Notation

When informally reasoning about the complexity of a function, or of a code block, it is customary to make assertions of the form "this code has asymptotic

complexity $O(1)$", "that code has asymptotic complexity $O(n)$", and so on. Yet, these assertions are too informal: they do not have sufficiently precise meaning, and can be easily abused to produce flawed paper proofs.

A striking example appears in Fig. 2, which shows how one might "prove" that a recursive function has complexity $O(1)$, whereas its actual cost is $O(n)$. The flawed proof exploits the (valid) relation $O(1) + O(1) = O(1)$, which means that a sequence of two constant-time code fragments is itself a constant-time code fragment. The flaw lies in the fact that the O notation hides an existential quantification, which is inadvertently swapped with the universal quantification over the parameter n. Indeed, the claim is that "there exists a constant c such that, for every n, `waste`(n) runs in at most c computation steps". However, the proposed proof by induction establishes a much weaker result, to wit: "for every n, there exists a constant c such that `waste`(n) runs in at most c steps". This result is certainly true, yet does not entail the claim.

An example of a different nature appears in Fig. 3. There, the auxiliary function `g` takes two integer arguments n and m and involves two nested loops, over the intervals $[1, n]$ and $[1, m]$. Its asymptotic complexity is $O(n + nm)$, which, *under the hypothesis that m is large enough*, can be simplified to $O(nm)$. The reasoning, thus far, is correct. The flaw lies in our attempt to substitute 0 for m

Incorrect claim: The OCaml function `waste` has asymptotic complexity $O(1)$.

```
let rec waste n =
  if n > 0 then waste (n-1)
```

Flawed proof:
Let us prove by induction on n that `waste`(n) costs $O(1)$.

- **Case** $n \leq 0$: `waste`(n) terminates immediately. Therefore, its cost is $O(1)$.
- **Case** $n > 0$: A call to `waste`(n) involves constant-time processing, followed with a call to `waste`$(n-1)$. By the induction hypothesis, the cost of the recursive call is $O(1)$. We conclude that the cost of `waste`(n) is $O(1) + O(1)$, that is, $O(1)$.

Fig. 2. A flawed proof that `waste`(n) costs $O(1)$, when its actual cost is $O(n)$.

Incorrect claim: The OCaml function `f` has asymptotic complexity $O(1)$.

```
let g (n, m) =
  for i = 1 to n do
    for j = 1 to m do () done
  done
let f n = g (n, 0)
```

Flawed proof:

- `g`(n, m) involves nm inner loop iterations, thus costs $O(nm)$.
- The cost of `f`(n) is the cost of `g`$(n, 0)$, plus $O(1)$. As the cost of `g`(n, m) is $O(nm)$, we find, by substituting 0 for m, that the cost of `g`$(n, 0)$ is $O(0)$. Thus, `f`(n) is $O(1)$.

Fig. 3. A flawed proof that `f`(n) costs $O(1)$, when its actual cost is $O(n)$.

Incorrect claim: The OCaml function `h` has asymptotic complexity $O(nm^2)$.

```
1    let h (m, n) =
2      for i = 0 to m−1 do
3        let p = (if i = 0 then pow2 n else n∗i) in
4        for j = 1 to p do () done
5      done
```

Flawed proof:

- The body of the outer loop (lines 3-4) has asymptotic cost $O(ni)$. Indeed, as soon as $i > 0$ holds, the inner loop performs ni constant-time iterations. The case where $i = 0$ does not matter in an asymptotic analysis.
- The cost of $\mathtt{h}(m, n)$ is the sum of the costs of the iterations of the outer loop:

$$\sum_{i=0}^{m-1} O(ni) \;=\; O\left(n \cdot \sum_{i=0}^{m-1} i\right) \;=\; O(nm^2).$$

Fig. 4. A flawed proof that $\mathtt{h}(m, n)$ costs $O(nm^2)$, when its actual cost is $O(2^n + nm^2)$.

in the bound $O(nm)$. Because this bound is valid only for sufficiently large m, it does not make sense to substitute a specific value for m. In other words, from the fact that "$\mathtt{g}(n, m)$ costs $O(nm)$ when n and m are sufficiently large", one *cannot* deduce anything about the cost of $\mathtt{g}(n, 0)$. To repair this proof, one must take a step back and prove that $\mathtt{g}(n, m)$ has asymptotic complexity $O(n + nm)$ *for sufficiently large n and for every m.* This fact *can* be instantiated with $m = 0$, allowing one to correctly conclude that $\mathtt{g}(n, 0)$ costs $O(n)$. We come back to this example in Sect. 3.3.

One last example of tempting yet invalid reasoning appears in Fig. 4. We borrow it from Howell [19]. This flawed proof exploits the dubious idea that "the asymptotic cost of a loop is the sum of the asymptotic costs of its iterations". In more precise terms, the proof relies on the following implication, where $f(m, n, i)$ represents the true cost of the i-th loop iteration and $g(m, n, i)$ represents an asymptotic bound on $f(m, n, i)$:

$$f(m, n, i) \in O(g(m, n, i)) \quad \Rightarrow \quad \sum_{i=0}^{m-1} f(m, n, i) \in O\left(\sum_{i=0}^{m-1} g(m, n, i)\right)$$

As pointed out by Howell, this implication is in fact invalid. Here, $f(m, n, 0)$ is 2^n and $f(m, n, i)$ when $i > 0$ is ni, while $g(m, n, i)$ is just ni. The left-hand side of the above implication holds, but the right-hand side does not, as $2^n + \sum_{i=1}^{m-1} ni$ is $O(2^n + nm^2)$, not $O(nm^2)$. The Summation lemma presented later on in this paper (Lemma 8) rules out the problem by adding the requirement that f be a nondecreasing function of the loop index i. We discuss in depth later on (Sect. 4.5) why cost functions should and can be monotonic.

The examples that we have presented show that the informal reasoning style of paper proofs, where the O notation is used in a loose manner, is unsound. One cannot hope, in a formal setting, to faithfully mimic this reasoning style. In this paper, we do assign O specifications to functions, because we believe that

this style is elegant, modular and scalable. However, during the analysis of a function body, we abandon the O notation. We first synthesize a cost expression for the function body, then check that this expression is indeed dominated by the asymptotic bound that appears in the specification.

3 Formalizing the O Notation

3.1 Domination

In many textbooks, the fact that f is bounded above by g asymptotically, up to constant factor, is written "$f = O(g)$" or "$f \in O(g)$". However, the former notation is quite inappropriate, as it is clear that "$f = O(g)$" cannot be literally understood as an equality. Indeed, if it truly were an equality, then, by symmetry and transitivity, $f_1 = O(g)$ and $f_2 = O(g)$ would imply $f_1 = f_2$. The latter notation makes much better sense: $O(g)$ is then understood as a set of functions. This approach has in fact been used in formalizations of the O notation [3]. Yet, in this paper, we prefer to think directly in terms of a *domination* preorder between functions. Thus, instead of "$f \in O(g)$", we write $f \preceq g$.

Although the O notation is often defined in the literature only in the special case of functions whose domain is $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{R}$, we must define domination in the general case of functions whose domain is an arbitrary type A. By later instantiating A with a product type, such as $\mathbb{Z}^k$, we get a definition of domination that covers the multivariate case. Thus, let us fix a type A, and let f and g inhabit the function type $A \to \mathbb{Z}$.[1]

Fixing the type A, it turns out, is not quite enough. In addition, the type A must be equipped with a *filter* [6]. To see why that is the case, let us work towards the definition of domination. As is standard, we wish to build a notion of "growing large enough" into the definition of domination. That is, instead of requiring a relation of the form $|f(x)| \leq c\,|g(x)|$ to be "everywhere true", we require it to be "ultimately true", that is, "true when x is large enough".[2] Thus, $f \preceq g$ should mean, roughly:

> "up to a constant factor, ultimately, $|f|$ is bounded above by $|g|$."

That is, somewhat more formally:

> "for some c, for every sufficiently large x, $|f(x)| \leq c\,|g(x)|$"

In mathematical notation, we would like to write: $\exists c.\ \mathbb{U}x.\ |f(x)| \leq c\,|g(x)|$. For such a formula to make sense, we must define the meaning of the formula $\mathbb{U}x.P$, where x inhabits the type A. This is the reason why the type A must be

[1] At this time, we require the codomain of f and g to be $\mathbb{Z}$. Following Avigad and Donnelly [3], we could allow it to be an arbitrary nondegenerate ordered ring. We have not yet needed this generalization.

[2] When A is $\mathbb{N}$, provided $g(x)$ is never zero, requiring the inequality to be "everywhere true" is in fact the same as requiring it to be "ultimately true". Outside of this special case, however, requiring the inequality to hold everywhere is usually too strong.

equipped with a filter $\mathbb{U}$, which intuitively should be thought of as a quantifier, whose meaning is "ultimately". Let us briefly defer the definition of a filter (Sect. 3.2) and sum up what has been explained so far:

Definition 1 (Domination). *Let A be a filtered type, that is, a type A equipped with a filter $\mathbb{U}_A$.*

The relation $\preceq_A$ on $A \to \mathbb{Z}$ is defined as follows:

$$f \preceq_A g \quad \equiv \quad \exists c.\, \mathbb{U}_A\, x.\, |f(x)| \leq c\,|g(x)|$$

3.2 Filters

Whereas $\forall x.P$ means that P holds of *every* x, and $\exists x.P$ means that P holds of *some* x, the formula $\mathbb{U}x.P$ should be taken to mean that P holds of every *sufficiently large* x, that is, P *ultimately* holds.

The formula $\mathbb{U}x.P$ is short for $\mathbb{U}\ (\lambda x.P)$. If x ranges over some type A, then $\mathbb{U}$ must have type $\mathcal{P}(\mathcal{P}(A))$, where $\mathcal{P}(A)$ is short for $A \to \text{Prop}$. To stress this better, although Bourbaki [6] states that a filter is "a set of subsets of A", it is crucial to note that $\mathcal{P}(\mathcal{P}(A))$ is the type of a quantifier in higher-order logic.

Definition 2 (Filter). *A* filter [6] *on a type A is an object $\mathbb{U}$ of type $\mathcal{P}(\mathcal{P}(A))$ that enjoys the following four properties, where $\mathbb{U}x.P$ is short for $\mathbb{U}\ (\lambda x.P)$:*

(1)	$(P_1 \Rightarrow P_2) \Rightarrow \mathbb{U}x.P_1 \Rightarrow \mathbb{U}x.P_2$	*(covariance)*
(2a)	$\mathbb{U}x.P_1 \wedge \mathbb{U}x.P_2 \Rightarrow \mathbb{U}x.(P_1 \wedge P_2)$	*(stability under binary intersection)*
(2b)	$\mathbb{U}x.\mathit{True}$	*(stability under 0-ary intersection)*
(3)	$\mathbb{U}x.P \Rightarrow \exists x.P$	*(nonemptiness)*

Properties (1)–(3) are intended to ensure that the intuitive reading of $\mathbb{U}x.P$ as: "for sufficiently large x, P holds" makes sense. Property (1) states that if P_1 implies P_2 and if P_1 holds when x is large enough, then P_2, too, should hold when x is large enough. Properties (2a) and (2b), together, state that if each of $P_1, \ldots, P_k$ independently holds when x is large enough, then $P_1, \ldots, P_k$ should simultaneously hold when x is large enough. Properties (1) and (2b) together imply $\forall x.P \Rightarrow \mathbb{U}x.P$. Property (3) states that if P holds when x is large enough, then P should hold of some x. In classical logic, it would be equivalent to $\neg(\mathbb{U}x.\text{False})$.

In the following, we let the metavariable A stand for a *filtered type*, that is, a pair of a carrier type and a filter on this type. By abuse of notation, we also write A for the carrier type. (In Coq, this is permitted by an implicit projection.) We write $\mathbb{U}_A$ for the filter.

3.3 Examples of Filters

When $\mathbb{U}$ is a *universal filter*, $\mathbb{U}x.Q(x)$ is (by definition) equivalent to $\forall x.Q(x)$. Thus, a predicate Q is "ultimately true" if and only if it is "everywhere true". In other words, the universal quantifier is a filter.

Definition 3 (Universal filter). *Let T be a nonempty type. Then $\lambda Q.\forall x.Q(x)$ is a filter on T.*

When $\mathbb{U}$ is the *order filter* associated with the ordering $\leq$, the formula $\mathbb{U}x.Q(x)$ means that, when x becomes sufficiently large with respect to $\leq$, the property $Q(x)$ becomes true.

Definition 4 (Order filter). *Let $(T, \leq)$ be a nonempty ordered type, such that every two elements have an upper bound. Then $\lambda Q.\exists x_0.\forall x \geq x_0.\ Q(x)$ is a filter on T.*

The order filter associated with the ordered type $(\mathbb{Z}, \leq)$ is the most natural filter on the type $\mathbb{Z}$. Equipping the type $\mathbb{Z}$ with this filter yields a filtered type, which, by abuse of notation, we also write $\mathbb{Z}$. Thus, the formula $\mathbb{U}_{\mathbb{Z}}\, x.Q(x)$ means that $Q(x)$ becomes true "as x tends towards infinity".

By instantiating Definition 1 with the filtered type $\mathbb{Z}$, we recover the classic definition of domination between functions of $\mathbb{Z}$ to $\mathbb{Z}$:

$$f \preceq_{\mathbb{Z}} g \iff \exists c.\ \exists n_0.\ \forall n \geq n_0.\ |f(n)| \leq c\,|g(n)|$$

We now turn to the definition of a filter on a product type $A_1 \times A_2$, where A_1 and A_2 are filtered types. Such a filter plays a key role in defining domination between functions of several variables. The following *product filter* is the most natural construction, although there are others:

Definition 5 (Product filter). *Let A_1 and A_2 be filtered types. Then*

$$\lambda Q.\exists Q_1, Q_2.\ \begin{cases} \mathbb{U}_{A_1}\, x_1.\ Q_1 \\ \wedge\ \mathbb{U}_{A_2}\, x_2.\ Q_2 \\ \wedge\ \forall x_1, x_2.\ Q_1(x_1) \wedge Q_2(x_2) \Rightarrow Q(x_1, x_2) \end{cases}$$

is a filter on the product type $A_1 \times A_2$.

To understand this definition, it is useful to consider the special case where A_1 and A_2 are both $\mathbb{Z}$. Then, for $i \in \{1, 2\}$, the formula $\mathbb{U}_{A_i}\, x_i.\ Q_i$ means that the predicate Q_i contains an infinite interval of the form $[a_i, \infty)$. Thus, the formula $\forall x_1, x_2.\ Q_1(x_1) \wedge Q_2(x_2) \Rightarrow Q(x_1, x_2)$ requires the predicate Q to contain the infinite rectangle $[a_1, \infty) \times [a_2, \infty)$. Thus, a predicate Q on $\mathbb{Z}^2$ is "ultimately true" w.r.t. to the product filter if and only if it is "true on some infinite rectangle". In Bourbaki's terminology [6, Chap. 1, Sect. 6.7], the infinite rectangles form a *basis* of the product filter.

We view the product filter as the default filter on the product type $A_1 \times A_2$. Whenever we refer to $A_1 \times A_2$ in a setting where a filtered type is expected, the product filter is intended.

We stress that there are several filters on $\mathbb{Z}$, including the universal filter and the order filter, and therefore several filters on $\mathbb{Z}^k$. Therefore, it does not make sense to use the O notation without specifying which filter one considers. Consider again the function $\mathtt{g}(n, m)$ in Fig. 3 (Sect. 2). One can prove that

$\mathsf{g}(n,m)$ has complexity $O(nm+n)$ with respect to the standard filter on $\mathbb{Z}^2$. With respect to *this filter*, this complexity bound is equivalent to $O(mn)$, as the functions $\lambda(m,n).mn+n$ and $\lambda(m,n).mn$ dominate each other. Unfortunately, this *does not allow* deducing anything about the complexity of $\mathsf{g}(n,0)$, since the bound $O(mn)$ holds only when n and m grow large. An alternate approach is to prove that $\mathsf{g}(n,m)$ has complexity $O(nm+n)$ with respect to a stronger filter, namely the product of the standard filter on $\mathbb{Z}$ and the universal filter on $\mathbb{Z}$. With respect to *that filter*, the functions $\lambda(m,n).mn+n$ and $\lambda(m,n).mn$ are *not* equivalent. This bound *does allow* instantiating m with 0 and deducing that $\mathsf{g}(n,0)$ has complexity $O(n)$.

3.4 Properties of Domination

Many properties of the domination relation can be established with respect to an arbitrary filtered type A. Here are two example lemmas; there are many more. As before, f and g range over $A \to \mathbb{Z}$. The operators $f+g$, $\max(f,g)$ and $f.g$ denote pointwise sum, maximum, and product, respectively.

Lemma 6 (Sum and Max Are Alike). *Assume f and g are ultimately nonnegative, that is, $\mathbb{U}_A\, x.\ f(x) \geq 0$ and $\mathbb{U}_A\, x.\ g(x) \geq 0$ hold. Then, we have $\max(f,g) \preceq_A f+g$ and $f+g \preceq_A \max(f,g)$.*

Lemma 7 (Multiplication). *$f_1 \preceq_A g_1$ and $f_2 \preceq_A g_2$ imply $f_1.f_2 \preceq_A g_1.g_2$.*

Lemma 7 corresponds to Howell's Property 5 [19]. Whereas Howell states this property on $\mathbb{N}^k$, our lemma is polymorphic in the type A. As noted by Howell, this lemma is useful when the cost of a loop body is independent of the loop index. In the case where the cost of the i-th iteration may depend on the loop index i, the following, more complex lemma is typically used instead:

Lemma 8 (Summation). *Let f,g range over $A \to \mathbb{Z} \to \mathbb{Z}$. Let $i_0 \in \mathbb{Z}$. Assume the following three properties:*

1. $\mathbb{U}_A\, a.\ \forall i \geq i_0.\ f(a)(i) \geq 0$.
2. $\mathbb{U}_A\, a.\ \forall i \geq i_0.\ g(a)(i) \geq 0$.
3. *for every a, the function $\lambda i.f(a)(i)$ is nondecreasing on the interval $[i_0,\infty)$.*

Then,

$$\lambda(a,i).f(a)(i) \ \preceq_{A\times\mathbb{Z}}\ \lambda(a,i).g(a)(i)$$

implies

$$\lambda(a,n).\ \textstyle\sum_{i=i_0}^{n} f(a)(i) \ \preceq_{A\times\mathbb{Z}}\ \lambda(a,n).\ \sum_{i=i_0}^{n} g(a)(i).$$

Lemma 8 uses the product filter on $A\times\mathbb{Z}$ in its hypothesis and conclusion. It corresponds to Howell's property 2 [19]. The variable i represents the loop index, while the variable a collectively represents all other variables in scope, so the type A is usually instantiated with a tuple type (an example appears in Sect. 6).

An important property is the fact that function composition is compatible, in a certain sense, with domination. This allows transforming the parameters under which an asymptotic analysis is carried out (examples appear in Sect. 6). Due to space limitations, we refer the reader to the Coq library for details [16].

3.5 Tactics

Our formalization of filters and domination forms a stand-alone Coq library [16]. In addition to many lemmas about these notions, the library proposes automated tactics that can prove nonnegativeness, monotonicity, and domination goals. These tactics currently support functions built out of variables, constants, sums and maxima, products, powers, logarithms. Extending their coverage is ongoing work. This library is not tied to our application to the complexity analysis of programs. It could have other applications in mathematics.

4 Specifications with Asymptotic Complexity Claims

In this section, we first present our existing approach to verified time complexity analysis. This approach, proposed by the second and third authors [11], does not use the O notation: instead, it involves explicit cost functions. We then discuss how to extend this approach with support for asymptotic complexity claims. We find that, even once domination (Sect. 3) is well-understood, there remain nontrivial questions as to the style in which program specifications should be written. We propose one style which works well on small examples and which we believe should scale well to larger ones.

4.1 CFML with Time Credits for Cost Analysis

CFML [9,10] is a system that supports the interactive verification of OCaml programs, using higher-order Separation Logic, inside Coq. It is composed of a trusted standalone tool and a Coq library. The CFML tool transforms a piece of OCaml code into a *characteristic formula*, a Coq formula that describes the semantics of the code. The characteristic formula is then exploited, inside Coq, to state that the code satisfies a certain specification (a Separation Logic triple) and to interactively prove this statement. The CFML library provides a set of Coq tactics that implement the reasoning rules of Separation Logic.

In prior work [11], the second and third authors extend CFML with time credits [2,22] and use it to simultaneously verify the functional correctness and the (amortized) time complexity of OCaml code. To illustrate the style in which they write specifications, consider a function that computes the length of a list:

```
let rec length l =
    match l with
    | []      -> 0
    | _ :: l -> 1 + length l
```

About this function, one can prove the following statement:

$$\forall (A : \mathsf{Type})(l : \mathsf{list}\, A).\ \{\, \$(|l|+1)\,\}\,(\texttt{length}\ l)\,\{\lambda y.\,[\,y = |l|\,]\}$$

This is a Separation Logic triple $\{H\}\,(t)\,\{Q\}$. The postcondition $\lambda y.\,[\,y = |l|\,]$ asserts that the call `length` l returns the length of the list l.[3] The precondition $\$(|l|+1)$ asserts that this call requires $|l|+1$ credits. This triple is proved in a variant of Separation Logic where every function call and every loop iteration consumes one credit. Thus, the above specification guarantees that the execution of `length` l involves no more than $|l|+1$ function calls or loop iterations. Our previous paper [11, Definition 2] gives a precise definition of the meaning of triples.

As argued in prior work [11, Sect. 2.7], bounding the number of function calls and loop iterations is equivalent, up to a constant factor, to bounding the number of reduction steps of the program. Assuming that the OCaml compiler is complexity-preserving, this is equivalent, up to a constant factor, to bounding the number of instructions executed by the compiled code. Finally, assuming that the machine executes one instruction in bounded time, this is equivalent, up to a constant factor, to bounding the execution time of the compiled code. Thus, the above specification guarantees that `length` runs in linear time.

Instead of understanding Separation Logic with Time Credits as a variant of Separation Logic, one can equivalently view it as standard Separation Logic, applied to an instrumented program, where a `pay()` instruction has been inserted at the beginning of every function body and loop body. The proof of the program is carried out under the axiom $\{\$1\}\,(\texttt{pay()})\,\{\lambda\_.\top\}$, which imposes the consumption of one time credit at every `pay()` instruction. This instruction has no runtime effect: it is just a way of marking where credits must be consumed.

For example, the OCaml function `length` is instrumented as follows:

```
let rec length l =
    pay();
    match l with [] -> 0 | _ :: l -> 1 + length l
```

Executing "`length` l" involves executing `pay()` exactly $|l|+1$ times. For this reason, a valid specification of this instrumented code in ordinary Separation Logic must require at least $|l|+1$ credits in its precondition.

4.2 A Modularity Challenge

The above specification of `length` guarantees that `length` runs in linear time, but does not allow predicting how much real time is consumed by a call to `length`. Thus, this specification is already rather abstract. Yet, it is still too precise. Indeed, we believe that it would not be wise for a list library to publish a specification of `length` whose precondition requires exactly $|l|+1$ credits. Indeed, there are implementations of `length` that do not meet this specification. For example, the tail-recursive implementation found in the OCaml standard library, which in practice is more efficient than the naïve implementation shown

[3] The square brackets denote a pure Separation Logic assertion. $|l|$ denotes the length of the Coq list l. CFML transparently reflects OCaml integers as Coq relative integers and OCaml lists as Coq lists.

above, involves exactly $|l|+2$ function calls, therefore requires $|l|+2$ credits. By advertising a specification where $|l|+1$ credits suffice, one makes too strong a guarantee, and rules out the more efficient implementation.

After initially publishing a specification that requires $\$(|l|+1)$, one could of course still switch to the more efficient implementation and update the published specification so as to require $\$(|l|+2)$ instead of $\$(|l|+1)$. However, that would in turn require updating the specification and proof of every (direct and indirect) client of the list library, which is intolerable.

To leave some slack, one should publish a more abstract specification. For example, one could advertise that the cost of `length` l is an affine function of the length of the list l, that is, the cost is $a \cdot |l| + b$, for some constants a and b:

$$\exists(a, b : \mathbb{Z}).\ \forall(A : \mathsf{Type})(l : \mathsf{list}\ A).\ \ \{\$(a \cdot |l| + b)\}\ (\texttt{length}\ l)\ \{\lambda y.\, [\, y = |l|\,]\}$$

This is a better specification, in the sense that it is more modular. The naïve implementation of `length` shown earlier and the efficient implementation in OCaml's standard library both satisfy this specification, so one is free to choose one or the other, without any impact on the clients of the list library. In fact, any reasonable implementation of `length` should have linear time complexity and therefore should satisfy this specification.

That said, the style in which the above specification is written is arguably slightly too low-level. Instead of directly expressing the idea that the cost of `length` l is $O(|l|)$, we have written this cost under the form $a \cdot |l| + b$. It is preferable to state at a more abstract level that *cost* is dominated by $\lambda n.n$: such a style is more readable and scales to situations where multiple parameters and nonstandard filters are involved. Thus, we propose the following statement:

$$\exists \mathit{cost} : \mathbb{Z} \to \mathbb{Z}.\ \begin{cases} \mathit{cost} \preceq_{\mathbb{Z}} \lambda n.\, n \\ \forall(A : \mathsf{Type})(l : \mathsf{list}\ A).\ \ \{\$\mathit{cost}(|l|)\}\ (\texttt{length}\ l)\ \{\lambda y.\, [\, y = |l|\,]\} \end{cases}$$

Thereafter, we refer to the function *cost* as the *concrete cost* of `length`, as opposed to the *asymptotic bound*, represented here by the function $\lambda n.\, n$. This specification asserts that there exists a concrete cost function *cost*, which is dominated by $\lambda n.\, n$, such that $\mathit{cost}(|l|)$ credits suffice to justify the execution of `length` l. Thus, $\mathit{cost}(|l|)$ is an upper bound on the actual number of `pay()` instructions that are executed at runtime.

The above specification informally means that `length` l has time complexity $O(n)$ where the parameter n represents $|l|$, that is, the length of the list l. The fact that n represents $|l|$ is expressed by applying *cost* to $|l|$ in the precondition. The fact that this analysis is valid when n grows large enough is expressed by using the standard filter on $\mathbb{Z}$ in the assertion $\mathit{cost} \preceq_{\mathbb{Z}} \lambda n.\, n$.

In general, it is up to the user to choose what the parameters of the cost analysis should be, what these parameters represent, and which filter on these parameters should be used. The example of the Bellman-Ford algorithm (Sect. 6) illustrates this.

```
Record specO (A : filterType) (le : A → A → Prop)
             (bound : A → Z) (P : (A → Z) → Prop)
  := { cost : A → Z;
       cost_spec : P cost;
       cost_dominated : dominated A cost bound;
       cost_nonneg : ∀x, 0 ≤ cost x;
       cost_monotonic : monotonic le Z.le cost; }.
```

Fig. 5. Definition of `specO`.

4.3 A Record for Specifications

The specifications presented in the previous section share a common structure. We define a record type that captures this common structure, so as to make specifications more concise and more recognizable, and so as to help users adhere to this specification pattern.

This type, `specO`, is defined in Fig. 5. The first three fields in this record type correspond to what has been explained so far. The first field asserts the existence of a function `cost` of A to $\mathbb{Z}$, where A is a user-specified filtered type. The second field asserts that a certain property `P cost` is satisfied; it is typically a Separation Logic triple whose precondition refers to `cost`. The third field asserts that `cost` is dominated by the user-specified function `bound`. The need for the last two fields is explained further on (Sects. 4.4 and 4.5).

Using this definition, our proposed specification of `length` (Sect. 4.2) is stated in concrete Coq syntax as follows:

```
Theorem length_spec:
  specO Z_filterType Z.le (fun n ⇒ n) (fun cost ⇒
    ∀A (l:list A), triple (length l)
      PRE ($ (cost |l|))
      POST (fun y ⇒ [ y = |l| ]))
```

The key elements of this specification are `Z_filterType`, which is $\mathbb{Z}$, equipped with its standard filter; the asymptotic bound `fun n ⇒ n`, which means that the time complexity of `length` is $O(n)$; and the Separation Logic triple, which describes the behavior of `length`, and refers to the concrete cost function `cost`.

One key technical point is that `specO` is a strong existential, whose witness can be referred to via to the first projection, `cost`. For instance, the concrete cost function associated with `length` can be referred to as `cost length_spec`. Thus, at a call site of the form `length xs`, the number of required credits is `cost length_spec |xs|`.

In the next subsections, we explain why, in the definition of `specO`, we require the concrete cost function to be nonnegative and monotonic. These are design decisions; although these properties may not be strictly necessary, we find that enforcing them greatly simplifies things in practice.

4.4 Why Cost Functions Must Be Nonnegative

There are several common occasions where one is faced with the obligation of proving that a cost expression is nonnegative. These proof obligations arise from several sources.

One source is the Separation Logic axiom for splitting credits, whose statement is $\$(m+n) = \$m \star \$n$, subject to the side conditions $m \geq 0$ and $n \geq 0$. Without these side conditions, out of \$0, one would be able to create $\$1 \star \$(-1)$. Because our logic is affine, one could then discard $\$(-1)$, keeping just \$1. In short, an unrestricted splitting axiom would allow creating credits out of thin air.[4] Another source of proof obligations is the Summation lemma (Lemma 8), which requires the functions at hand to be (ultimately) nonnegative.

Now, suppose one is faced with the obligation of proving that the expression `cost length_spec |xs|` is nonnegative. Because `length_spec` is an existential package (a `specO` record), this is impossible, unless this information has been recorded up front within the record. This is the reason why the field `cost_nonneg` in Fig. 5 is needed.

For simplicity, we require cost functions to be nonnegative everywhere, as opposed to within a certain domain. This requirement is stronger than necessary, but simplifies things, and can easily be met in practice by wrapping cost functions within "$\max(0, -)$". Our Coq tactics automatically insert "$\max(0, -)$" wrappers where necessary, making this issue mostly transparent to the user. In the following, for brevity, we write c^+ for $\max(0, c)$, where $c \in \mathbb{Z}$.

4.5 Why Cost Functions Must Be Monotonic

One key reason why cost functions should be monotonic has to do with the "avoidance problem". When the cost of a code fragment depends on a local variable x, can this cost be reformulated (and possibly approximated) in such a way that the dependency is removed? Indeed, a cost expression that makes sense outside the scope of x is ultimately required.

The problematic cost expression is typically of the form $E[|x|]$, where $|x|$ represents some notion of the "size" of the data structure denoted by x, and E is an arithmetic context, that is, an arithmetic expression with a hole. Furthermore, an upper bound on $|x|$ is typically available. This upper bound can be exploited if the context E is monotonic, i.e., if $x \leq y$ implies $E[x] \leq E[y]$. Because the hole in E can appear as an actual argument to an abstract cost function, we must record the fact that this cost function is monotonic.

To illustrate the problem, consider the following OCaml function, which counts the positive elements in a list of integers. It does so, in linear time, by first building a sublist of the positive elements, then computing the length of this sublist.

[4] Another approach would be to define $\$n$ only for $n \in \mathbb{N}$, in which case an unrestricted axiom would be sound. However, as we use $\mathbb{Z}$ everywhere, that would be inconvenient. A more promising idea is to view $\$n$ as linear (as opposed to affine) when n is negative. Then, $\$(-1)$ cannot be discarded, so unrestricted splitting is sound.

```
let count_pos l =
    let l' = List.filter (fun x -> x > 0) l in
    List.length l'
```

How would one go about proving that this code actually has linear time complexity? On paper, one would informally argue that the cost of the sequence `pay(); filter; length` is $O(1) + O(|l|) + O(|l'|)$, then exploit the inequality $|l'| \leq |l|$, which follows from the semantics of `filter`, and deduce that the cost is $O(|l|)$.

In a formal setting, though, the problem is not so simple. Assume that we have two specification lemmas `length_spec` and `filter_spec` for `List.length` and `List.filter`, which describe the behavior of these OCaml functions and guarantee that they have linear-time complexity. For brevity, let us write just g and f for the functions `cost length_spec` and `cost filter_spec`. Also, at the mathematical level, let us write $l{\downarrow}$ for the sublist of the positive elements of the list l. It is easy enough to check that the cost of the expression "`pay(); let l' = ... in List.length l'`" is $1 + f(|l|) + g(|l'|)$. The problem, now, is to *find an upper bound* for this cost *that does not depend on* l', a local variable, and to verify that this upper bound, *expressed as a function of* $|l|$, is dominated by $\lambda n.\, n$. Indeed, this is required in order to establish a `spec0` statement about `count_pos`.

What might this upper bound be? That is, which functions *cost* of $\mathbb{Z}$ to $\mathbb{Z}$ are such that (A) $1 + f(|l|) + g(|l'|) \leq \mathit{cost}(|l|)$ can be proved (in the scope of the local variable l') and (B) $\mathit{cost} \preceq_{\mathbb{Z}} \lambda n.\, n$ holds? Three potential answers come to mind:

1. Within the scope of l', the equality $l' = l{\downarrow}$ is available, as it follows from the postcondition of `filter`. Thus, within this scope, $1 + f(|l|) + g(|l'|)$ is provably equal to *let* $l' = l{\downarrow}$ *in* $1 + f(|l|) + g(|l'|)$, that is, $1 + f(|l|) + g(|l{\downarrow}|)$. This remark may seem promising, as this cost expression does not depend on l'. Unfortunately, this approach falls short, because this cost expression cannot be expressed as the application of a closed function *cost* to $|l|$. Indeed, the length of the filtered list, $|l{\downarrow}|$, is not a function of the length of l. In short, substituting local variables away in a cost expression does not always lead to a usable cost function.
2. Within the scope of l', the inequality $|l'| \leq |l|$ is available, as it follows from $l' = l{\downarrow}$. Thus, inequality (A) can be proved, provided we take:

$$\mathit{cost} = \lambda n.\ \max_{0 \leq n' \leq n}\ 1 + f(n) + g(n')$$

Furthermore, for this definition of *cost*, the domination assertion (B) holds as well. The proof relies on the fact the functions g and $\hat{g}$, where $\hat{g}$ is $\lambda n.\ \max_{0 \leq n' \leq n}\ g(n')$ [19], dominate each other. Although this approach seems viable, and does not require the function g to be monotonic, it is a bit more complicated than we would like.

3. Let us now assume that the function g is monotonic, that is, nondecreasing. As before, within the scope of l', the inequality $|l'| \leq |l|$ is available. Thus, the cost expression $1 + f(|l|) + g(|l'|)$ is bounded by $1 + f(|l|) + g(|l|)$. Therefore, inequalities (A) and (B) are satisfied, provided we take:

$$cost = \lambda n.\, 1 + f(n) + g(n)$$

We believe that approach 3 is the simplest and most intuitive, because it allows us to easily eliminate l', without giving rise to a complicated cost function, and without the need for a running maximum.

However, this approach requires that the cost function g, which is short for `cost length_spec`, be monotonic. This explains why we build a monotonicity condition in the definition of `specO` (Fig. 5, last line). Another motivation for doing so is the fact that some lemmas (such as Lemma 8, which allows reasoning about the asymptotic cost of an inner loop) also have monotonicity hypotheses.

The reader may be worried that, in practice, there might exist concrete cost functions that are not monotonic. This may be the case, in particular, of a cost function f that is obtained as the solution of a recurrence equation. Fortunately, in the common case of functions of $\mathbb{Z}$ to $\mathbb{Z}$, the "running maximum" function $\hat{f}$ can always be used in place of f: indeed, it is monotonic and has the same asymptotic behavior as f. Thus, we see that both approaches 2 and 3 above involve running maxima in some places, but their use seems less frequent with approach 3.

5 Interactive Proofs of Asymptotic Complexity Claims

To prove a specification lemma, such as `length_spec` (Sect. 4.3) or `loop_spec` (Sect. 4.4), one must construct a `specO` record. By definition of `specO` (Fig. 5), this means that one must exhibit a concrete cost function *cost* and prove a number of properties of this function, including the fact that, when supplied with $(*cost* \ldots)$, the code runs correctly (`cost_spec`) and the fact that *cost* is dominated by the desired asymptotic bound (`cost_dominated`).

Thus, the very first step in a naïve proof attempt would be to *guess* an appropriate cost function for the code at hand. However, such an approach would be painful, error-prone, and brittle. It seems much preferable, if possible, to enlist the machine's help in *synthesizing* a cost function *at the same time as we step through the code*—which we have to do anyway, as we must build a Separation Logic proof of the correctness of this code.

To illustrate the problem, consider the recursive function `p`, whose integer argument `n` is expected to satisfy $n \geq 0$. For the sake of this example, `p` calls an auxiliary function `g`, which we assume runs in constant time.

```
let rec p n =
    if n <= 1 then () else begin g(); p(n-1) end
```

Suppose we wish to establish that `p` runs in linear time. As argued at the beginning of the paper (Sect. 2, Fig. 2), it does not make sense to attempt a proof

by induction on n that "`p n` runs in time $O(n)$". Instead, in a formal framework, we must exhibit a concrete cost function *cost* such that $cost(n)$ credits justify the call `p n` and *cost* grows linearly, that is, $cost \preceq_{\mathbb{Z}} \lambda n.\, n$.

Let us assume that a specification lemma `g_spec` for the function `g` has been established already, so the number of credits required by a call to `g` is `cost g_spec ()`. In the following, we write G as a shorthand for this constant.

Because this example is very simple, it is reasonably easy to manually come up with an appropriate cost function for `p`. One valid guess is $\lambda n.\ 1+\Sigma_{i=2}^{n}(1+G)$. Another valid guess, obtained via a simplification step, is $\lambda n.\ 1+(1+G)(n-1)^{+}$. Another witness, obtained via an approximation step, is $\lambda n.\ 1 + (1 + G)n^{+}$. As the reader can see, there is in fact a spectrum of valid witnesses, ranging from verbose, low-level to compact, high-level mathematical expressions. Also, it should be evident that, as the code grows larger, it can become very difficult to guess a valid concrete cost function.

This gives rise to two questions. Among the valid cost functions, which one is preferable? Which ones can be systematically constructed, without guessing?

Among the valid cost functions, there is a tradeoff. At one extreme, a low-level cost function has exactly the same syntactic structure as the code, so it is easy to prove that it is an upper bound for the actual cost of the code, but a lot of work may be involved in proving that it is dominated by the desired asymptotic bound. At the other extreme, a high-level cost function can be essentially identical to the desired asymptotic bound, up to explicit multiplicative and additive constants, so the desired domination assertion is trivial, but a lot of accounting work may be involved in proving that this function represents enough credits to execute the code. Thus, by choosing a cost function, we shift some of the burden of the proof from one subgoal to another. From this point of view, no cost function seems inherently preferable to another.

From the point of view of systematic construction, however, the answer is more clear-cut. It seems fairly clear that it is possible to systematically build a cost function whose syntactic structure is the same as the syntactic structure of the code. This idea goes at least as far back as Wegbreit's work [26]. Coming up with a compact, high-level expression of the cost, on the other hand, seems to require human insight.

To provide as much machine assistance as possible, our system mechanically synthesizes a low-level cost expression for a piece of OCaml code. This is done transparently, at the same time as the user constructs a proof of the code in Separation Logic. Furthermore, we take advantage of the fact that we are using an interactive proof assistant: we allow the user to guide the synthesis process. For instance, the user controls how a local variable should be eliminated, how the cost of a conditional construct should be approximated (i.e., by a conditional or by a maximum), and how recurrence equations should be solved. In the following, we present this semi-interactive synthesis process. We first consider straight-line (nonrecursive) code (Sect. 5.1), then recursive functions (Sect. 5.2).

5.1 Synthesizing Cost Expressions for Straight-Line Code

The CFML library provides the user with interactive tactics that implement the reasoning rules of Separation Logic. We set things up in such a way that, as these rules are applied, a cost expression is automatically synthesized.

$$\textsc{WeakenCost}\ \frac{\{\$\,c_2^+ \star H\}\,(e)\,\{Q\} \qquad c_2^+ \le c_1}{\{\$\,c_1 \star H\}\,(e)\,\{Q\}} \qquad \textsc{Seq}\ \frac{\{\$\,c_1^+ \star H\}\,(e_1)\,\{Q'\} \qquad \{\$\,c_2^+ \star Q'()\}\,(e_2)\,\{Q\}}{\{\$\,(c_1^+ + c_2^+)^+ \star H\}\,(e_1; e_2)\,\{Q\}}$$

$$\textsc{Let}\ \frac{\{\$\,c_1^+ \star H\}\,(e_1)\,\{Q'\} \qquad \forall x.\ \{\$\,c_2^+ \star Q'(x)\}\,(e_2)\,\{Q\}}{\{\$\,(c_1^+ + c_2^+)^+ \star H\}\,(\text{let } x = e_1 \text{ in } e_2)\,\{Q\}} \qquad \textsc{Val}\ \frac{H \Vdash Q(v)}{\{\$\,0^+ \star H\}\,(v)\,\{Q\}}$$

$$\textsc{If}\ \frac{\begin{array}{c} b = \text{true} \Rightarrow \{\$\,c_1^+ \star H\}\,(e_1)\,\{Q\} \\ b = \text{false} \Rightarrow \{\$\,c_2^+ \star H\}\,(e_2)\,\{Q\} \end{array}}{\{\$\,(\mathit{if}\ b\ \mathit{then}\ c_1\ \mathit{else}\ c_2)^+ \star H\}\,(\text{if } b \text{ then } e_1 \text{ else } e_2)\,\{Q\}} \qquad \textsc{Pay}\ \frac{H \Vdash Q()}{\{\$\,1^+ \star H\}\,(\text{pay}())\,\{Q\}}$$

$$\textsc{For}\ \frac{\forall i.\ a \le i < b\ \Rightarrow\ \{\$\,c(i)^+ \star I(i)\}\,(e)\,\{I(i+1)\} \qquad H \Vdash I(a) \star Q}{\{\$\,(\Sigma_{a \le i < b}\ c(i)^+)^+ \star H\}\,(\text{for } i = a \text{ to } b - 1 \text{ do } e \text{ done})\,\{I(b) \star Q\}}$$

Fig. 6. The reasoning rules of Separation Logic, specialized for cost synthesis.

To this end, we use specialized variants of the reasoning rules, whose premises and conclusions take the form $\{\$\,n \star H\}\,(e)\,\{Q\}$. Furthermore, to simplify the nonnegativeness side conditions that must be proved while reasoning, we make all cost expressions obviously nonnegative by wrapping them in $\max(0, -)$. Recall that c^+ stands for $\max(0, c)$, where $c \in \mathbb{Z}$. Our reasoning rules work with triples of the form $\{\$\,c^+ \star H\}\,(e)\,\{Q\}$. They are shown in Fig. 6.

Because we wish to *synthesize* a cost expression, our Coq tactics maintain the following invariant: whenever the goal is $\{\$\,c^+ \star H\}\,(e)\,\{Q\}$, the cost c is *uninstantiated*, that is, it is represented in Coq by a metavariable, a placeholder. This metavariable is instantiated when the goal is proved by applying one of the reasoning rules. Such an application produces new subgoals, whose preconditions contain new metavariables. As this process is repeated, a cost expression is incrementally constructed.

The rule WeakenCost is a special case of the consequence rule of Separation Logic. It is typically used once at the root of the proof: even though the initial goal $\{\$\,c_1 \star H\}\,(e)\,\{Q\}$ may not satisfy our invariant, because it lacks a $-^+$ wrapper and because c_1 is not necessarily a metavariable, WeakenCost gives rise to a subgoal $\{\$\,c_2^+ \star H\}\,(e)\,\{Q\}$ that satisfies it. Indeed, when this rule is applied, a fresh metavariable c_2 is generated. WeakenCost can also be explicitly applied by the user when desired. It is typically used just before leaving the scope

of a local variable x to approximate a cost expression c_2^+ that depends on x with an expression c_1 that does not refer to x.

The Seq rule is a special case of the Let rule. It states that the cost of a sequence is the sum of the costs of its subexpressions. When this rule is applied to a goal of the form $\{\$\, c^+ \star H\}\,(e)\,\{Q\}$, where c is a metavariable, two new metavariables c_1 and c_2 are introduced, and c is instantiated with $c_1^+ + c_2^+$.

The Let rule is similar to Seq, but involves an additional subtlety: the cost c_2 must not refer to the local variable x. Naturally, Coq enforces this condition: any attempt to instantiate the metavariable c_2 with an expression where x occurs fails. In such a situation, it is up to the user to use WeakenCost so as to avoid this dependency. The example of `count_pos` (Sect. 4.5) illustrates this issue.

The Val rule handles values, which in our model have zero cost. The symbol $\Vdash$ denotes entailment between Separation Logic assertions.

The If rule states that the cost of an OCaml conditional expression is a mathematical conditional expression. Although this may seem obvious, one subtlety lurks here. Using WeakenCost, the cost expression *if* b *then* c_1 *else* c_2 can be approximated by $\max(c_1, c_2)$. Such an approximation can be beneficial, as it leads to a simpler cost expression, or harmful, as it causes a loss of information. In particular, when carried out in the body of a recursive function, it can lead to an unsatisfiable recurrence equation. We let the user decide whether this approximation should be performed.

The Pay rule handles the `pay()` instruction, which is inserted by the CFML tool at the beginning of every function and loop body (Sect. 4.1). This instruction costs one credit.

The For rule states that the cost of a `for` loop is the sum, over all values of the index i, of the cost of the i-th iteration of the body. In practice, it is typically used in conjunction with WeakenCost, which allows the user to simplify and approximate the iterated sum $\Sigma_{a \le i < b}\ c(i)^+$. In particular, if the synthesized cost $c(i)$ happens to not depend on i, or can be approximated so as to not depend on i, then this iterated sum can be expressed under the form $c(b-a)^+$. A variant of the For rule, not shown, covers this common case. There is in principle no need for a primitive treatment of loops, as loops can be encoded in terms of higher-order recursive functions, and our program logic can express the specifications of these combinators. Nevertheless, in practice, primitive support for loops is convenient.

This concludes our exposition of the reasoning rules of Fig. 6. Coming back to the example of the OCaml function `p` (Sect. 5), under the assumption that the cost of the recursive call `p(n-1)` is $f(n-1)$, we are able, by repeated application of the reasoning rules, to automatically find that the cost of the OCaml expression:

```
if n <= 1 then () else begin g(); p(n-1) end
```

is: $1 + \mathit{if}\ n \le 1\ \mathit{then}\ 0\ \mathit{else}\ (G + f(n-1))$. The initial 1 accounts for the implicit `pay()`. This may seem obvious, and it is. The point is that this cost expression is automatically constructed: its synthesis adds no overhead to an interactive proof of functional correctness of the function `p`.

5.2 Synthesizing and Solving Recurrence Equations

There now remains to explain how to deal with recursive functions. Suppose $S(f)$ is the Separation Logic triple that we wish to establish, where f stands for an as-yet-unknown cost function. Following common informal practice, we would like to do this in two steps. First, from the code, derive a "recurrence equation" $E(f)$, which in fact is usually not an equation, but a constraint (or a conjunction of constraints) bearing on f. Second, prove that this recurrence equation admits a solution that is dominated by the desired asymptotic cost function g. This approach can be formally viewed as an application of the following tautology:

$$\forall E.\ (\forall f.E(f) \to S(f)) \ \to\ (\exists f.E(f) \wedge f \preceq g) \ \to\ (\exists f.S(f) \wedge f \preceq g)$$

The conclusion $S(f) \wedge f \preceq g$ states that the code is correct and has asymptotic cost g. In Coq, applying this tautology gives rise to a new metavariable E, as the recurrence equation is initially unknown, and two subgoals.

During the proof of the first subgoal, $\forall f.E(f) \to S(f)$, the cost function f is abstract (universally quantified), but we are allowed to assume $E(f)$, where E is initially a metavariable. So, should the need arise to prove that f satisfies a certain property, this can be done just by instantiating E. In the example of the OCaml function `p` (Sect. 5), we prove $S(f)$ by induction over n, under the hypothesis $n \geq 0$. Thus, we assume that the cost of the recursive call `p(n-1)` is $f(n-1)$, and must prove that the cost of `p n` is $f(n)$. We synthesize the cost of `p n` as explained earlier (Sect. 5.1) and find that this cost is $1 + \mathit{if}\ n \leq 1\ \mathit{then}\ 0\ \mathit{else}\ (G + f(n-1))$. We apply WeakenCost and find that our proof is complete, provided we are able to prove the following inequation:

$$1 + \mathit{if}\ n \leq 1\ \mathit{then}\ 0\ \mathit{else}\ (G + f(n-1)) \ \leq\ f(n)$$

We achieve that simply by instantiating E as follows:

$$E := \lambda f.\ \forall n.\ n \geq 0 \ \to\ 1 + \mathit{if}\ n \leq 1\ \mathit{then}\ 0\ \mathit{else}\ (G + f(n-1)) \ \leq\ f(n)$$

This is our "recurrence equation"—in fact, a universally quantified, conditional inequation. We are done with the first subgoal.

We then turn to the second subgoal, $\exists f.E(f) \wedge f \preceq g$. The metavariable E is now instantiated. The goal is to solve the recurrence and analyze the asymptotic growth of the chosen solution. There are at least three approaches to solving such a recurrence.

First, one can guess a closed form that satisfies the recurrence. For example, the function $f := \lambda n.\ 1 + (1 + G)n^+$ satisfies $E(f)$ above. But, as argued earlier, guessing is in general difficult and tedious.

Second, one can invoke Cormen *et al.*'s Master Theorem [12] or the more general Akra-Bazzi theorem [1,21]. Unfortunately, at present, these theorems are not available in Coq, although an Isabelle/HOL formalization exists [13].

The last approach is Cormen *et al.*'s substitution method [12, Sect. 4]. The idea is to guess a parameterized *shape* for the solution; substitute this shape into

the goal; gather a set of constraints that the parameters must satisfy for the goal to hold; finally, show that these constraints are indeed satisfiable. In the above example, as we expect the code to have linear time complexity, we propose that the solution f should have the shape $\lambda n.(an^+ + b)$, where a and b are parameters, about which we wish to gradually accumulate a set of constraints. From a formal point of view, this amounts to applying the following tautology:

$$\forall P.\, \forall C.\quad (\forall ab.\, C(a,b) \to P(\lambda n.(an^+ + b))) \;\to\; (\exists ab.\, C(a,b)) \;\to\; \exists f.P(f)$$

This application again yields two subgoals. During the proof of the first subgoal, C is a metavariable and can be instantiated as desired (possibly in several steps), allowing us to gather a conjunction of constraints bearing on a and b. During the proof of the second subgoal, C is fixed and we must check that it is satisfiable. In our example, the first subgoal is:

$$E(\lambda n.(an^+ + b)) \quad\wedge\quad \lambda n.(an^+ + b) \preceq_{\mathbb{Z}} \lambda n.n$$

The second conjunct is trivial. The first conjunct simplifies to:

$$\forall n.\quad n \geq 0 \;\to\; 1 + \mathit{if}\ n \leq 1\ \mathit{then}\ 0\ \mathit{else}\ (G + a(n-1)^+ + b) \;\leq\; an^+ + b$$

By distinguishing the cases $n = 0$, $n = 1$, and $n > 1$, we find that this property holds provided we have $1 \leq b$ and $1 \leq a + b$ and $1 + G \leq a$. Thus, we prove this subgoal by instantiating C with $\lambda(a,b).(1 \leq b \wedge 1 \leq a + b \wedge 1 + G \leq a)$.

There remains to check the second subgoal, that is, $\exists ab.C(a,b)$. This is easy; we pick, for instance, $a := 1 + G$ and $b := 1$. This concludes our use of Cormen *et al.*'s substitution method.

In summary, by exploiting Coq's metavariables, we are able to set up our proofs in a style that closely follows the traditional paper style. During a first phase, as we analyze the code, we synthesize a cost function and (if the code is recursive) a recurrence equation. During a second phase, we guess the shape of a solution, and, as we analyze the recurrence equation, we synthesize a constraint on the parameters of the shape. During a last phase, we check that this constraint is satisfiable. In practice, instead of explicitly building and applying tautologies as above, we use the first author's `procrastination` library [16], which provides facilities for introducing new parameters, gradually gathering constraints on these parameters, and eventually checking that these constraints are satisfiable.

6 Examples

Binary Search. We prove that binary search has time complexity $O(\log n)$, where $n = j - i$ denotes the width of the search interval $[i, j)$. The code is as in Fig. 1, except that the flaw is fixed by replacing `i+1` with `k+1` on the last line. As outlined earlier (Sect. 5), we synthesize the following recurrence equation on the cost function f:

$$f(0) + 3 \leq f(1) \quad\wedge\quad \forall n \geq 0.\ 1 \leq f(n) \quad\wedge\quad \forall n \geq 2.\ f(n/2) + 3 \leq f(n)$$

We apply the substitution method and search for a solution of the form $\lambda n.\ \mathit{if}\ n \leq 0\ \mathit{then}\ 1\ \mathit{else}\ a \log n + b$, which is dominated by $\lambda n. \log n$. Substituting this shape into the above constraints, we find that they boil down to $(4 \leq b) \wedge (0 \leq a \wedge 1 \leq b) \wedge (3 \leq a)$. Finally, we guess a solution, namely $a := 3$ and $b := 4$.

Dependent Nested Loops. Many algorithms involve dependent nested `for` loops, that is, nested loops, where the bounds of the inner loop depend on the outer loop index, as in the following simplified example:

```
for i = 1 to n do
    for j = 1 to i do () done
  done
```

For this code, the cost function $\lambda n.\ \sum_{i=1}^{n}(1 + \sum_{j=1}^{i} 1)$ is synthesized. There remains to prove that it is dominated by $\lambda n.n^2$. We could recognize and prove that this function is equal to $\lambda n.\frac{n(n+3)}{2}$, which clearly is dominated by $\lambda n.n^2$. This works because this example is trivial, but, in general, computing explicit closed forms for summations is challenging, if at all feasible.

A higher-level approach is to exploit the fact that, if f is monotonic, then $\sum_{i=1}^{n} f(i)$ is less than $n.f(n)$. Applying this lemma twice, we find that the above cost function is less than $\lambda n. \sum_{i=1}^{n}(1+i)$ which is less than $\lambda n.n(1+n)$ which is dominated by $\lambda n.n^2$. This simple-minded approach, which does not require the Summation lemma (Lemma 8), is often applicable. The next example illustrates a situation where the Summation lemma is required.

A Loop Whose Body Has Exponential Cost. In the following simple example, the loop body is just a function call:

```
for i = 0 to n-1 do b(i) done
```

Thus, the cost of the loop body is not known exactly. Instead, let us assume that a specification for the auxiliary function `b` has been proved and that its cost is $O(2^i)$, that is, $\texttt{cost b} \preceq_{\mathbb{Z}} \lambda i.\ 2^i$ holds. We then wish to prove that the cost of the whole loop is also $O(2^n)$.

For this loop, the cost function $\lambda n.\ \sum_{i=0}^{n}(1 + \texttt{cost b}\ (i))$ is automatically synthesized. We have an asymptotic bound for the cost of the loop body, namely: $\lambda i.\ 1 + \texttt{cost b}\ (i) \preceq_{\mathbb{Z}} \lambda i.\ 2^i$. The side conditions of the Summation lemma (Lemma 8) are met: in particular, the function $\lambda i.\ 1 + \texttt{cost b}\ (i)$ is monotonic. The lemma yields $\lambda n.\ \sum_{i=0}^{n}(1 + \texttt{cost b}\ (i)) \preceq_{\mathbb{Z}} \lambda n.\ \sum_{i=0}^{n} 2^i$. Finally, we have $\lambda n.\ \sum_{i=0}^{n} 2^i = \lambda n.\ 2^{n+1} - 1 \preceq_{\mathbb{Z}} \lambda n.\ 2^n$.

The Bellman-Ford Algorithm. We verify the asymptotic complexity of an implementation of Bellman-Ford algorithm, which computes shortest paths in a weighted graph with n vertices and m edges. The algorithm involves an outer loop that is repeated $n-1$ times and an inner loop that iterates over all m edges. The specification asserts that the asymptotic complexity is $O(nm)$:

$$\exists \mathit{cost} : \mathbb{Z}^2 \to \mathbb{Z}.\ \begin{cases} \mathit{cost} \preceq_{\mathbb{Z}^2} \lambda(m,n).\, nm \\ \{\$\mathit{cost}(\#\mathit{edges}(g), \#\mathit{vertices}(g))\}\ (\texttt{bellmanford}\ g)\ \{\ldots\} \end{cases}$$

By exploiting the fact that a graph without duplicate edges must satisfy $m \leq n^2$, we prove that the complexity of the algorithm, viewed as a function of n, is $O(n^3)$.

$$\exists cost : \mathbb{Z} \to \mathbb{Z}. \begin{cases} cost \preceq_{\mathbb{Z}} \lambda n.\, n^3 \\ \{\$cost(\#vertices(g))\}\ (\texttt{bellmanford}\ g)\ \{\ldots\} \end{cases}$$

To prove that the former specification implies the latter, one instantiates m with n^2, that is, one exploits a composition lemma (Sect. 3.4). In practice, we publish both specifications and let clients use whichever one is more convenient.

Union-Find. Charguéraud and Pottier [11] use Separation Logic with Time Credits to verify the correctness and time complexity of a Union-Find implementation. For instance, they prove that the (amortized) concrete cost of `find` is $2\alpha(n) + 4$, where n is the number of elements. With a few lines of proof, we derive a specification where the cost of `find` is expressed under the form $O(\alpha(n))$:

```
specO Z_filterType Z.le (fun n ⇒ alpha n) (fun cost ⇒
      ∀D R V x, x \in D → triple (UnionFind_ml.find x)
        PRE (UF D R V ⋆ $(cost (card D)))
        POST (fun y ⇒ UF D R V ⋆ [R x = y ])).
```

Union-Find is a mutable data structure, whose state is described by the abstract predicate `UF D R V`. In particular, the parameter `D` represents the domain of the data structure, that is, the set of all elements created so far. Thus, its cardinal, `card D`, corresponds to n. This case study illustrates a situation where the cost of an operation depends on the current state of a mutable data structure.

7 Related Work

Our work builds on top of Separation Logic [23] with Time Credits [2], which has been first implemented in a verification tool and exploited by the second and third authors [11]. We refer the reader to their paper for a survey of the related work in the general area of formal reasoning about program complexity, including approaches based on deductive program verification and approaches based on automatic complexity analysis. In this section, we restrict our attention to informal and formal treatments of the O notation.

The O notation and its siblings are documented in several textbooks [7,15,20]. Out of these, only Howell [19,20] draws attention to the subtleties of the multivariate case. He shows that one cannot take for granted that the properties of the O notation, which in the univariate case are well-known, remain valid in the multivariate case. He states several properties which, at first sight, seem natural and desirable, then proceeds to show that they are inconsistent, so no definition of the O notation can satisfy them all. He then proposes a candidate notion of domination between functions whose domain is $\mathbb{N}^k$. His notation, $f \in \hat{O}(g)$, is defined as the conjunction of $f \in O(g)$ and $\hat{f} \in O(\hat{g})$, where the function $\hat{f}$ is a "running

maximum" of the function f, and is by construction monotonic. He shows that this notion satisfies all the desired properties, provided some of them are restricted by additional side conditions, such as monotonicity requirements.

In this work, we go slightly further than Howell, in that we consider functions whose domain is an arbitrary filtered type A, rather than necessarily $\mathbb{N}^k$. We give a standard definition of O and verify all of Howell's properties, again restricted with certain side conditions. We find that we do not need $\hat{O}$, which is fortunate, as it seems difficult to define $\hat{f}$ in the general case where f is a function of domain A. The monotonicity requirements that we impose are not exactly the same as Howell's, but we believe that the details of these administrative conditions do not matter much, as all of the functions that we manipulate in practice are everywhere nonnegative and monotonic.

Avigad and Donnelly [3] formalize the O notation in Isabelle/HOL. They consider functions of type $A \to B$, where A is arbitrary and B is an ordered ring. Their definition of "$f = O(g)$" requires $|f(x)| \leq c|g(x)|$ for every x, as opposed to "when x is large enough". Thus, they get away without equipping the type A with a filter. The price to pay is an overly restrictive notion of domination, except in the case where A is $\mathbb{N}$, where both $\forall x$ and $\mathbb{U}x$ yield the same notion of domination—this is Brassard and Bratley's "threshold rule" [7]. Avigad and Donnelly suggest defining "$f = O(g)$ eventually" as an abbreviation for $\exists f', (f' = O(g) \wedge \mathbb{U}x.f(x) = f'(x))$. In our eyes, this is less elegant than parameterizing O with a filter in the first place.

Eberl [13] formalizes the Akra-Bazzi method [1,21], a generalization of the well-known Master Theorem [12], in Isabelle/HOL. He creates a library of Landau symbols specifically for this purpose. Although his paper does not mention filters, his library in fact relies on filters, whose definition appears in Isabelle's Complex library. Eberl's definition of the O symbol is identical to ours. That said, because he is concerned with functions of type $\mathbb{N} \to \mathbb{R}$ or $\mathbb{R} \to \mathbb{R}$, he does not define product filters, and does not prove any lemmas about domination in the multivariate case. Eberl sets up a decision procedure for domination goals, like $x \in O(x^3)$, as well as a procedure that can simplify, say, $O(x^3+x^2)$ to $O(x^3)$.

TiML [25] is a functional programming language where types carry time complexity annotations. Its type-checker generates proof obligations that are discharged by an SMT solver. The core type system, whose metatheory is formalized in Coq, employs concrete cost functions. The TiML implementation allows associating a O specification with each toplevel function. An unverified component recognizes certain classes of recurrence equations and automatically applies the Master Theorem. For instance, *mergesort* is recognized to be $O(mn \log n)$, where n is the input size and m is the cost of a comparison. The meaning of the O notation in the multivariate case is not spelled out; in particular, which filter is meant is not specified.

Boldo *et al.* [4] use Coq to verify the correctness of a C program which implements a numerical scheme for the resolution of the one-dimensional acoustic wave equation. They define an ad hoc notion of "uniform O" for functions of type $\mathbb{R}^2 \to \mathbb{R}$, which we believe can in fact be viewed as an instance of our

generic definition of domination, at an appropriate product filter. Subsequent work on the Coquelicot library for real analysis [5] includes general definitions of filters, limits, little-o and asymptotic equivalence. A few definitions and lemmas in Coquelicot are identical to ours, but the focus in Coquelicot is on various filters on $\mathbb{R}$, whereas we are more interested in filters on $\mathbb{Z}^k$.

The tools RAML [17] and Pastis [8] perform fully automated amortized time complexity analysis of OCaml programs. They can be understood in terms of Separation Logic with Time Credits, under the constraint that the number of credits that exist at each program point must be expressed as a polynomial over the variables in scope at this point. The a priori unknown coefficients of this polynomial are determined by an LP solver. Pastis produces a proof certificate that can be checked by Coq, so the trusted computing base of this approach is about the same as ours. RAML and Pastis offer much stronger automation than our approach, but have weaker expressive power. It would be very interesting to offer access to a Pastis-like automated system within our interactive system.

References

1. Akra, M.A., Bazzi, L.: On the solution of linear recurrence equations. Comput. Optim. Appl. **10**(2), 195–210 (1998). https://doi.org/10.1023/A:1018373005182
2. Atkey, R.: Amortised resource analysis with separation logic. Log. Methods Comput. Sci. **7**(2:17) (2011). http://bentnib.org/amortised-sep-logic-journal.pdf
3. Avigad, J., Donnelly, K.: Formalizing O notation in Isabelle/HOL. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 357–371. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25984-8_27
4. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: a comprehensive mechanized proof of a C program. J. Autom. Reason. **50**(4), 423–456 (2013). https://hal.inria.fr/hal-00649240
5. Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: a user-friendly library of real analysis for Coq. Math. Comput. Sci. **9**(1), 41–62 (2015). https://hal.inria.fr/hal-00860648
6. Bourbaki, N.: General Topology, Chapters 1–4. Springer, Heidelberg (1995). https://doi.org/10.1007/978-3-642-61701-0
7. Brassard, G., Bratley, P.: Fundamentals of Algorithmics. Prentice Hall, Upper Saddle River (1996)
8. Carbonneaux, Q., Hoffmann, J., Reps, T., Shao, Z.: Automated resource analysis with Coq proof objects. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017, Part II. LNCS, vol. 10427, pp. 64–85. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_4
9. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: International Conference on Functional Programming (ICFP), pp. 418–430, September 2011. http://www.charguer aud.org/research/2011/cfml/main.pdf
10. Charguéraud, A.: The CFML tool and library (2016). http://www.charguer aud.org/softs/cfml/
11. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. J. Autom. Reason. September 2017. http://gallium.inria.fr/~fpottier/publis/charguer aud-pottier-uf-sltc.pdf

12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009). http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11866
13. Eberl, M.: Proving divide and conquer complexities in Isabelle/HOL. J. Autom. Reason. **58**(4), 483–508 (2017). https://www21.in.tum.de/~Eeberlm/divide_and_conquer_isabelle.pdf
14. Filliâtre, J.-C., Letouzey, P.: Functors for proofs and programs. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 370–384. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24725-8_26
15. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics: A Foundation for Computer Science. Addison-Wesley (1994). http://www-cs-faculty.stanford.edu/~knuth/gkp.html
16. Guéneau, A., Charguéraud, A., Pottier, F.: Electronic appendix, January 2018. http://gallium.inria.fr/~agueneau/bigO/
17. Hoffmann, J., Das, A., Weng, S.: Towards automatic resource bound analysis for OCaml. In: Principles of Programming Languages (POPL), pp. 359–373, January 2017. http://www.cs.cmu.edu/~janh/papers/HoffmannDW17.pdf
18. Hopcroft, J.E.: Computer science: the emergence of a discipline. Commun. ACM **30**(3), 198–202 (1987). http://doi.acm.org/10.1145/214748.214750
19. Howell, R.R.: On asymptotic notation with multiple variables. Technical report 2007–4, Kansas State University, January 2008. http://people.cs.ksu.edu/~rhowell/asymptotic.pdf
20. Howell, R.R.: Algorithms: a top-down approach, July 2012, draft. http://people.cs.ksu.edu/~rhowell/algorithms-text/text/
21. Leighton, T.: Notes on better master theorems for divide-and-conquer recurrences (1996). http://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf
22. Pilkiewicz, A., Pottier, F.: The essence of monotonic state. In: Types in Language Design and Implementation (TLDI), January 2011. http://gallium.inria.fr/~fpottier/publis/pilkiewicz-pottier-monotonicity.pdf
23. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Logic in Computer Science (LICS), pp. 55–74 (2002). http://www.cs.cmu.edu/~jcr/seplogic.pdf
24. Tarjan, R.E.: Algorithm design. Commun. ACM **30**(3), 204–212 (1987). http://doi.acm.org/10.1145/214748.214752
25. Wang, P., Wang, D., Chlipala, A.: TiML: a functional language for practical complexity analysis with invariants. Proc. ACM Program. Lang. **1**(OOPSLA), 79:1–79:26 (2017). http://adam.chlipala.net/papers/TimlOOPSLA17/TimlOOPSLA17.pdf
26. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9), 528–539 (1975). http://doi.acm.org/10.1145/361002.361016

Verified Learning Without Regret

From Algorithmic Game Theory to Distributed Systems with Mechanized Complexity Guarantees

Samuel Merten(✉), Alexander Bagnall, and Gordon Stewart

Ohio University, Athens, OH, USA
{sm137907,ab667712,gstewart}@ohio.edu

Abstract. Multiplicative Weights (MW) is a simple yet powerful algorithm for learning linear classifiers, for ensemble learning à la boosting, for approximately solving linear and semidefinite systems, for computing approximate solutions to multicommodity flow problems, and for online convex optimization, among other applications. Recent work in algorithmic game theory, which applies a computational perspective to the design and analysis of systems with mutually competitive actors, has shown that no-regret algorithms like MW naturally drive games toward approximate Coarse Correlated Equilibria (CCEs), and that for certain games, approximate CCEs have bounded cost with respect to the optimal states of such systems.

In this paper, we put such results to practice by building distributed systems such as routers and load balancers with performance and convergence guarantees mechanically verified in Coq. The main contributions on which our results rest are (1) the first mechanically verified implementation of Multiplicative Weights (specifically, we show that our MW is no regret) and (2) a language-based formulation, in the form of a DSL, of the class of games satisfying Roughgarden smoothness, a broad characterization of those games whose approximate CCEs have cost bounded with respect to optimal. Composing (1) with (2) within Coq yields a new strategy for building distributed systems with mechanically verified complexity guarantees on the time to convergence to near-optimal system configurations.

Keywords: Multiplicative weights · Algorithmic game theory
Smooth games · Interactive theorem proving · Coq

1 Introduction

The Multiplicative Weights algorithm (MW, [1,25]) solves the general problem of "combining expert advice", in which an agent repeatedly chooses which action, or "expert", to play against an adaptive environment. The agent, after playing an action, learns from the environment both the cost of that action and of other actions it could have played in that round. The environment, in turn, may adapt

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 561–588, 2018.
https://doi.org/10.1007/978-3-319-89884-1_20

in order to minimize environment costs. MW works by maintaining a weighted distribution over the action space, in which each action initially has equal weight, and by updating weights with a linear or exponential loss function to penalize poorly performing actions.

MW is a *no-regret* algorithm: its expected cost approaches that of the best fixed action the agent could have chosen in hindsight (i.e., external regret tends to zero) as time $t \to \infty$. Moreover, this simple algorithm performs remarkably well: in number of rounds logarithmic in the size of the action space, MW's expected regret can be bounded by a small constant ϵ (MW has bounded external regret). In [1], Arora, Hazan, and Kale showed that MW has wide-ranging connections to numerous problems in computer science, including optimization, linear and semidefinite programming, and machine learning (cf. boosting [14]).

Our work targets another important application of MW: the approximate solution of multi-agent games, especially as such games relate to the construction of distributed systems. It is well known (cf. [30, Chapter 4]) that no-regret algorithms such as MW converge, when played by multiple independent agents, to a large equilibrium class known as Coarse Correlated Equilibria (CCEs). CCEs may not be socially optimal, but for some games, such as Roughgarden's smooth games [35], the social cost of such equilibrium states can be bounded by a constant factor of the optimal cost of the game (the game has bounded Price of Anarchy, or POA). Therefore, to drive the social cost of a smooth game to near optimal, it suffices simply to let each agent play a no-regret algorithm such as MW.

Moreover, a number of distributed systems can be encoded as games, especially when the task being distributed is viewed as an optimization problem. Consider, for example, distributed balancing of network flows over a set of web servers, an application we return to in Sect. 3. Assuming the set of flows is fixed, and that the cost of (or latency incurred by) assigning a flow to a particular web server increases as a function of the number of flows already assigned to that server (the traffic), then the load balancing application is encodable as a game in which each flow is a "player" attempting to optimize its cost (latency). An optimal solution of this game minimizes the total latency across all flows. Since the game is Roughgarden smooth (assuming affine cost functions), the social cost of its CCEs as induced by letting each player independently run MW is bounded with respect to that of an optimal solution.

1.1 Contributions

In this paper, we put such results to work by building the first verified implementation of the MW algorithm – which we use to drive all games to approximate CCEs – and by defining a language-based characterization of a subclass of games called Roughgarden smooth games that have robust Price of Anarchy guarantees extending even to approximate CCEs. Combining our verified MW with smooth games, we construct distributed systems for applications such as routing and load balancing that have verified convergence and correctness guarantees.

Specifically, our main contributions are:

- a new architecture, as embodied in the CAGE system (https://github.com/gstew5/cage), for the construction of distributed systems with verified complexity guarantees, by composition of verified Multiplicative Weights (MW) with robust Price of Anarchy bounds via Roughgarden smoothness;
- the first formally verified implementation of the MW algorithm;
- a language-based characterization of Roughgarden smooth games, in the form of a mechanized DSL for the construction of such games together with smoothness preservation theorems showing that each combinator in the language preserves smoothness;
- the application of the resulting system to distributed routing and load balancing.

By *verified*, we mean our MW implementation has mechanically checked convergence bounds and proof of correctness within an interactive theorem prover (specifically, Ssreflect [16], an extension of the Coq [5] system). By *convergence* and *correctness*, we mean that we prove both that MW produces the right answer (functional correctness with respect to a high-level functional specification), but also that it does so with external regret[1] bounded by a function of the number of iterations of the protocol (convergence). Convergence of MW in turn implies convergence to an approximate CCE. By composing this second convergence property with Roughgarden smoothness, we bound the social, or total, cost of the resulting system state with respect to the optimal.

As we've mentioned, MW has broad application across a number of subdisciplines of computer science, including linear programming, optimization, and machine learning. Although our focus in this paper is the use of MW to implement no-regret dynamics, a general strategy for computing the CCEs of multi-agent games, our implementation of MW (Sect. 5.3) could be used to build, e.g., a verified LP solver or verified implementation of boosting as well.

Limitations. The approach we outline above does not apply to all distributed systems, nor even to all distributed systems encodable as games. In particular, in order to prove POA guarantees in our approach, the game encoding a particular distributed system must first be shown Roughgarden smooth, a condition which does not always apply (e.g., to network formation games [35, Section 2]). More positively, the Smooth Games DSL we present in Sects. 3 and 4 provides one method by which to explore the combinatorial nature of Roughgarden smoothness, as we demonstrate with some examples in Sect. 3.

Relationship to Prior Work. Some of the ideas we present in this paper previously appeared in summary form in a 3-page brief announcement at PODC 2017 [4]. The current paper significantly expands on the architecture of the CAGE system, our verified implementation of Multiplicative Weights, the definition of the Smooth Games DSL, and the composition theorems of Sect. 6 proving that the pieces fit together to imply system-wide convergence and quality bounds.

[1] The expected (per-step) cost of the algorithm minus that of the best fixed action.

1.2 Organization

The following section provides background on games, algorithmic game theory, and smoothness. Section 3 presents an overview of the main components of the CAGE approach, via application to examples. Section 4 provides more detail on the combinators of our Smooth Games DSL. Section 5 presents our verified implementation of MW. Section 6 describes the composition theorems proving that multi-agent MW converges to near-optimal ϵ-CCEs. Sections 7 and 8 present related work and conclude.

2 Background

2.1 Games

Von Neumann, Morgenstern, and Nash [28,29] (in the US) and Bachelier, Borel, and Zermelo [3,8,43] (in Europe) were the first to study the mathematical theory of strategic interaction, modern game theory. Nash's famous result [27] showed that in all finite games, mixed-strategy equilibria (those in which players are allowed to randomize) always exist. Since the 1950s, game theory has had huge influence in numerous fields, especially economics.

In our context, a game is a tuple of a finite type A (the strategy space) and a cost function C_i mapping tuples of strategies of type $A_1 \times A_2 \times \ldots \times A_N$ to values of type $\mathbb{R}$, the cost to player i of state $(a_1, \ldots, a_i, \ldots, a_N)$. For readers interested in formalization-related aspects, Listing 1 provides additional details.

Listing 1: Games in Ssreflect-Coq

In SSREFLECT-COQ, an extension of the standard Coq system, a finite type A : finType pairs the type A with an enumerator enum : list A such that for all $a : A$, count a enum $= 1$ (every element is included exactly once). To define games, we use operational type classes [38], which facilitate parameter sharing:

```
Class game (A : finType) (N : nat) (ℝ : realFieldType)
  '(costClass : CostClass N ℝ A) : Type ≜ {}.
```

costClass declares the cost function C_i, and N is the number of players.

A state $s : A_1 \times A_2 \times \ldots \times A_N$ is a *Pure Nash Equilibrium (PNE)* when no player $i \in [1, N]$ has incentive to change its strategy: $\forall s'_i.\ C_i(s) \leq C_i(s'_i, s_{-i})$. Here s'_i is an arbitrary strategy. Strategy s_i is player i's move in state s. By s'_i, s_{-i} we denote the state in which player i's strategy is s'_i and all other players play s. In other words, no player can decrease its cost by unilateral deviation.

Pure-strategy Nash equilibria do not always exist. Mixed Nash Equilibria (MNE), which *do* exist in all finite games, permit players to randomize over the strategy space, by playing a distribution σ_i over A. The overall state is the product distribution over the player distributions. Every PNE is trivially an MNE, by letting players choose deterministic distributions σ_i.

Correlated Equilibria (CEs) generalize MNEs to situations in which players coordinate via a trusted third party. In what follows, we'll mostly be interested in a generalization of CEs, called *Coarse Correlated Equilibria (CCEs)*, and their approximate relaxations. Specifically, a distribution σ over A^N (Listing 2) is a CCE when $\forall i \forall s'_i.\ \mathbb{E}_{s\sim\sigma}[C_i(s)] \leq \mathbb{E}_{s\sim\sigma}[C_i(s'_i, s_{-i})]$. $\mathbb{E}_{s\sim\sigma}[C_i(s)]$ is the expected cost to player i in distribution σ. The CCE condition states that there is no s'_i that could decrease player i's expected cost. CCEs are essentially a relaxation of MNEs which do not require σ to be a product distribution (i.e., the players' strategies may be correlated). CEs are a subclass of CCEs in which $\mathbb{E}_{s\sim\sigma}[C_i(s'_i, s_{-i})]$ may be conditioned on s_i.

A distribution σ over states may only be *approximately* a CCE. Define as ϵ-approximate those CCEs σ for which $\forall i \forall s'.\ \mathbb{E}_{s\sim\sigma}[C_i(s)] \leq \mathbb{E}_{s\sim\sigma}[C_i(s'_i, s_{-i})] + \epsilon$. Moving to s'_i can decrease player i's expected cost, but only by at most ϵ.

Listing 2: Discrete Distributions in Ssreflect-Coq

Since our games A are finite, discrete distributions suffice to formalize MNEs, CEs, and CCEs. We model such distributions as finite functions (those with finite domain) from the strategy space A to $\mathbb{R}$:

```
Record dist (A : finType) : Type ≜
  mkDist { pmf :> {ffun A → ℝ}; dist_ax : dist_axiom pmf }.
```

Here {ffun $A \to \mathbb{R}$} is Ssreflect syntax for the type of finite functions from A to $\mathbb{R}$. The second projection of the record, dist_ax, asserts that pmf represents a valid distribution: pmf is positive and $\sum_{a:A}$ pmf $a = 1$.
The Coq predicate eCCE:

```
Definition eCCE (ε : ℝ) (σ : dist A^N) : Prop ≜
  ∀(i : [0..N − 1]) (s' : A),
  expectedCost i σ ≤ (expectedUnilateralCost i σ s') + ε.
```

states that distribution σ (over N-tuples of strategies A, one per player) is an ϵ-approximate CCE, or ϵ-CCE.

2.2 Algorithmic Game Theory

Equilibria are only useful if we're able to quantify, with respect to the game being analyzed:

1. How good equilibrium states are with respect to the optimal configurations of a game. By optimal, we usually mean states s^* that optimize the social cost: $\forall s.\ \sum_i C_i(s^*) \leq \sum_i C_i(s)$.
2. How "easy" (read computationally tractable) it is to drive competing players of the game toward an equilibrium state.

Algorithmic game theory and the related fields of mechanism design and distributed optimization provide excellent tools here.

Good Equilibria. The *Price of Anarchy*, or POA, of game (A, C) quantifies the cost of equilibrium states of (A, C) with respect to optimal configurations. Precisely, define POA as the ratio of the social cost of the worst equilibrium s to the social cost of an optimal state s^*. POA near 1 indicates high-quality equilibria: finding an equilibrium in such a game leads to overall social cost close to optimal. Prior work in algorithmic game theory has established nontrivial POA bounds for a number of game classes: on various classes of congestion and routing games [2,6,10], on facility location games [40], and others [11,32].

In the system of Sect. 3, we use the related concept of *Roughgarden smooth games* [35], or simply *smooth games*, which define a subclass of games with canonical POA proofs. To each smooth game are associated two constants, λ and μ. The precise definition of the smoothness condition is less relevant here than its consequences: if a cost-minimization game is (λ, μ)-smooth, then it has POA $\lambda/(1-\mu)$. Not all games are smooth, but for those that are, the POA bound above extends even to CCEs and their approximations, a particularly large (and therefore tractable) class of equilibria [35, Sects. 3 and 4].

Tractable Dynamics. Good equilibrium bounds are most useful when we know how quickly a particular game converges to equilibrium [7,9,12,13,17]. Certain classes of games, e.g. potential games [26], reach equilibria under a simple model of dynamics called best response. As we've mentioned, we use a different distributed learning algorithm in this work, variously called Multiplicative Weights (MW) [1] or sometimes Randomized Weighted Majority [25], which drives *all* games to CCEs, a larger class of equilibrium states than those achieved by potential games under best response.

3 Cage by Example

No-regret algorithms such as MW can be used to drive multi-agent systems toward the ϵ-CCEs of arbitrary games. Although the CCEs of general games may have high social cost, those of *smooth* games, as identified by Roughgarden [35], have robust Price of Anarchy (POA) bounds that extend even to ϵ-CCEs. Figure 1 depicts how these pieces fit together in the high-level architecture of our Cage system, which formalizes the results of Sect. 2 in Coq. Shaded boxes are program-related components while white boxes are proof related.

3.1 Overview

At the top, we have a domain-specific language in Coq (DSL, box 1) that generates games with automatically verified POA bounds. To execute such games, we have verified (also in Coq) an implementation of the Multiplicative Weights algorithm (MW, 2). Correctness of MW implies convergence bounds on the games it executes: $O((ln\ |A|)/\epsilon^2)$ iterations suffice to drive the game to an ϵ-CCE (here, $|A|$ is the size of the action space, or game type, A).

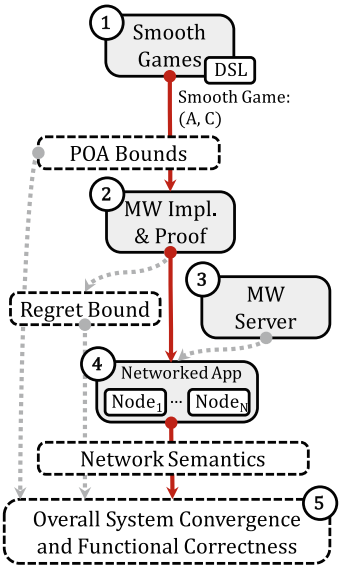


Fig. 1. System architecture

We compose N instances of multiplicative weights (4), one per agent, with a server (3) that facilitates communication, implemented in OCaml and modeled by an operational semantics in Coq. To actually execute games, we use Coq's code extraction mechanism to generate OCaml code that runs clients against the server, using an unverified OCaml shim to send and receive messages. We prove performance guarantees in Coq from POA bounds on the game and from the regret bound on MW.

3.2 Smooth Games DSL

The combinators exposed by the Smooth Games DSL operate over game types A, cost functions C, and smoothness parameters λ and μ. Basic combinators in this language include (i) Resource and (ii) Unit games, the first for coordinating access to shared resources under congestion and the second with fixed cost 0. Combinators that take other games as arguments include:

- the bias combinator $\mathsf{Bias}(A, b)$, which adds the fixed value b to the cost function associated with game A;
- the scalar combinator $\mathsf{Scalar}(A, m)$, which multiplies the output of the cost function C associated with game A by a fixed value m;
- the product combinator $A \times B$, corresponding to the parallel composition of two games A and B with cost equal to the sum of the costs in the two games;
- the subtype game $\{x : A,\ P(x)\}$, which constructs a new game over the dependent sum type $\Sigma x : A.P(x)$ (values x satisfying the predicate P);

– the singleton game $\mathsf{Singleton}(A)$, which has cost 1 if if player i "uses" the underlying resource ($\mathbb{B}_{\mathsf{Resource}}(f\ i) = \mathsf{true}$), and 0 otherwise. The function $\mathbb{B}_{\_}(-)$ generalizes the notion of resource usage beyond the primitive Resource game. For example, $\mathbb{B}_{\mathsf{Scalar}(A,m)}(x) = \mathbb{B}_A(x)$: usage in a game built from the scalar combinator reduces to usage in the underlying game.

3.3 Example: Distributed Routing

We illustrate the Smooth Games DSL with an example: distributed routing over networks with affine latency functions (Fig. 2). This game is known to have POA 5/2 [35].

In a simple version of the game, N routing agents each choose a path from a global source vertex s to a global sink vertex t. Latency over edge e, modeled by an affine cost function $c_e(x) = a_e x + b_e$, scales in the amount of traffic x over that edge. An optimal solution minimizes the total cost to all agents.

We model each link in the network as a Resource game, which in its most basic form is defined by the following inductive datatype:

```
Inductive Resource : Type ≜
  | RYes : Resource
  | RNo : Resource.
```

RYes indicates the agent chose to use the resource (a particular edge) and RNo otherwise. The cost function for Resource is defined by:

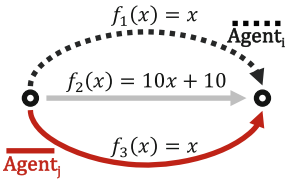


Fig. 2. Routing game with affine edge costs

```
Definition ResourceCostFun (i : [0..N − 1]) (s : [0..N − 1] →fin Resource) : ℝ ≜
  if s_i is RYes then traffic s else 0.
```

in which s is a map from agent labels to resource strategies and traffic s is the total number of agents that chose to use resource s. An agent pays traffic s if it uses the resource, otherwise 0. We implement Resource as a distinct inductive type, even though it's isomorphic to bool, to ensure that types in the Smooth Games DSL have unique game instances. To give each resource the more interesting cost function $c_e(x) = a_e x + b_e$, we compose Resource with a second combinator, $\mathsf{Affine}(a_e, b_e, \mathsf{Resource})$, which has cost 0 if an agent does not use the resource, and cost $a_e * (\mathsf{traffic}\ s) + b_e$ otherwise. This combinator preserves (λ, μ)-smoothness assuming $\lambda + \mu \geq 1$, a side condition which holds for Resource games.

We encode m affine resources by applying Affine to Resource m times, then folding under product:

```
T ≜ Affine(a_1,b_1,Resource)
  × Affine(a_2,b_2,Resource)
  × ...
  × Affine(a_m,b_m,Resource)
```

The associated cost function is the sum of the individual resource cost functions.

Values of type T may assign RYes to a subset of resources that doesn't correspond to a valid path in a graph $G = (V, E)$. To prevent this behavior, we apply to T the subtype combinator Σ, specialized to a predicate isValidPath(G, s, t) enforcing that strategies $(r_1, r_2, \ldots, r_{|E|})$ correspond to valid paths from s to t: T' $\triangleq \Sigma_{\mathsf{isValidPath}(G,s,t)}$(T). The game T' is (5/3, 1/3)-smooth, just like the underlying Resource game, which implies POA of $(5/3)/(1 - 1/3) = 5/2$.

3.4 Example: Load Balancing

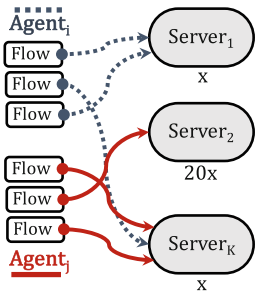


Fig. 3. Load balancing game

As a second example, consider the load balancing game depicted in Fig. 3, in which a number of network flows are distributed over several servers with affine cost functions. In general, N load balancing agents are responsible for distributing M flows over K servers. The cost of allocating a flow to a server is modeled by an affine cost function which scales in the total load (number of flows) on that server. Like routing, the load balancing game has POA 5/2. This is no coincidence; both are special cases of "finite congestion games", a class of games which have POA 5/2 when costs are linear [10]. The connection between them can be seen more concretely by observing that they are built up from the same primitive Resource game.

We model the system as an NM-player K-resource game in which each player corresponds to a single network flow. Each load balancing agent poses as multiple players (MW instances) in the game, one per flow, and composes the actions chosen by these players to form its overall strategy. The result of running the game is an approximate CCE with respect to the distribution of flows over servers.

Each server is defined as a Resource with an affine cost function, using the same data type and cost function as in the routing example. Instead of isValidPath, we use a new predicate exactlyOne to ensure that each network flow is assigned to exactly one server.

4 Smooth Games

Roughgarden smoothness [35] characterizes a subclass of games with canonical Price of Anarchy (POA) proofs. In [35], Roughgarden showed that smooth games have canonical POA bounds not only with respect to pure Nash equilibria but also with respect to mixed Nash equilibria, correlated equilibra, CCEs, and their approximate relaxations. In the context of Cage, we use smoothness to bound the social cost of games executed by multiple clients each running MW. We show how the technical pieces fit together, in the form of bounds on an operational semantics of the entire Cage system, in Sect. 6. This section introduces the technical definition of smoothness and the language of combinators,

Syntax

$$\begin{array}{l} \textit{Scalars } m, b; \quad \textit{Predicates } P \\ \textit{Game types } A, B ::= \mathsf{Resource} \mid \mathsf{Unit} \mid \mathsf{Bias}(A, b) \mid \mathsf{Scalar}(A, m) \\ \qquad\qquad\qquad\qquad \mid A \times B \mid \{x : A,\ P(x)\} \mid \mathsf{Singleton}(A) \end{array}$$

Judgment $\boxed{\vdash_{(\lambda,\mu)} (A, C)}$ read "Game (A, C) is (λ, μ)-smooth."

$$\frac{}{\vdash_{(\frac{5}{3},\frac{1}{3})} (\mathsf{Resource}, \mathsf{ResourceCostFun})}\ \mathsf{ResourceSmooth}$$

$$\frac{}{\vdash_{(1,0)} (\mathsf{Unit}, \mathsf{fun}\ i\ f.\ 0)}\ \mathsf{UnitSmooth}$$

$$\frac{\vdash_{(\lambda,\mu)} (A, C)}{\vdash_{(1,0)} (\mathsf{Singleton}(A), \mathsf{fun}\ i\ f.\ \mathsf{if}\ \mathbb{B}_A(f\ i)\ \mathsf{then}\ 1\ \mathsf{else}\ 0)}\ \mathsf{SingletonSmooth}$$

$$\frac{\vdash_{(\lambda,\mu)} (A, C)}{\vdash_{(\lambda,\mu)} (\{x : A,\ P(x)\}, \mathsf{fun}\ i\ f.\ C_i\ (\mathsf{fun}\ j.\ (f\ j).1))}\ \mathsf{SigmaSmooth}$$

$$\frac{\vdash_{(\lambda,\mu)} (A, C) \quad 1 \le \lambda + \mu \quad 0 \le b}{\vdash_{(\lambda,\mu)} (\mathsf{Bias}(A, b), \mathsf{fun}\ i\ f.\ C_i\ f + b)}\ \mathsf{BiasSmooth}$$

$$\frac{\vdash_{(\lambda,\mu)} (A, C) \quad 0 \le m}{\vdash_{(\lambda,\mu)} (\mathsf{Scalar}(A, m), \mathsf{fun}\ i\ f.\ m * C_i\ f)}\ \mathsf{ScalarSmooth}$$

$$\frac{\vdash_{(\lambda_A,\mu_A)} (A, C^A) \qquad \vdash_{(\lambda_B,\mu_B)} (B, C^B)}{\vdash_{(\max(\lambda_A,\lambda_B),\max(\mu_A,\mu_B))} (A \times B, \mathsf{fun}\ i\ f.\ C_i^A\ f + C_i^B\ f)}\ \mathsf{ProductSmooth}$$

Fig. 4. Smooth games DSL

or Smooth Games DSL of Sect. 3, that we use to build games that are smooth by construction.

Definition 1 (Smoothness). *A game (A, C) is (λ, μ)-smooth if for any two states $s, s^* : A^N$, the following inequality holds:*

$$\sum_{i=1}^{k} C_i(s_i^*, s_{-i}) \le \lambda \cdot C(s^*) + \mu \cdot C(s).$$

Here, $C_i(s_i^*, s_{-i})$ denotes the individual cost to player i in the mixed state where all other players follow their strategies from s, while player i follows the corresponding strategy from s^*. Smooth games bound the individual cost of players' unilateral deviations from state s to s^* by the weighted social costs of s and s^*. In essence, when λ and μ are small, the effect of any single player's deviation from a given state has minimal effect.

The smoothness inequality leads to natural proofs of POA for a variety of equilibrium classes. As an example, consider the following bound on the expected cost of ϵ-CCEs of (λ, μ)-smooth games:

Lemma smooth_eCCE $(d : \mathsf{dist}\ (\mathsf{state}\ N\ T))\ (s' : \mathsf{state}\ N\ T)\ (\epsilon : \mathbb{R})$:
$\mathsf{eCCE}\ \epsilon\ d \rightarrow \mathsf{optimal}\ s' \rightarrow$
$\mathsf{ExpectedCost}\ d \leq \lambda * (\mathsf{Cost}\ s') + \mu * (\mathsf{ExpectedCost}\ d) + N * \epsilon.$

ExpectedCost d is the sum for all players i of the expected cost to player i of distribution d. N is the number of players in the game.

The smooth_eCCE bound implies the following Price of Anarchy bound on the expected cost, summed across all players, of distribution d:

Lemma smooth_POA $\epsilon\ (d : \mathsf{dist}\ (\mathsf{state}\ N\ T))\ s'$:
$\mathsf{eCCE}\ \epsilon\ d \rightarrow \mathsf{optimal}\ s' \rightarrow$
$\mathsf{ExpectedCost}\ d \leq \lambda/(1-\mu) * (\mathsf{Cost}\ s') + (N * \epsilon)/(1-\mu).$

If d is an ϵ-CCE, then its cost is no more than $\lambda/(1$ - $\mu)$ times the optimal cost of s', plus an additional term that scales in the number of players N. For example, for concrete values $\lambda = 5/3$, $\mu = 1/3$, $\epsilon = 0.0375$, and $N = 5$, we get multiplicative approximation factor $\lambda/(1-\mu) = 5/2$ and additive factor 0.28. A value of $\epsilon = 0.0375$ is reasonable; as Sect. 5 will show, it takes fewer than 20,000 iterations of the Multiplicative Weights algorithm, in a game with strategy space of size 1000, to produce $\epsilon \leq 0.0375$.

4.1 Combinators

Figure 4 lists the syntax and combinators of the Smooth Games DSL we used in Sect. 3 to build smooth routing and load balancing games.

The smoothness proof accompanying the judgment of Resource games is the least intuitive, and provides some insight into the behavior of smooth games. The structure of our proof borrows from a stronger result given by Roughgarden [35]: smoothness for resource games with affine cost functions and multiple resources. The key step is the following inequality first noted by Christodoulou and Koutsoupias [10]:

$$y(z+1) \leq \frac{5}{3}y^2 + \frac{1}{3}z^2$$

for non-negative integers y and z. We derive $(\frac{5}{3}, \frac{1}{3})$-smoothness of Resource games from the following inequalities:

$$\sum_{i=0}^{N-1} C_i(s_i^*, s_{-i}) \leq (\mathsf{traffic}\ s^*) \cdot (\mathsf{traffic}\ s + 1) \tag{1}$$

$$(\mathsf{traffic}\ s^*) \cdot (\mathsf{traffic}\ s + 1) \leq \frac{5}{3} \cdot (\mathsf{traffic}\ s^*)^2 + \frac{1}{3} \cdot (\mathsf{traffic}\ s)^2 \tag{2}$$

$$(\mathsf{traffic}\ s^*) \cdot (\mathsf{traffic}\ s + 1) \leq \frac{5}{3} \cdot C(s^*) + \frac{1}{3} \cdot C(s) \tag{3}$$

$$\sum_{i=0}^{N-1} C_i(s_i^*, s_{-i}) \leq \frac{5}{3} \cdot C(s^*) + \frac{1}{3}\mu \cdot C(s) \tag{4}$$

The inequality in step 1 is due to the fact that the cost per player in state s^* is at most $\mathsf{traffic}\ s+1$, and there are exactly $\mathsf{traffic}\ s^*$ players incurring such cost. I.e., $(\mathsf{traffic}\ s^*) \cdot (\mathsf{traffic}\ s+1)$ is the number of nonzero terms times the upper bound on each term. The substitution in step 3 comes from the fact that in any state s, $C(s) = (\mathsf{traffic}\ s)^2$; each of the m players using the resource incur cost m.

The proofs of smoothness for other combinators are straightforward. For example, since Unit games always have cost 0, all values of λ and μ satisfy the smoothness inequality: $0 \leq \lambda \cdot 0 + \mu \cdot 0$. We restrict the range of the cost function in SingletonSmooth games to $\{0,1\}$ by applying the function $\mathbb{B}_A(\cdot)$, which generalizes the notion of "using a resource" to all the game types of Fig. 4. Smoothness of the Singleton game follows by case analysis on the results of $\mathbb{B}_A(\cdot)$ in the states s and s^* of the smoothness inequality. The games produced by the SigmaSmooth combinator have costs equal to those of the underlying games but restrict the domain to those states satisfying a predicate P. Since smoothness of the underlying bound holds for all states in A, the same bound holds of the restricted domain of states $a \in A$ drawn from P. Smoothness of product games relies on the fact that smoothness still holds if λ and μ are replaced with larger values. Thus, each of the argument games to ProductSmooth is $(\mathsf{max}(\lambda_A, \lambda_B), \mathsf{max}(\mu_A, \mu_B))$-smooth. The overall product game, which sums the costs of its argument games, is $(\mathsf{max}(\lambda_A, \lambda_B), \mathsf{max}(\mu_A, \mu_B))$-smooth as well.

It's possible to derive combinators from those defined in Fig. 4. For example, define as $\mathsf{Affine}(m, b, A)$ the game with cost function $mx + b$. We implement this game as $\{p : \mathsf{Scalar}(m, A) \times \mathsf{Scalar}(b, \mathsf{Singleton}(A)),\ p.1 = p.2\}$, or the subset of product games over the scalar game $\mathsf{Scalar}(m, A)$ and the $\{0,1\}$ scalar game over b such that the first and second projections of each strategy p are equal.

5 Multiplicative Weights (MW)

At the heart of the CAGE architecture of Sect. 3 lies our verified implementation of the Multiplicative Weights algorithm. In this section, we present the details of the algorithm and sketch its convergence proof. Section 5.3 presents our verified MW implementation and mechanized proof of convergence.

For all $a \in A$, client initializes $w_1(a) = 1$.

For time $t \in [1 \ldots T]$:

Client	*Environment*
Let $\Gamma_t \triangleq \sum_{a \in A} w_t(a)$.	
Play strategy $p_t(a) = w_t(a)/\Gamma_t$.	
	Choose cost vector c_t.
Update weights $w_{t+1}(a) \triangleq w_t(a) * (1 - \eta * c_t(a))$	

Fig. 5. Multiplicative Weights (MW)

5.1 The Algorithm

The MW algorithm (Fig. 5) pits a client, or agent, against an adaptive environment. The agent maintains a weight distribution w over the action space, initialized to give each action equal weight. At each time step $t \in [1 \dots T]$, the agent commits to the distribution $w_t / \sum_{a \in A} w_t(a)$, communicating this mixed strategy to the environment. After receiving a cost vector c_t from the environment, the agent updates its weights w_{t+1} to penalize high-cost actions, at a rate determined by a learning constant $\eta \in (0, 1/2]$. High η close to $1/2$ leads to higher penalties, and thus relatively less exploration of the action space.

The environment is typically adaptive, and may be implemented by a number of other agents also running instances of MW. The algorithm proceeds for a fixed number of epochs T, or until some bound on expected external regret (expected cost minus the cost of the best fixed action) is achieved. In what follows, we always assume that costs lie in the range $[-1, 1]$. Costs in an arbitrary but bounded range are also possible (with a concomitant relaxation of the algorithm's regret bounds), as are variations of MW to solve payoff maximization instead of cost minimization.

5.2 MW Is No Regret

The MW algorithm converges reasonably quickly: To achieve expected regret at most ϵ, it's sufficient to run the algorithm $O((ln\ |A|)/\epsilon^2)$ iterations, where $|A|$ is the size of the action space [36, Chapter 17]. Regret can be driven arbitrarily small as the number of iterations approaches infinity. Bounded regret suffices to prove convergence to an approximate CCE, as [36] also shows.

In this section, we present a high-level sketch of the proof that MW is no regret. We follow [36, Chapter 17], which has additional details. At the level of the mathematics, our formal proof makes no significant departures from Roughgarden.

Definition 2 (Per-Step External Regret). *Let a^* be the best fixed action in hindsight (i.e., the action with minimum cost given the cost vectors received from the environment) and let $OPT \triangleq \sum_{t=1}^{T} c_t(a^*)$. The expected per-step external regret of MW is*

$$\left(\sum_{t=1}^{T} \zeta_t - OPT \right) \ / \ T.$$

The summed term defines the cumulative expected cost of the algorithm for time $t \in [1 \dots T]$, where by ζ_t we denote the expected cost at time t:

$$\zeta_t = \sum_{a \in A} p_t(a) \cdot c_t(a) = \sum_{a \in A} \frac{w_t(a)}{\Gamma_t} \cdot c_t(a)$$

To get per-step expected regret, we subtract the cumulative cost of a^* and divide by the number of time steps T.

Theorem 1 (MW Has Bounded Regret). *The algorithm of Fig. 5 has expected per-step external regret at most* $\eta + ln\ |A|\ /\ \eta T$.

Proof Sketch. The proof of Theorem 1 uses a potential-function argument, with potential Φ_t equal the sum of the weights $\Gamma_t = \sum_{a \in A} w_t(a)$ at time t. It proceeds by relating the cumulative expected cost $\sum_t \zeta_t$ of the algorithm to OPT, the cost of the best fixed action, through the intermediate quantity Γ_{T+1}.

The proof additionally relies on the following two facts derived from the Taylor expansion $ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \cdots$:

$$\begin{aligned} ln(1-x) &\leq -x, & x &< 1 \\ -x - x^2 &\leq ln(1-x), & x &\leq 1/2 \end{aligned}$$

□

By letting $\eta = \sqrt{ln\ |A|\ /\ T}$ (cf. [36, Chapter 17]), it's possible to restate the regret bound of Theorem 1 to the following arguably nicer bound:

Corollary 1 (MW Is No Regret)

$$\left(\sum_{t=1}^{T} \zeta_t - OPT\right) / T \leq 2\sqrt{ln\ |A|\ /\ T}$$

Here, the number of iterations T must be large enough to ensure that $\eta = \sqrt{ln\ |A|\ /\ T} \leq 1/2$, thus ensuring that $\eta \in (0, 1/2]$.

5.3 MW Architecture

Our implementation and proof of MW (Fig. 6) were designed to be extensible. At a high level, the proof structure follows the program refinement methodology, in which a high-level mathematical but inefficient specification of MW (High-Level Functional Specification) is gradually made more efficient by a series of refinements to various features of the program (for example, by replacing an inefficient implementation of a key-value map with a more efficient balanced binary tree).

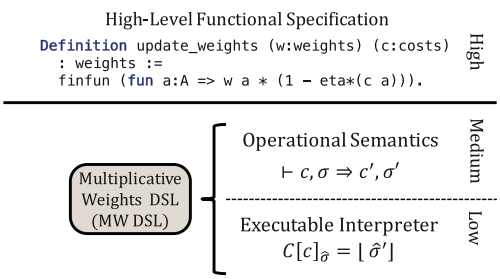


Fig. 6. MW architecture

For each such refinement, we prove that every behavior of the lower-level program is a possible behavior of the higher-level program it refines. Thus specifications proved for all behaviors of the high-level program also apply to each

behavior at the low level. By behavior here, we mean the trace of action distributions output by MW as it interacts with, and receives cost vectors from, the environment.

We factor the lower implementation layers (Medium and Low) into an interpreter and operational semantics over a domain-specific language specialized to MW-style algorithms (MW DSL). The DSL defines commands for maintaining and updating weights tables as well as commands for interacting with the environment. We prove, for any DSL program c, that the interpretation of that program refines its behavior with respect to the small-step operational semantics (Medium). Our overall proof specializes this general refinement to an implementation of MW as a command in the DSL, in order to relate that command's interpreted behavior to the high-level functional specification.

5.4 MW DSL

The syntax and semantics of the MW DSL are given in Fig. 7. The small-step operational semantics ($\vdash c, \sigma \Rightarrow c', \sigma'$) is parameterized by an environment oracle that defines functions for sending action distributions to the environment (oracle_send) and for receiving the resulting cost vectors (oracle_recv). The oracle will in general be implemented by other clients also running MW (Sect. 6) but is left abstract here to facilitate abstraction and reuse. The oracle is stateful (the type T, of oracle states, may be updated both by oracle_send and oracle_recv).

Most of the operational semantics rules are straightforward. In the MW-Step-Weights rule for updating the state's weights table, we make use of an auxiliary expression evaluation function $E\_[-]$ (standard and therefore not shown in Fig. 7). The only other interesting rules are those for send and recv, which call oracle_send and oracle_recv respectively. In the relation oracle_recv, the first two arguments are treated as inputs (the input oracle state of type T and the channel) while the second two are treated as outputs (the cost vector of type $A \to \mathbb{Q}$ and the output oracle state). In the relation oracle_send, the first three arguments are inputs while only the last (the output oracle state) is an output.

Multiplicative Weights. As an example of an MW DSL program, consider our implementation (Listing 1.1) of the high-level MW of Fig. 5. To the right of each program line, we give comments describing the effect of each command. The program is itself divided into three functions:mult_weights_init, which initializes the weights table to assign weight 1 to each action a in the action space A; mult_weights_body, which defines the body of the main loop of MW; and mult_weights, which simply composes mult_weights_init with mult_weights_body.

Listing 1.1. MW DSL Implementation of Multiplicative Weights

```
Definition mult_weights_init (A : Type) ≜
  update (λ a : A ⇒ 1); (* For all a ∈ A, initialize w_1(a) = 1. *)
  send. (* Commit to the uniform distribution over actions. *)

Definition mult_weights_body (A : Type) ≜
  recv; (* Block until agent receives cost vector c_t from environment. *)
  update (λ a : A ⇒ weight a * (1 − η * cost a)); (* Update weights. *)
  send. (* Commit to distribution w_t/Γ_t. *)

Definition mult_weights (A : Type) (n : N.t) ≜
  mult_weights_init A; (* Initialize weights and commit to initial mixed strategy. *)
  iter n (mult_weights_body A). (* Do n iterations of the MW main loop. *)
```

The MW DSL contains commands and expressions that are specialized to MW-style applications. Consider the function mult_weights_body (line 5). It first receives a cost vector from the environment using the specialized recv command. At the level of the MW DSL, recv is somewhat abstract. The program does not specify, e.g., which network socket to use. Implementation details such as these are resolved by the MW interpreter, which we discuss below in Sect. 5.5.

After recv, mult_weights_body implements an update to its weights table as defined by the command: update $(\lambda a : A \Rightarrow \mathsf{weight}\ a * (1 - \eta * \mathsf{cost}\ a))$. As an argument to the update, we embed a function from actions $a \in A$ to expressions that defines how the weight of each action a should change at this step (time $t+1$). The expressions weight a and cost a refer to the weight and cost, respectively, of action a at time t. The anonymous function term is defined in SSREFLECT-COQ, the metalanguage in which the MW DSL is defined.

5.5 Interpreter

To run MW DSL programs, we wrote an executable interpreter in Coq with type:

```
interp (c : com A) (s : cstate) : option cstate.
```

The type cstate defines the state of the interpreter after each step, and in general maps quite closely to the type of states σ used in the MW DSL operational semantics. It is given by the record:

Syntax

$$\begin{aligned} \textit{Binary operators}\ \oplus &::= + \mid - \mid * \\ \textit{Expressions}\ \ e &::= d \mid -e \mid \mathsf{weight}\ a \mid \mathsf{cost}\ a \mid \eta \mid e_1 \oplus e_2 \\ \textit{Commands}\ \ c &::= \mathsf{skip} \mid \mathsf{update}\ (\lambda a : A \Rightarrow e) \mid c_1; c_2 \mid \mathsf{iter}\ n\ c \mid \mathsf{recv} \mid \mathsf{send} \end{aligned}$$

Environment Oracle

$$\begin{aligned} &\mathsf{oracle\_recv} : T \to \mathsf{oracle\_chanty} \to (A \to \mathbb{Q}) \to T \to \mathsf{Prop} \\ &\mathsf{oracle\_send} : T \to \mathsf{dist}\ A \to \mathsf{oracle\_chanty} \to T \to \mathsf{Prop} \end{aligned}$$

States $\sigma \triangleq$

$\{$ SCosts : $A \to \mathbb{Q}$; SCostsOk : $\forall a.\ \lvert\mathsf{SCosts}\ a\rvert \le 1$	Current cost vector
; SPrevCosts : seq $\{c : A \to \mathbb{Q} \mid \forall a.\ \lvert c\ a\rvert \le 1\}$	Previous cost vectors
; SWeights : $A \to \mathbb{Q}$	Weights table
; SWeightsOk : $\forall a.\ 0 < \mathsf{SWeights}\ a$	
; SEta : $\mathbb{Q}$; SEtaOk : $0 < \mathsf{SEta} \le 1/2$	The η parameter
; SOutputs : seq (dist A)	Committed distributions
; SChan : oracle_chanty	I/O channel
; SOracleSt : T $\}$.	Environment/oracle state

Operational Semantics

$$\frac{\sigma' = \sigma\{\mathsf{SWeights} \triangleq \lambda a : A \Rightarrow E_\sigma[e[x \leftarrow a]]\}}{\vdash \mathsf{update}\ (\lambda x : A \Rightarrow e), \sigma \Rightarrow \mathsf{skip}, \sigma'}\ \text{MW-Step-Weights}$$

$$\frac{}{\vdash \mathsf{skip}; c_2, \sigma \Rightarrow c_2, \sigma} \qquad \frac{\vdash c_1, \sigma \Rightarrow c_1', \sigma'}{\vdash c_1; c_2, \sigma \Rightarrow c_1'; c_2, \sigma'}$$

$$\frac{}{\vdash \mathsf{iter}\ 1\ c, \sigma \Rightarrow c, \sigma} \qquad \frac{1 < n}{\vdash \mathsf{iter}\ n\ c, \sigma \Rightarrow c; \mathsf{iter}\ (n-1)\ c, \sigma}$$

$$\frac{\mathsf{oracle\_recv}\ (\mathsf{SOracleSt}\ \sigma)\ (\mathsf{SChan}\ \sigma)\ c\ t}{\vdash \mathsf{recv}, \sigma \Rightarrow \mathsf{skip}, \sigma\{\mathsf{SCosts} \triangleq c;\ \mathsf{SPrevCosts} \triangleq \mathsf{SCosts}\ \sigma :: \mathsf{SPrevCosts}\ \sigma;\ \mathsf{SOracleSt} \triangleq t\}}$$

$$\frac{\mathsf{oracle\_send}\ (\mathsf{SOracleSt}\ \sigma)\ d\ ch\ t}{\vdash \mathsf{send}, \sigma \Rightarrow \mathsf{skip}, \sigma\{\mathsf{SOutputs} \triangleq d :: \mathsf{SOutputs}\ \sigma;\ \mathsf{SChan} \triangleq ch;\ \mathsf{SOracleSt} \triangleq t\}}$$

Fig. 7. MW DSL syntax and operational semantics, parameterized by an environment oracle defining the type T of environment states and the functions oracle_recv and oracle_send for interacting with the environment. The type A is that of states in the underlying game.

Record cstate : Type $\triangleq$	
$\{$ SCosts : M.t $\mathbb{Q}$	Current cost vector
; SPrevCosts : list (M.t $\mathbb{Q}$)	Previous cost vectors
; SWeights : M.t $\mathbb{Q}$	Weights table
; SEta : $\mathbb{Q}$	The η parameter
; SOutputs : list ($A \to \mathbb{Q}$)	Committed distributions
; SChan : oracle_chanty	I/O channel
; SOracleSt : T $\}$.	Environment/oracle state

At the level of cstates, we use efficient purely functional data structures such as AVL trees. For example, the type M.t $\mathbb{Q}$ denotes an AVL-tree map from actions A to rational numbers $\mathbb{Q}$. In the small-step semantics state, by contrast, we model the weights table not as a balanced binary tree but as a SSREFLECT-COQ finite function, of type $\{\mathsf{ffun}\ A \to \mathbb{Q}\}$, which directly maps actions of type A to values of type $\mathbb{Q}$.

To speed up computation on rationals, we use a dyadic representation $q = \frac{n}{2^d}$, which facilitates fast multiplication. We do exact arithmetic on dyadic $\mathbb{Q}$ instead of floating point arithmetic to avoid floating-point precision error. Verification of floating-point error bounds is an interesting but orthogonal problem (cf. [31,34]).

The field SOutputs in the cstate record, a list of functions mapping actions $a \in A$ to their probabilities, stores the history of weights distributions generated by the interpreter as send commands are executed. To implement commands such as send and recv, we parameterize our MW interpreter by an environment oracle, just as we did the operational semantics. The operations implemented by the interpreter environment oracle are functional versions of the operational semantics oracle_send and oracle_recv:

oracle_send′ : $\forall A$:Type, $T \to A \to$ oracle_chanty $* T$
oracle_recv′ : $\forall A$:Type, $T \to$ oracle_chanty $\to$ list $(A * \mathbb{Q}) * T$

The oracle state type T is provided by the implementation of the oracle, as in the operational semantics. The command oracle_send′ takes a state of type T and a value of type A as arguments and returns a pair of a channel of type oracle_chanty (on which to listen for a response from the environment) and a new oracle state of type T. The command oracle_recv′ takes as arguments the oracle state and channel and returns a list of (a, q) pairs, representing a cost vector over actions, along with the new oracle state.

5.6 Proof

The top-level theorem proved of our high-level functional specification of MW is:

Theorem perstep_weights_noregret :
(expCostsR − OPTR)/T $\leq \eta +$ ln size_A / ($\eta *$T).

The expression expCostsR is the cumulative expected cost of MW on a sequence of cost vectors, or the sum, for each time t, of the expected cost of the MW algorithm at time t. OPTR is the cumulative cost over T rounds of the best fixed action. The number η (a dyadic rational required to lie in range $(0, 1/2]$) is the learning parameter provided to MW and ln size_A is the natural log of the size of the action space A. T is the number of time steps. In contrast to the interpreter and semantics of Sect. 5.3 (where we do exact arithmetic on dyadics), for reasoning and specification at the level of the proof we use Coq's real number library and real-valued functions such as square root and log.

By choosing η to equal $\sqrt{ln\ \mathsf{size\_A} / T}$, Corollary 1 showed that it's possible to restate the right-hand side of the inequality in perstep_weights_noregret to

2 * sqrt (ln size_A / T), thus giving an arguably nicer bound. Since in our implementation of MW we require that η be a dyadic rational, we cannot implement $\eta = \sqrt{ln\ \text{size\_A} / T}$ directly (ln size_A is irrational). We do, however, prove the following tight approximation for all values of η approaching $\sqrt{ln\ \text{size\_A} / T}$:

```
Lemma perstep_weights_noregret' :
∀r : ℝ. r ≠ −1 → η = (1+r)*(sqrt (ln size_A / T)) →
(expCostsR − OPTR)/T ≤
(1+r)*(sqrt (ln size_A / T)) + (sqrt (ln size_A / T))/(1+r).
```

In the statement of this lemma, the r term quantifies the error (how far η is from its optimal value sqrt (ln size_A / T). We require that $r \neq -1$ to ensure that division by $1 + r$ is well-defined. The resulting bound approaches 2 * sqrt (ln size_A / T) as r approaches 0.

High-Level Functional Specification. Our high-level functional specification of MW closely models the mathematical specification of MW given in Fig. 5. For example, the following four definitions:

```
Definition weights : Type ≜ {ffun A → ℚ}.
Definition costs : Type ≜ {ffun A → ℚ}.
Definition init_weights : weights ≜ λ(_ : A) ⇒ 1.
Definition update_weights (w:weights) (c:costs) : weights ≜
  λa : A ⇒ w a * (1 − η * c a).
```

construct the types of weight (weights) and cost vectors (costs), represented as finite functions from A to $\mathbb{Q}$; define the initial weight vector (init_weights), which maps all actions to cost 1; and define the MW weight update rule (update_weights). The recursive function:

```
Fixpoint weights_of (cs : seq costs) (w : weights) : weights ≜
  if cs is c :: cs' then update_weights (weights_of cs' w) c else w.
```

defines the vector that results from using update_weights to repeatedly update w with respect to cost vectors cs.

Adaptive Vs. Oblivious Adversaries. In our high-level specification of MW, we parameterize functions like weights_of by a fixed sequence of cost vectors cs rather than model interaction with the environment, as is done in Fig. 5. An execution of our low-level interpreted MW, even against an adaptive adversary, is always simulatable by the high-level functional specification by recording in the low-level execution the cost vectors produced by the adversary, as is done by the SPrevCosts field (Sect. 5.5), and then passing this sequence to weights_of. This strategy is quite similar to using backward induction to solve the MW game for an oblivious adversary.

Connecting the Dots. To connect the MW interpreter to the high-level specification, we prove a series of refinement theorems (technically, backward simulations). As example, consider:

```
Lemma interp_step_plus :
  ∀(a0 : A) (s : state A) (t t' : cstate) (c : com A),
  interp c t = Some t' →
  match_states s t →
  ∃c' s', final_com c' ∧
    ((c =CSkip ∧ s = s') ∨ step_plus a0 c s c' s') ∧
    match_states s' t'.
```

which relates the behavior of the interpreter (interp c t) when run on an arbitrary command c in cstate t to our model of MW DSL commands as specified by the operational semantics.

To prove that the operational semantics correctly refines our high-level functional specification of MW (and therefore satisfies the regret bounds given at the start of Sect. 5.6), we prove a similar series of refinements. Since backward simulations compose transitively, we prove regret bounds on our interpreted MW just by composing the refinements in series. The bounds we prove in this way are parametric in the environment oracle with which MW is instantiated. When the oracle state types differ from source to target in a particular simulation, as is the case in our proof that the MW DSL interpreter refines the operational semantics, we require that the oracles simulate as well.

6 Coordinated MW

A system of multiple agents each running MW yields an ϵ-CCE of the underlying game. If the game being played is smooth – for example, it was built using the combinators of the Smooth Games DSL of Sect. 4 – then the resulting ϵ-CCE has bounded social cost with respect to a globally optimal strategy. In this section, we put these results together by (1) defining an operational semantics of distributed interaction among multiple clients each running MW, and (2) proving that distributed executions of this semantics yield near-optimal solutions, as long as the underlying game being played is smooth.

6.1 Machine Semantics

We model the evolution of the distributed machine by the operational semantics in Fig. 8. Client states (client_state) bundle commands from the MW DSL (Sect. 5) with MW states parameterized by the ClientPkg oracle. The client oracle send and receive functions model single-element (pin) queues, represented as values of type option (dist A), storing values sent by an MW node, and of type option ($A \to Q$), storing values received by an MW node.

States of the coordinated machine (type machine_state N A) map client indices in range $[0..N-1]$ to client states (type client_state A). Machine states also record, at each iteration of the distributed MW protocol, the history of distributions received from the clients in that round (type seq ($[0..N-1] \to$ dist A)), which will be used to prove Price of Anarchy bounds in the next section (Sect. 6.2). We say that all_clients_have_sent in a particular machine state m,

Client Oracle

$$\begin{aligned}
&\mathsf{ClientPkg} \triangleq \\
&\quad \{\ \mathsf{sent} : \mathsf{option}\ (\mathsf{dist}\ A); \\
&\qquad \mathsf{received} : \mathsf{option}\ (A \to \mathbb{Q}); \\
&\qquad \mathsf{received_ok} : \forall v.\ \mathsf{received} = \mathsf{Some}\ v \to \forall a.\ 0 \le v_a \le 1\ \} \\
&\mathsf{client_oracle_recv}\ A\ (p : \mathsf{ClientPkg})\ (- : \mathsf{unit})\ (v : A \to \mathbb{Q})\ (p' : \mathsf{ClientPkg}) \triangleq \\
&\quad p.\mathsf{received} = \mathsf{Some}\ v \wedge p'.\mathsf{received} = \mathsf{None} \wedge p'.\mathsf{sent} = p.\mathsf{sent} \\
&\mathsf{client_oracle_send}\ A\ (p : \mathsf{ClientPkg})\ (d : \mathsf{dist}\ A)\ (- : \mathsf{unit})\ (p' : \mathsf{ClientPkg}) \triangleq \\
&\quad p.\mathsf{sent} = \mathsf{None} \wedge p'.\mathsf{sent} = \mathsf{Some}\ d \wedge p'.\mathsf{received} = p.\mathsf{received}
\end{aligned}$$

Machine States

$$\begin{aligned}
&\mathsf{client_state}\ A \ni \sigma \triangleq (\mathsf{com}\ A * \mathsf{state}\ A\ \mathsf{ClientPkg}\ \mathsf{unit}) \\
&\mathsf{machine_state}\ N\ A \ni m \triangleq \\
&\quad \{\ \mathsf{clients} : [0..N-1] \to \mathsf{client_state}\ A; \\
&\qquad \mathsf{hist} : \mathsf{seq}\ ([0..N-1] \to \mathsf{dist}\ A)\ \} \\
&\mathsf{all_clients_have_sent}\ A\ (m : \mathsf{machine_state})\ (f : [0..N-1] \to \mathsf{dist}\ A) \triangleq \\
&\quad \forall i : [0..N-1].\ \textbf{let}\ (-, \sigma) \triangleq m.\mathsf{clients}\ i\ \textbf{in} \\
&\quad (\mathsf{SOracleSt}\ \sigma).\mathsf{received} = \mathsf{None} \wedge (\mathsf{SOracleSt}\ \sigma).\mathsf{sent} = \mathsf{Some}\ f_i.
\end{aligned}$$

Machine Step $\boxed{\vdash m \Longrightarrow m'}$

$$\frac{\begin{array}{c}\mathsf{cost\_vec}\ A\ i\ :\ A \to \mathbb{Q}\ \triangleq \lambda a.\ \sum_{(p:[0..N-1]\to A \mid p_i = a)} \prod_{(j \mid i \neq j)} f_j\ p_j * C_i\ p \\ m.\mathsf{clients}\ i = (c, \sigma) \qquad m'.\mathsf{clients}\ i = (c, \sigma') \qquad \sigma\ \sim_{\mathcal{O}}\ \sigma' \\ (\mathsf{SOracleSt}\ \sigma).\mathsf{sent} = \mathsf{None} \qquad (\mathsf{SOracleSt}\ \sigma').\mathsf{received} = \mathsf{Some}\ (\mathsf{cost\_vec}\ f\ i)\end{array}}{\mathsf{server\_sent\_cost\_vector}\ i\ f\ m\ m'}$$

$$\frac{m.\mathsf{clients}\ i = (c, \sigma) \qquad (\mathsf{SOracleSt}\ \sigma).\mathsf{sent} = \mathsf{None} \qquad \vdash c, \sigma \Rightarrow c', \sigma'}{\vdash m \Longrightarrow m\{\ \mathsf{clients} \triangleq m.\mathsf{clients}[i \mapsto (c', \sigma')]\ \}}\ \mathsf{ClientStep}$$

$$\frac{\begin{array}{c}\mathsf{all\_clients\_have\_sent}\ m\ f \\ (\forall i.\ \mathsf{server\_sent\_cost\_vector}\ i\ f\ m\ m') \qquad m'.\mathsf{hist} = f :: m.\mathsf{hist}\end{array}}{\vdash m \Longrightarrow m'}\ \mathsf{ServerStep}$$

Fig. 8. Semantics of the distributed machine

committing to the set of distributions f, if each client's received buffer is empty and its sent buffer contains the distribution f_i, of type dist A.

The machine step relation models a server–client protocol, distinguishing server steps (ServerStep) from client steps (ClientStep). Client steps, which run commands in the language of Fig. 7, may interleave arbitrarily. Server steps are synchronized by the all_clients_have_sent relation to run only after all clients have

completed the current round. The work done by the server is modeled by the auxiliary relation server_sent_cost_vector i f m m', which constructs and sends to client i the cost vector derived from the set of client distributions f. The relation $\sigma \sim_{\mathcal{O}} \sigma'$ states that σ and σ' are equal up to their SOracleSt components.

In the distributed MW setting, the cost to player i of a particular action $a : A$ is defined as the expected value, over all N-player strategy vectors p in which player i chose action a ($p_i = a$), of the cost to player i of p, with the expectation over the $(N-1)$-size product distribution induced by the players $j \neq i$.

6.2 Convergence and Optimality

Our proof that MW is no regret (Sect. 5) extends to system-wide convergence and optimality guarantees, with respect to the distributed execution model of Fig. 8 in which each client runs our MW implementation. The proof has three major steps:

1. Show that no-regret clients implementing MW are still no regret when interleaved in the distributed semantics of Fig. 8.
2. Prove that per-client regret bounds – one for each client running MW – imply system-wide convergence to an ϵ-CCE.
3. Use POA results for smooth games from Sect. 4 to bound the cost, with respect to that of an optimal state, of all such ϵ-CCEs.

Composing 1, 2, and 3 proves that the distributed machine of Fig. 8 – when instantiated to clients running MW – converges to near-optimal solutions to smooth games. We briefly describe each part in turn.

Part 1 : No-regret clients are still no regret when interleaved. That MW no-regret bounds lift to an MW client running in the context of the distributed operational semantics of Fig. 8 follows from the oracular structure of our implementation of MW (Sect. 5) – clients interact with other clients and with the server only through the oracle.

In particular, for any execution $\vdash m \Longrightarrow^{+} m'$ of the machine of Fig. 8, and for any client i, there is a corresponding execution of client i with respect to a small nondeterministic oracle that simply "guesses" which cost vector to supply every time the MW client executes a recv operation. Because MW is no regret for all possible sequences of cost vectors, proving a refinement against the nondeterministic oracle implies a regret bound on client i's execution from state m_i to state m'_i.

We lift this argument to all the clients running in the Fig. 8 semantics by proving the following theorem:

Theorem all_clients_bounded_regret A m m' T (ϵ : rat) :
hist m = nil $\rightarrow$ 0 < size (hist m') $\rightarrow$ final_state m' $\rightarrow$
$\vdash m \Longrightarrow^{+} m' \rightarrow$
($\forall i$, m.clients i = (mult_weights A T, init_state A η tt (init_ClientPkg A))) $\rightarrow$
η + ln size_A/(η *T) $\leq \epsilon \rightarrow$
machine_regret_eps m' ϵ.

The predicate machine_regret_eps holds in state s', against regret bound ϵ, if all clients have expected regret in state s' at most ϵ (with respect to the σ_T distribution we describe below), for any rational ϵ larger than $\eta + \ln \text{size\_A}/(\eta * \text{T})$ (the regret bound we proved of MW in Sect. 5).

We assume that the history is empty in the initial state (hist m = nil), and that at least one round was completed (0 < size (hist m')). By final_state m', we mean that all clients have synchronized with the server (by receiving a cost vector and sending a distribution) and then have terminated in CSkip. All clients in state m are initialized to execute T steps of MW over game A (mult_weights A T), from an initial state and initial ClientPkg.

Part 2: System-wide convergence to an ϵ-CCE. The machine semantics of Fig. 8 converges to an approximate Coarse Correlated Equilibrium (ϵ-CCE).

More formally, consider an execution $\vdash m \Longrightarrow^+ m'$ of the Fig. 8 semantics that results in a state m' for which machine_regret_eps m' ϵ (all clients have regret at most ϵ, as established in Part I). The distribution σ_T, defined as the time-averaged history of the product of the distributions output by the MW clients at each round, is an ϵ-CCE:

$$\sigma_T \triangleq \lambda p.\ \frac{\sum_{i=1}^{T} \prod_{j=1}^{N} (\text{hist } m')_i^j\ p_j}{T}$$

By $(\text{hist } m')_i^j$ we mean the distribution associated to player j at time i, as recorded in the execution history stored in state m'. The value $((\text{hist } m')_i^j\ p_j)$ is the probability that client j chose action p_j in round i.

We formalize this property in the following Coq theorem:

Theorem machine_regret_eCCE m' ϵ :
machine_regret_eps m' ϵ →
eCCE ϵ σ_T.

which states that σ_T is an eCCE, with approximation factor ϵ, as long as each client's expected regret over σ_T is at most ϵ (machine_regret_eps m' ϵ) – exactly the property we proved in Part 1 above.

Part 3 System-wide regret bounds. The machine semantics of Fig. 8 converge to a state with expected cost bounded with respect to the optimal cost.

Consider an execution of the Fig. 8 semantics $\vdash m \Longrightarrow^+ m'$ and an ϵ satisfying the conditions of all_clients_bounded_regret. If the underlying game is smooth, the expected cost of the time-averaged distribution of the clients in m', σ_T, is bounded with respect to the cost of an optimal strategy profile s' by the following Coq theorem:

Theorem systemwide_POA_bound A m m' T (ϵ : rat) s' :
hist m = nil → $\vdash m \Longrightarrow^+ m'$ → 0 < size (hist m') → final_state m' →
($\forall i$, m.clients i = (mult_weights A T, init_state A η tt (init_ClientPkg A))) →
η + ln size_A/(η*T) ≤ ϵ →
optimal s' →
ExpectedCost σ_T ≤ λ/(1−μ) * Cost s' + (N*ϵ/(1−μ))

In the above theorem, λ and μ are the smoothness parameters of the game A while N is the number of players. Cost s' is the social (total) cost of the optimal state s'.

7 Related Work

Reinforcement Learning, Bandits. There is extensive work on reinforcement learning [39], multi-agent reinforcement learning (MARL [19]), and multi-armed bandits (MAB, [15]), more than can be cited here. We note, however, that Q-learning [41], while similar in spirit to MW, addresses the more general scenario in which an agent's action space is modeled by an arbitrary Markov Decision Process (in MW, the action space is a single set A). Our verified MW implementation is most suitable, therefore, for use in the full-information analog of MAB problems, in which actions are associated with "arms" and each agent learns the cost of all arms – not just the one it pulled – at each time step. In this domain, MW has good convergence bounds, as we prove formally of our implementation in this paper. Relaxing our verified MW and formal proofs to the partial information Bandit setting is interesting future work.

Verified Distributed Systems. EventML [33] is a domain-specific language for specifying distributed algorithms in the Logic of Events, which can be mechanically verified within the Nuprl proof assistant. Work has been done to develop methods for formally verifying distributed systems in Isabelle [20]. Model checking has been used extensively (e.g., [21,24]) to test distributed systems for bugs.

Verdi [42] is a Coq framework for implementing verified distributed systems. A Verdi system is implemented as a collection of handler functions which exchange messages through the network or communicate with the "outside world" via input and output. Application-level safety properties of the system can be proved with respect to a simple, idealized network semantics. A verified system transformer (VST) can then be used to transform the executable system into one which is robust to network faults such as reordering, duplication, and dropping of packets. The safety properties of the system proved under the original network semantics are preserved under the new faulty semantics, with minimal additional proof effort required of the programmer.

The goals of Verdi are complementary to our own. We implement a verified no-regret MW algorithm, together with a language of Roughgarden smooth games, for constructing distributed systems with verified convergence and correctness guarantees. Verdi allows safety properties of a distributed system to be lifted to analogous systems which tolerate various network faults, and provides a robust runtime system for execution in a practical setting. It stands to reason, then, that Verdi (as well as follow-on related work such as [37]) may provide a natural avenue for building robust executable versions of our distributed applications. We leave this for future work.

Chapar [23] is a Coq framework for verifying causal consistency of distributed key-value stores as well as correctness of client programs with respect to causally

consistent key-value stores. The implementation of a key-value store is proved correct with respect to a high-level specification using a program refinement method similar to ours. Although Chapar's goal isn't to verify robustness to network faults, node crashes and message losses are modeled by its abstract operational semantics.

IronFleet [18] is a framework and methodology for building verified distributed systems using a mix of TLA-style state machine refinement, Hoare logic, and automated theorem proving. An IronFleet system is comprised of three layers: a high-level state machine specification of the overall system, a more detailed distributed protocol layer which describes the behavior of each agent in the system as a state machine, and the implementation layer in which each agent is programmed using a variant of the Dafny [22] language extended with a trusted set of UDP networking operations. Correctness properties are proved with respect to the high-level specifications, and a series of refinements is used to prove that every behavior in the implementation layer is a refinement of some behavior in the high-level specification. IronFleet has been used to prove safety and liveness properties of IronRSL, a Paxos-based replicated state machine, as well as IronKV, a shared key-value store.

Alternative Proofs. Variant proofs of Theorem 1, such as the one via KL-divergence (cf. [1, Section 2.2]), could be formalized in our framework without modifying most parts of the MW implementation. In particular, because we have proved once and for all that our interpreted MW refines a high-level specification of MW, it would be sufficient to formalize the new proof just with respect to the high-level program of Sect. 5.6.

8 Conclusion

This paper reports on the first formally verified implementation of Multiplicative Weights (MW), a simple yet powerful algorithm for approximately solving Coarse Correlated Equilibria, among many other applications. We prove our MW implementation correct via a series of program refinements with respect to a high-level implementation of the algorithm. We present a DSL for building smooth games and show how to compose MW with smoothness to build distributed systems with verified Price of Anarchy bounds. Our implementation and proof are open source and available online.

Acknowledgments. This material is based on work supported by the National Science Foundation under Grant No. CCF-1657358. We thank the ESOP anonymous referees for their comments on an earlier version of this paper.

References

1. Arora, S., Hazan, E., Kale, S.: The multiplicative weights update method: a meta-algorithm and applications. Theor. Comput. **8**(1), 121–164 (2012)
2. Awerbuch, B., Azar, Y., Epstein, A.: The price of routing unsplittable flow. In: Proceedings of the thirty-seventh annual ACM Symposium on Theory of Computing, pp. 57–66. ACM (2005)
3. Bachelier, L.: Théorie mathématique du jeu. Annales Scientifiques de l'Ecole Normale Supérieure **18**, 143–209 (1901)
4. Bagnall, A., Merten, S., Stewart, G.: Brief announcement: certified multiplicative weights update: verified learning without regret. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017. ACM (2017)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2013)
6. Bhawalkar, K., Gairing, M., Roughgarden, T.: Weighted congestion games: price of anarchy, universal worst-case examples, and tightness. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 17–28. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15781-3_2
7. Blum, A., Monsour, Y.: Learning, regret minimization, and equilibria. In: Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.V. (eds.) Algorithmic Game Theory. Cambridge University Press, Cambridge (2007). Chapter 4
8. Borel, E.: La théorie du jeu et les équations intégrales à noyau symétrique. Comptes rendus de l'Académie des Sci. **173**(1304–1308), 58 (1921)
9. Borowski, H., Marden, J.R., Shamma, J.: Learning efficient correlated equilibrium. Submitted for journal publication (2015)
10. Christodoulou, G., Koutsoupias, E.: The price of anarchy of finite congestion games. In: Proceedings of the 37th Annual ACM Symposium on Theory of Computing, pp. 67–73. ACM (2005)
11. Demaine, E.D., Hajiaghayi, M., Mahini, H., Zadimoghaddam, M.: The price of anarchy in network creation games. In: Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 292–298. ACM (2007)
12. Fanelli, A., Moscardelli, L., Skopalik, A.: On the impact of fair best response dynamics. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 360–371. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32589-2_33
13. Foster, D.P., Vohra, R.V.: Calibrated learning and correlated equilibrium. Games Econ. Behav. **21**(1), 40–55 (1997)
14. Freund, Y., Schapire, R.E.: A desicion-theoretic generalization of on-line learning and an application to boosting. In: Vitányi, P. (ed.) EuroCOLT 1995. LNCS, vol. 904, pp. 23–37. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59119-2_166
15. Gittins, J.C.: Bandit processes and dynamic allocation indices. J. R. Stat. Soc. Ser. B (Methodol.) **41**(2), 148–177 (1979)
16. Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. Technical report, INRIA (2015)
17. Hart, S., Mas-Colell, A.: A simple adaptive procedure leading to correlated equilibrium. Econometrica **68**(5), 1127–1150 (2000)

18. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: IronFleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 1–17. ACM (2015)
19. Hu, J., Wellman, M.P., et al.: Multiagent reinforcement learning: theoretical framework and an algorithm. In: ICML, vol. 98, pp. 242–250. Citeseer (1998)
20. Küfner, P., Nestmann, U., Rickmann, C.: Formal verification of distributed algorithms. In: Baeten, J.C.M., Ball, T., de Boer, F.S. (eds.) TCS 2012. LNCS, vol. 7604, pp. 209–224. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33475-7_15
21. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
22. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
23. Lesani, M., Bell, C.J., Chlipala, A.: Chapar: certified causally consistent distributed key-value stores. In: ACM SIGPLAN Notices, vol. 51. ACM (2016)
24. Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MODIST: transparent model checking of unmodified distributed systems. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (2009)
25. Littlestone, N., Warmuth, M.K.: The weighted majority algorithm. In: Proceedings of the 30th Annual Symposium on Foundations of Computer Science. IEEE (1989)
26. Monderer, D., Shapley, L.S.: Potential games. Games Econ. Behav. **14**(1), 124–143 (1996)
27. Nash, J.: Non-cooperative games. Ann. Math. **54**(2), 286–295 (1951)
28. Nash, J.F.: Equilibrium in n-player games. Proc. Natl. Acad. Sci. (PNAS) **36**(1), 48–49 (1950)
29. von Neumann, J., Morgenstern, O.: Theory of Games and Economic Behavior, vol. 60. Princeton University Press, New Jersey (1944)
30. Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.V.: Algorithmic Game Theory, vol. 1. Cambridge University Press, New York (2007)
31. Panchekha, P., et al.: Automatically improving accuracy for floating point expressions. ACM SIGPLAN Not. **50**(6), 1–11 (2015)
32. Perakis, G., Roels, G.: The price of anarchy in supply chains: quantifying the efficiency of price-only contracts. Manag. Sci. **53**(8), 1249–1268 (2007)
33. Rahli, V.: Interfacing with proof assistants for domain specific programming using EventML. In: Proceedings of the 10th International Workshop on User Interfaces for Theorem Provers, Bremen, Germany (2012)
34. Ramananandro, T., et al.: A unified Coq framework for verifying C programs with floating-point computations. In: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs. ACM (2016)
35. Roughgarden, T.: Intrinsic robustness of the price of anarchy. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, pp. 513–522. ACM (2009)
36. Roughgarden, T.: Twenty Lectures on Algorithmic Game Theory. Cambridge University Press, New York (2016)
37. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. In: Proceedings of the ACM on Programming Languages, vol. 2 (POPL), Article 28 (2018)

38. Spitters, B., Van der Weegen, E.: Type classes for mathematics in type theory. Math. Struct. Comput. Sci. **21**(04), 795–825 (2011)
39. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, vol. 1. MIT press, Cambridge (1998)
40. Vetta, A.: Nash equilibria in competitive societies, with applications to facility location, traffic routing and auctions. In: Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science, pp. 416–425. IEEE (2002)
41. Watkins, C.J., Dayan, P.: Q-learning. Mach. Learn. **8**(3–4), 279–292 (1992)
42. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: ACM SIGPLAN Notices, vol. 50, pp. 357–368. ACM (2015)
43. Zermelo, E.: Über eine anwendung der mengenlehre auf die theorie des schachspiels. In: Proceedings of the Fifth International Congress of Mathematicians, vol. 2, pp. 501–504. II, Cambridge University Press, Cambridge (1913)

Program Verification by Coinduction

Brandon Moore[1], Lucas Peña[2(✉)], and Grigore Rosu[1,2]

[1] Runtime Verification, Inc., Urbana, IL, USA
[2] University of Illinois at Urbana-Champaign, Urbana, IL, USA
lpena7@illinois.edu

Abstract. We present a novel program verification approach based on coinduction, which takes as input an operational semantics. No intermediates like program logics or verification condition generators are needed. Specifications can be written using any state predicates. We implement our approach in Coq, giving a certifying language-independent verification framework. Our proof system is implemented as a single module imported unchanged into language-specific proofs. Automation is reached by instantiating a generic heuristic with language-specific tactics. Manual assistance is also smoothly allowed at points the automation cannot handle. We demonstrate the power and versatility of our approach by verifying algorithms as complicated as Schorr-Waite graph marking and instantiating our framework for object languages in several styles of semantics. Finally, we show that our coinductive approach subsumes reachability logic, a recent language-independent sound and (relatively) complete logic for program verification that has been instantiated with operational semantics of languages as complex as C, Java and JavaScript.

1 Introduction

Formal verification is a powerful technique for ensuring program correctness, but it requires a suitable verification framework for the target language. Standard approaches such as Hoare logic [1] (or verification condition generators) require significant effort to adapt and prove sound and relatively complete for a given language, with few or no theorems or tools that can be reused between languages. To use a software engineering metaphor, Hoare logic is a design pattern rather than a library. This becomes literal when we formalize it in a proof assistant.

We present instead a single language-independent program verification framework, to be used with an executable semantics of the target programming language given as input. The core of our approach is a simple theorem which gives a coinduction principle for proving partial correctness.

To trust a non-executable semantics of a desired language, an equivalence to an executable semantics is typically proved. Executable semantics of programming languages abound in the literature. Recently, executable semantics of several real languages have been proposed, e.g. of C [2], Java [3], JavaScript [4,5], Python [6], PHP [7], CAML [8], thanks to the development of executable semantics engineering frameworks like K [9], PLT-Redex [10], Ott [11], etc., which

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 589–618, 2018.
https://doi.org/10.1007/978-3-319-89884-1_21

make defining a formal semantics for a programming language almost as easy as implementing an interpreter, if not easier. Our coinductive program verification approach can be used with any of these executable semantics or frameworks, and is correct-by-construction: no additional "axiomatic semantics", "program logic", or "semantics suitable for verification" with soundness proofs needed.

As detailed in Sect. 6, we are not the first to propose a language-independent verification infrastructure that takes an operational semantics as input, nor the first to propose coinduction for proving isolated properties about some programs. However, we believe that coinduction can offer a fresh, promising and general approach as a language-independent verification infrastructure, with a high potential for automation that has not been fully explored yet. In this paper we make two steps in this direction, by addressing the following research questions:

RQ1 *Is it feasible to have a sound and (relatively) complete verification infrastructure based on coinduction, which is language-independent and versatile, i.e., takes an arbitrary language as input, given by its operational semantics?*

RQ2 *Is it possible to match, or even exceed, the capabilities of existing language-independent verification approaches based on operational semantics?*

To address RQ1, we make use of a key mathematical result, Theorem 1, which has been introduced in more general forms in the literature, e.g., in [12,13] and in [14]. We mechanized it in Coq in a way that allows us to instantiate it with a transition relation corresponding to any target language semantics, hereby producing certifying program verification for that language. Using the resulting coinduction principle to show that a program meets a specification produces a proof which depends only on the operational semantics. We demonstrate our proofs can be effectively automated, on examples including heap data structures and recursive functions, and describe the implemented proof strategy and how it can be reused across languages defined using a variety of operational styles.

To address RQ2, we show that our coinductive approach not only subsumes reachability logic [15], whose practicality has been demonstrated with languages like C, Java, and JavaScript, but also offers several specific advantages. Reachability logic consists of a sound and (relatively) complete proof system that takes a given language operational semantics as a *theory* and derives reachability properties about programs in that language. A mechanical procedure can translate any proof using reachability logic into a proof using our coinductive approach.

We first introduce our approach with a simple intuitive example, then prove its correctness. We then discuss mechanical verification experiments across different languages, show how reachability logic proofs can be translated into coinductive proofs, and conclude with related and future work. Our entire Coq formalization, proofs and experiments are available at [16].

2 Overview and Basic Notions

Section 4 will show the strengths of our approach by means of verifying rather complex programs. Here our objective is different, namely to illustrate it by verifying a trivial IMP (C-style) program: `s=0; while (--n) {s=s+n;}`. Let `sum` stand for the program and `loop` for its `while` loop. When run with a positive initial value n of `n`, it sets `s` to the sum of $1, \ldots, n-1$. To illustrate non-termination, we assume unbounded integers, so `loop` runs forever for non-positive n. An IMP language syntax sufficient for this example and a possible execution trace are given in Fig. 1. The exact step granularity is not critical for our approach, as long as diverging executions produce infinite traces.

Pgm ::= *Stmt*	⟨`s=0; while (--n) {s=s+n;}` | `n` ↦ 4⟩
Exp ::= *Id*	⟨`while (--n) {s=s+n;}` | `n` ↦ 4, `s` ↦ 0⟩
| *Int*	⟨`if (--n) {s=s+n; loop} else {skip}` | `n` ↦ 4, `s` ↦ 0⟩
| `--` *Id*	⟨`if (3) {s=s+n; loop} else {skip}` | `n` ↦ 3, `s` ↦ 0⟩
| *Exp* `op` *Exp*	⟨`s=s+n; loop` | `n` ↦ 3, `s` ↦ 0⟩
	⟨`s=0+n; loop` | `n` ↦ 3, `s` ↦ 0⟩
Stmt ::= `skip`	⟨`s=0+3; loop` | `n` ↦ 3, `s` ↦ 0⟩
| *Stmt* *Stmt*	⟨`s=3; loop` | `n` ↦ 3, `s` ↦ 0⟩
| *Id* `=` *Exp* `;`	⟨`skip; loop` | `n` ↦ 3, `s` ↦ 3⟩
| `if` *Exp* `{` *Stmt* `}`	⟨`while (--n) {s=s+n;}` | `n` ↦ 3, `s` ↦ 3⟩
`else {` *Stmt* `}`	... | ...
| `while` *Exp* `{` *Stmt* `}`	⟨`while (--n) {s=s+n;}` | `n` ↦ 1, `s` ↦ 6⟩
	⟨`if (--n) {s=s+n; loop} else {skip}` | `n` ↦ 1, `s` ↦ 6⟩
	⟨`if (0) {s=s+n; loop} else {skip}` | `n` ↦ 0, `s` ↦ 6⟩
	⟨`skip` | `n` ↦ 0, `s` ↦ 6⟩

Fig. 1. Syntax of **IMP** (left) and sample execution of `sum` (right)

While our coinductive program verification approach is self-contained and thus can be presented without reliance on other verification approaches, we prefer to start by discussing the traditional Hoare logic approach, for two reasons. First, it will put our coinductive approach in context, showing also how it avoids some of the limitations of Hoare logic. Second, we highlight some of the subtleties of Hoare logic when related to operational semantics, which will help understand the reasons and motivations underlying our definitions and notations.

2.1 Intuitive Hoare Logic Proof

A Hoare logic specification/triple has the form $\{\!|\varphi_{pre}|\!\}$ `code` $\{\!|\varphi_{post}|\!\}$. The convenience of this notation depends on specializing to a particular target language, such as allowing variable names to be used directly in predicates to stand for their values, or writing only the current statement. This hides details of the environment/state representation, and some framing conventions or compositionality assumptions over the unmentioned parts. A Hoare triple specifies a set of (partial correctness) reachability claims about a program's behavior, and it is

(IMP statement rules)

$$\frac{}{\{\!|\varphi[\mathtt{e}/\mathtt{x}]|\!\}\ \mathtt{x = e;}\ \{\!|\varphi|\!\}} \quad \text{(HL-ASGN)}$$

$$\frac{\{\!|\varphi_1|\!\}\ \mathtt{s}_1\ \{\!|\varphi_2|\!\},\ \ \{\!|\varphi_2|\!\}\ \mathtt{s}_2\ \{\!|\varphi_3|\!\}}{\{\!|\varphi_1|\!\}\ \mathtt{s}_1\ \mathtt{s}_2\ \{\!|\varphi_3|\!\}} \quad \text{(HL-SEQ)}$$

$$\frac{\{\!|\varphi \wedge \mathtt{e} \neq 0|\!\}\ \mathtt{s}_1\ \{\!|\varphi'|\!\},\quad \{\!|\varphi \wedge \mathtt{e} = 0|\!\}\ \mathtt{s}_2\ \{\!|\varphi'|\!\}}{\{\!|\varphi|\!\}\ \mathtt{if\ (e)\ then}\ \{\mathtt{s}_1\}\ \mathtt{else}\ \{\mathtt{s}_2\}\ \{\!|\varphi'|\!\}} \quad \text{(HL-IF)}$$

$$\frac{\{\!|\varphi \wedge \mathtt{e} \neq 0|\!\}\ \mathtt{s}\ \{\!|\varphi|\!\}}{\{\!|\varphi|\!\}\ \mathtt{while\ (e)}\ \{\mathtt{s}\}\ \{\!|\varphi \wedge \mathtt{e} = 0|\!\}} \quad \text{(HL-WHILE)}$$

(Generic rule)

$$\frac{\models \psi \rightarrow \varphi,\ \ \{\!|\varphi|\!\}\ \mathtt{s}\ \{\!|\varphi'|\!\},\ \ \models \varphi' \rightarrow \psi'}{\{\!|\psi|\!\}\ \mathtt{s}\ \{\!|\psi'|\!\}} \quad \text{(HL-CONSEQ)}$$

Fig. 2. IMP program logic.

typically an over-approximation (i.e., it specifies more reachability claims than desired or feasible). Specifically, assume some formal language semantics of IMP defining an execution step relation $R \subseteq C \times C$ on a set C of configurations of the form $\langle \mathtt{code} \,|\, \sigma \rangle$, like those in Fig. 1. We write $a \rightarrow_R b$ for $(a, b) \in R$. Section 2.3 (Fig. 3) discusses several operational semantics approaches we experimented with (Sect. 4), that yield such step relations R. A (partial correctness) *reachability claim* (c, P), relating an initial state $c \in C$ and a target set of states $P \subseteq C$, is *valid* (or *holds*) iff the initial state c can either reach a state in P or can take an infinite number of steps (with $\rightarrow_R$); we write $c \Rightarrow_R P$ to indicate that claim (c, P) is valid, and $a \rightarrow b$ or $c \Rightarrow P$ instead of $a \rightarrow_R b$ or $c \Rightarrow_R P$, resp., when R is understood. Then $\{\!|\varphi_{pre}|\!\}\mathtt{code}\{\!|\varphi_{post}|\!\}$ specifies the set of reachability claims

$$\{(\langle \mathtt{code} \,|\, \sigma_{pre} \rangle, \{\langle \mathtt{skip} \,|\, \sigma_{post} \rangle \mid \sigma_{post} \models \varphi_{post}\}) \mid \sigma_{pre} \models \varphi_{pre}\}$$

and it is *valid* iff all of its reachability claims are valid. It is necessary for P in reachability claims (c, P) specified by Hoare triples to be a set of configurations (and thus an over-approximation): it is generally impossible for φ_{post} to determine exactly the possible final configuration or configurations.

While one can prove Hoare triples valid directly using the step relation $\rightarrow_R$ and induction, or coinduction like we propose in this paper, the traditional approach is to define a language-specific proof system for deriving Hoare triples from other triples, also known as *a* Hoare logic, or program logic, for the target programming language. Figure 2 shows such a program logic for IMP. Hoare logics are generally not executable, so testing cannot show whether they match the *intended* semantics of the language. Even for a simple language like IMP, if one mistakenly writes $\mathtt{e} = 1$ instead of $\mathtt{e} \neq 0$ in rule (HL-WHILE), then one gets an incorrect program logic. When trusted verification is desired, the program logic

needs to be proved sound w.r.t. a reference executable semantics of the language, i.e, that each derivable Hoare triple is valid. This is a highly non-trivial task for complex languages (C, Java, JavaScript), in addition to defining a Hoare logic itself. Our coinductive approach completely avoids this difficulty by requiring no additional semantics of the programming language for verification purposes.

The property to prove is that `sum` (or more specifically `loop`) exits only when `n` is 0, with `s` as the sum $\sum_{i=1}^{n-1} i$ (or $\frac{n(n-1)}{2}$). In more detail, any configuration whose statement begins with `sum` and whose store defines `n` as n can run indefinitely or reach a state where it has just left the loop with $\mathtt{n} \mapsto 0$, $\mathtt{s} \mapsto \sum_{i=1}^{n-1} i$, and the store otherwise unchanged. As a Hoare logic triple, that specification is

$$\{\!|\mathtt{n} = n|\!\}\ \ \mathtt{s=0;\ while(--n)\{s=s+n;\}}\ \ \{\!|\mathtt{s} = \sum_{i=1}^{n-1} i \wedge \mathtt{n}{=}0|\!\}$$

As seen, this Hoare triple asserts the validity of the set of reachability claims

$$S \equiv \{(c_{n,\sigma}, P_{n,\sigma}) \mid \forall n, \forall \sigma \text{ undefined in } \mathtt{n}\} \tag{1}$$

where

$$c_{n,\sigma} \equiv \langle \mathtt{s=0;\ while(--n)\{s=s+n;\}} \,|\, \mathtt{n} \mapsto n,\ \sigma \rangle$$
$$P_{n,\sigma} \equiv \{\langle \mathtt{skip} \,|\, \mathtt{n} \mapsto 0, \mathtt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i, \sigma' \rangle \mid \forall \sigma' \text{ undefined in } \mathtt{n}, \mathtt{s}\}$$

We added the σ and σ' state frames above for the sake of complete details about what Hoare triples actually specify, and to illustrate why P in claims (c, P) needs to be a set. Since the addition/removal of σ and σ' does not change the subsequent proofs, for the remainder of this section, for simplicity, we drop them.

Now let us assume, without proof, that the proof system in Fig. 2 is sound (for the executable step relation $\rightarrow_R$ of IMP discussed above), and let us use it to derive a proof of the `sum` example. Note that the proof system in Fig. 2 assumes that expressions have no side effects and thus can be used unchanged in state formulae, which is customary in Hoare logics, so the program needs to be first translated out into an equivalent one without the problematic `--n` where expressions have no side effects. We could have had more Hoare logic rules instead of needing to translate the code segment, but this would quickly make our program logics significantly more complicated. Either way, with even a simple imperative programming language like we have here, it is necessary to either add Hoare logic rules to Fig. 2 or to modify our code segment. These inconveniences are taken for granted in Hoare logic based verifiers, and they require non-negligible additional effort if trusted verification is sought. For comparison, our coinductive verification approach proposed in this paper requires no transformation of the original program. After modifying the above problematic expression, our code segment gets translated to the (hopefully) equivalent code:

```
s=0; n=n-1; while (n) {s=s+n; n=n-1;}
```

Let `loop'` be the new loop and let φ_{inv}, its invariant, be

$$\mathtt{s} = \frac{((n-1) - \mathtt{n})\,(n + \mathtt{n})}{2}$$

The program variable `n` stands for its current value, while the mathematical variable n stands for the initial (sometimes called "old") value of `n`. Next, using the assign and sequence Hoare logic rules in Fig. 2, as well as basic arithmetic via the (HL-CONSEQ) rule, we derive

$$\{\!|\mathtt{n} = n|\!\}\ \ \mathtt{s=0;\ n=n-1;}\ \ \{\!|\varphi_{inv}|\!\} \tag{2}$$

Similarly, we can derive $\{\!|\varphi_{inv} \wedge \mathtt{n} \neq 0|\!\}$ `s=s+n; n=n-1;` $\{\!|\varphi_{inv}|\!\}$. Then, applying the while rule, we derive $\{\!|\varphi_{inv}|\!\}$ `loop'` $\{\!|\varphi_{inv} \wedge \mathtt{n} = 0|\!\}$. The rest follows by the sequence rule with the above, (2), and basic arithmetic.

This example is not complicated, in fact it is very intuitive. However, it abstracts out a lot of details in order to make it easy for a human to understand. It is easy to see the potential difficulties that can arise in larger examples from needing to factor out the side effect, and from mixing both program variables and mathematical variables in Hoare logic specifications and proofs. With our coinduction verification framework, all of these issues are mitigated.

2.2 Intuitive Coinduction Proof

Since our coinductive approach is language-independent, we do not commit to any particular, language-specific formalism for specifying reachability claims, such as Hoare triples. Consequently, we will work directly with raw reachability claims/specifications $S \subseteq C \times \mathcal{P}(C)$ consisting of sets of pairs (c, P) with $c \in C$ and $P \subseteq C$ as seen above. We show how to coinductively prove the claim for the example `sum` program in the form given in (1), relying on nothing but a general language-independent coinductive machinery and the trusted execution step relation $\rightarrow_R$ of IMP. Recall that we drop the state frames (σ) in (1).

Intuitively, our approach consists of symbolic execution with the language step relation, plus coinductive reasoning for circular behaviors. Specifically, suppose that $S_{circ} \subseteq C \times \mathcal{P}(C)$ is a specification corresponding to some code with circular behavior, say some loop. Pairs $(c, P) \in S_{circ}$ with $c \in P$ are already valid, that is, $c \Rightarrow_R P$ for those. "Execute" the other pairs $(c, P) \in S_{circ}$ with the step relation $\rightarrow_R$, obtaining a new specification S' containing pairs of the form (d, P), where $c \rightarrow_R d$; since we usually have a mathematical description of the pairs in S_{circ} and S', this step has the feel of symbolic execution. Note that S_{circ} is valid if S' is valid. Do the same for S' obtaining a new specification S'', and so on and so forth. If at any moment during this (symbolic) execution process we reach a specification S that is included in our original S_{circ}, then simply assume that S is valid. While this kind of cyclic reasoning may not seem sound, it is in fact valid, and justified by *coinduction*, which captures the essence of partial correctness, *language-independently*. Reaching something from the original specification shows we have reached some fixpoint, and coinduction is directly related to greatest fixpoints. This is explained in detail in Sect. 3.

In many examples it is useful to chain together individual proofs, similar to (HL-SEQ). Thus, we introduce the following sequential composition construct:

Definition 1. *For* $S_1, S_2 \subseteq C \times \mathcal{P}(C)$*, let* $S_1 \fatsemi S_2 \equiv \{(c,P) \mid \exists Q \,.\, (c,Q) \in S_1 \wedge \forall d \in Q, (d,P) \in S_2\}$*. Also, we define* trans(S) *as* $S \fatsemi S$ *(*trans *can be thought of as a transitivity proof rule).*

If S_1 and S_2 are valid then $S_1 \fatsemi S_2$ is also valid (Lemma 2).

Given n, let Q_n and T_n be the following sets of configurations, where Q_n and T_n represent the *invariant set* and *terminal set*, respectively:

$$Q_n \equiv \{\langle \texttt{loop} \,|\, \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i\rangle \mid \forall n'\}$$
$$T_n \equiv \{\langle \texttt{skip} \,|\, \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i\rangle\}$$

and let us define the following specifications:

$$S_1 \equiv \{(\langle \texttt{s=0; loop} \,|\, \texttt{n} \mapsto n\rangle, Q_n) \mid \forall n\}$$
$$S_2 \equiv \{(\langle \texttt{loop} \,|\, \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i\rangle, T_n) \mid \forall n, n'\}$$

Our target S in (1) is included in $S_1 \fatsemi S_2$, so it suffices to show that S_1 and S_2 are valid. S_1 clearly is: $\langle \texttt{s=0;loop} \,|\, \texttt{n} \mapsto n\rangle \rightarrow_R^+ \langle \texttt{loop} \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto 0\rangle$ represents the (symbolic) execution step or steps taken to assign program variable `s`, and the set of specifications $\{(\langle \texttt{loop} \,|\, \texttt{n} \mapsto n, \texttt{s} \mapsto 0\rangle, Q_n) \mid \forall n\}$ is vacuously valid (note $\sum_{i=n}^{n-1} i = 0$). For the validity of S_2, we partition it in two subsets, one where $n' = 1$ and another with $n' \neq 1$ (case analysis). The former holds same as S_1, noting that

$$\langle \texttt{loop} \,|\, \texttt{n} \mapsto 1, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i\rangle \rightarrow_R^+ \langle \texttt{skip} \,|\, \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i\rangle$$

The latter holds by coinduction (for S_2), because first

$$\langle \texttt{loop} \,|\, \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i\rangle \rightarrow_R^+ \langle \texttt{loop} \,|\, \texttt{n} \mapsto n' - 1, \texttt{s} \mapsto \textstyle\sum_{i=n'-1}^{n-1} i\rangle$$

and second the following inclusion holds:

$$\{(\langle \texttt{loop} \,|\, \texttt{n} \mapsto n' - 1, \texttt{s} \mapsto \textstyle\sum_{i=n'-1}^{n-1} i\rangle, T_n) \mid \forall n, n'\} \subseteq S_2$$

The key part of the proof above was to show that the reachability claim about the loop (S_2) was stable under the language semantics. Everything else was symbolic execution using the (trusted) operational semantics of the language. By allowing desirable program properties to be uniformly specified as reachability claims about the (executable) language semantics itself, our approach requires no auxiliary formalization of the language for verification purposes, and thus no soundness or equivalence proofs and no transformations of the original program to make it fit the restrictions of the auxiliary semantics. Unlike for the Hoare logic proof, the main "proof rules" used were just performing execution steps using the operational semantics rules, as well as the generic coinductive principle. Section 3 provides all the technical details.

Structural Operational Semantics

$$\langle x \mid \sigma \rangle \rightarrow \langle \sigma(x) \mid \sigma \rangle$$

$$\langle \texttt{--}x \mid \sigma \rangle \rightarrow \langle i \mid \sigma[i/x] \rangle \quad \text{if } i = \sigma(x) -_{Int} 1$$

$$\frac{\langle e_1 \mid \sigma \rangle \rightarrow \langle e_1' \mid \sigma' \rangle}{\langle e_1 \ \texttt{op} \ e_2 \mid \sigma \rangle \rightarrow \langle e_1' \ \texttt{op} \ e_2 \mid \sigma' \rangle}$$

$$\frac{\langle e_2 \mid \sigma \rangle \rightarrow \langle e_2' \mid \sigma' \rangle}{\langle i_1 \ \texttt{op} \ e_2 \mid \sigma \rangle \rightarrow \langle i_1 \ \texttt{op} \ e_2' \mid \sigma' \rangle}$$

$$\langle i_1 \ \texttt{op} \ i_2 \mid \sigma \rangle \rightarrow \langle i_1 \ op_{Int} \ i_2 \mid \sigma \rangle$$

$$\frac{\langle s_1 \mid \sigma \rangle \rightarrow \langle s_1' \mid \sigma' \rangle}{\langle s_1 \ \ s_2 \mid \sigma \rangle \rightarrow \langle s_1' \ \ s_2 \mid \sigma' \rangle}$$

$$\langle \texttt{skip} \ \ s \mid \sigma \rangle \rightarrow \langle s \mid \sigma \rangle$$

$$\frac{\langle e \mid \sigma \rangle \rightarrow \langle e' \mid \sigma' \rangle}{\langle x \ \texttt{:=} \ e \mid \sigma \rangle \rightarrow \langle x \ \texttt{:=} \ e' \mid \sigma' \rangle}$$

$$\langle x \ \texttt{:=} \ i \mid \sigma \rangle \rightarrow \langle \texttt{skip} \mid \sigma[i/x] \rangle$$

$$\frac{\langle e \mid \sigma \rangle \rightarrow \langle e' \mid \sigma' \rangle}{\langle \texttt{if} \ e \ \texttt{then} \ \{s_1\} \ \texttt{else} \ \{s_2\} \mid \sigma \rangle \rightarrow \langle \texttt{if} \ e' \ \texttt{then} \ \{s_1\} \ \texttt{else} \ \{s_2\} \mid \sigma' \rangle}$$

$$\langle \texttt{if} \ \ i \ \ \texttt{then} \ \{s_1\} \ \texttt{else} \ \{s_2\} \mid \sigma \rangle \rightarrow \langle s_1 \mid \sigma \rangle \quad \text{if } i \neq 0$$

$$\langle \texttt{if 0 then} \ \{s_1\} \ \texttt{else} \ \{s_2\} \mid \sigma \rangle \rightarrow \langle s_2 \mid \sigma \rangle$$

$$\langle \texttt{while} \ e \ \{s\} \mid \sigma \rangle \rightarrow \langle \texttt{if} \ e \ \texttt{then} \ \{s \ \texttt{while} \ e \ \{s\}\} \ \texttt{else} \ \ \{\texttt{skip}\} \mid \sigma \rangle$$

Reduction Semantics

(evaluation contexts syntax omitted— [17])

$$\frac{r \rightarrow r'}{E[r] \rightarrow E[r']}$$

$$\langle E \mid \sigma \rangle[x] \rightarrow \langle E \mid \sigma \rangle[\sigma(x)]$$

$$\langle E \mid \sigma \rangle[\texttt{--}x] \rightarrow \langle E \mid \sigma[i/x] \rangle[i] \quad \text{if } i = \sigma(x) -_{Int} 1$$

$$\langle E \mid \sigma \rangle[x \texttt{:=} i] \rightarrow \langle E \mid \sigma[i/x] \rangle[\texttt{skip}]$$

$$i_1 \ \texttt{op} \ i_2 \rightarrow i_1 \ op_{Int} \ i_2$$

$$\texttt{skip} \ \ s \rightarrow s$$

$$\texttt{if} \ \ i \ \ \texttt{then} \ \{s_1\} \ \texttt{else} \ \{s_2\} \rightarrow s_1 \quad \text{if } i \neq 0$$

$$\texttt{if 0 then} \ \{s_1\} \ \texttt{else} \ \{s_2\} \rightarrow s_2$$

$$\texttt{while} \ e \ \{s\} \rightarrow \texttt{if} \ e \ \texttt{then} \ \{s \ \texttt{while} \ e \ \{s\}\} \ \texttt{else} \ \{\texttt{skip}\}$$

K Semantics

(configuration and strictness omitted— [9])

$$\langle \underline{x} \ldots \rangle_k \ \langle \ldots x \mapsto i \ldots \rangle_{state}$$

$$\langle \underline{\texttt{--} \ x} \ldots \rangle_k \ \langle \ldots x \mapsto \underline{i} \ldots \rangle_{state}$$

$$\langle \underline{x \ \texttt{:=} \ i} \ldots \rangle_k \ \langle \ldots x \mapsto \underline{\ } \ldots \rangle_{state}$$

(plus the last five simple rules under reduction semantics)

Fig. 3. Three different operational semantics of **IMP**, generating the same execution step relation R (or $\rightarrow_R$).

2.3 Defining Execution Step Relations

Since our coinductive verification framework is parametric in a step relation, which also becomes the only trust base when certified verification is sought, it is imperative for its practicality to support a variety of approaches to define step relations. Ideally, it should not be confined to any particular semantic style that ultimately defines a step relation, and it should simply take existing semantics "off-the-shelf" and turn them into sound and relatively complete program verifiers for the defined languages. We briefly recall three of the semantic approaches that we experimented with in our Coq formalization [16].

Small-step structural operational semantics [18] (Fig. 3 top) is one of the most popular semantic approaches. It defines the transition relation inductively. This semantic style is easy to use, though often inconvenient to define some features such as abrupt changes of control and true concurrency. Additionally, finding the next successor of a configuration may take longer than in other approaches. Reduction semantics with evaluation contexts [17], depicted in the middle of Fig. 3, is another popular approach. It allows us to elegantly and compactly define complex evaluation strategies and semantics of control intensive constructs (e.g., call/cc), and it avoids a recursive definition of the transition relation. On the other hand, it requires an auxiliary definition of contexts along with splitting and plugging functions.

As discussed in Sect. 1, several large languages have been given formal semantics using K [9] (Fig. 3 bottom). K is more involved and less conventional than the other approaches, so it is a good opportunity to evaluate our hypothesis that we can just "plug-and-play" operational semantics in our coinductive framework. A K-style semantics extends the code in the configuration to a list of terms, and evaluates within subterms by having a transition that extracts the term to the front of the list, where it can be examined directly. This allows a non-recursive definition of transition, whose cases can be applied by unification.

In practice, in our automation, we only need to modify how a successor for a configuration is found. Besides that, the proofs remain exactly the same.

3 Coinduction as Partial Correctness

The intuitive coinductive proof of the correctness of `sum` in Sect. 2.2 likely raised a lot of questions. We give formal details of that proof in this section as well go through some definitions and results of the underlying theory. All proofs, including our Coq formalization, are in [16].

3.1 Definitions and Main Theorem

First, we introduce a definition that we used intuitively in the previous section:

Definition 2. *If* $R \subseteq C \times C$*, let* $\text{valid}_R \subseteq C \times \mathcal{P}(C)$ *be defined as* $\text{valid}_R = \{(c, P) \mid c \Rightarrow_R P \text{ holds}\}$.

Recall from Sect. 2.1 that $c \Rightarrow_R P$ holds iff the initial state c can either reach a state in P or can take an infinite number of steps (with $\rightarrow_R$). Pairs $(c, P) \in C \times \mathcal{P}(C)$ are called *claims* or *specifications*, and our objective is to prove they hold, i.e., $c \Rightarrow_R P$. Sets of claims $S \subseteq C \times \mathcal{P}(C)$ are valid if $S \subseteq \text{valid}_R$. To show such inclusions by coinduction, we notice that valid_R is a greatest fixpoint, specifically of the following operator:

Definition 3. *Given $R \subseteq C \times C$, let $\text{step}_R : \mathcal{P}(C \times \mathcal{P}(C)) \rightarrow \mathcal{P}(C \times \mathcal{P}(C))$ be*

$$\text{step}_R(S) = \{(c, P) \mid c \in P \ \vee \ \exists d \,.\, c \rightarrow_R d \wedge (d, P) \in S\}$$

Therefore, to prove $(c, P) \in \text{step}_R(S)$, one must show either that $c \in P$ or that $(\mathit{succ}(c), P) \in S$, where $\mathit{succ}(c)$ is a resulting configuration after taking a step from c by the operational semantics.

Definition 4. *Given a monotone function $F : \mathcal{P}(D) \rightarrow \mathcal{P}(D)$, let its F-*closure *$F^* : \mathcal{P}(D) \rightarrow \mathcal{P}(D)$ be defined as $F^*(X) = \mu Y.\, F(Y) \cup X$, where μ is the least fixpoint operator. This is well-defined as $Y \mapsto F(Y) \cup X$ is monotone for any X.*

The following lemma suffices for reachability verification:

Lemma 1. *For any $R \subseteq C \times C$ and $S \subseteq C \times \mathcal{P}(C)$, we have $S \subseteq \text{step}_R(\text{step}_R^*(S))$ implies $S \subseteq \text{valid}_R$.*

The intuition behind this lemma is captured in Sect. 2.2: we continue taking steps and once we reach a set of states already seen, we know our claim is valid. This would not be valid if $\text{step}_R(\text{step}_R^*(S))$ was replaced simply with $\text{step}_R^*(S)$, as $X \subseteq F^*(X)$ hold trivially for any F and X. Lemma 1 (along with elementary set properties) replaces the entire program logic shown in Fig. 2. The only formal definition specific to the target language is the operational semantics. Lemma 1 does not need to be modified or re-proven to use it with other languages or semantics. It generalizes into a more powerful result, that can be used to derive a variety of coinductive proof principles:

Theorem 1. *If $F, G : \mathcal{P}(D) \rightarrow \mathcal{P}(D)$ are monotone and $G(F(A)) \subseteq F(G^*(A))$ for any $A \subseteq D$, then $X \subseteq F(G^*(X))$ implies $X \subseteq \nu F$ for any $X \subseteq D$, where νF is the greatest fixpoint of F.*

Proofs, including a verified proof in our Coq formulation are in [16]. The proof can also be derived from [12–14], though techniques from these papers had previously not been applied to program verification. Lemma 1 is an easy corollary, with both F and G instantiated as step_R, along with a proof that $\nu\,\text{step}_R = \text{valid}_R$ (see [16]). However, instantiating F and G to be the same function is not always best. An interesting and useful G is the transitivity function trans in Definition 1, which satisfies the hypothesis in Theorem 1 when F is step_R. [16] shows other sound instantiations of G.

We can also use Theorem 1 with other definitions of validity expressible as a greatest fixpoint, e.g., all-path validity. For nondeterministic languages we might prefer to say $c \Rightarrow^\forall P$ holds if no path from c reaches a stuck configuration without passing through P. This is the greatest fixpoint of

$$\text{step}_R^\forall(S) = \{(c, P) \mid c \in P \ \vee \exists d \,.\, c \rightarrow_R d \wedge \forall d \,.\, (c \rightarrow_R d \text{ implies } (d, P) \in S)\}$$

The universe of validity notions that can be expressed coinductively, and thus the universe of instances of Theorem 1 is virtually limitless. Below is another notion of validity that we experimented with in our Coq formalization [16]. When proving global program invariants or safety properties of non-deterministic programs, we want to state not only reachability claims $c \Rightarrow P$, but also that all the transitions from c to configurations in P respect some additional property, say T. For example, a global state invariant I can be captured by a T such that $(a, b) \in T$ iff $I(a)$ and $I(b)$, while an arbitrary safety property can be captured by a T that encodes a monitor for it. This notion of validity, which we call (all-path) "until" validity, is the greatest fixpoint of:

$$\begin{aligned}\text{until}_R^{\forall}(S) = &\{(c, T, P) \mid c \in P \vee \\ &\exists d \,.\, c \rightarrow_R d \wedge \forall d \,.\, (c \rightarrow_R d \text{ implies } (c, d) \in T \wedge (d, T, P) \in S)\}\end{aligned}$$

This allows verification of properties that are not expressible using Hoare logic.

3.2 Example Proof: Sum

Now we demonstrate the results above by providing all the details that were skipped in our informal proof in Sect. 2.2. The property that we want to prove, expressed as a set of claims (c, P), is

$$\begin{aligned}S \equiv \{(&\langle \texttt{s=0;while(--n)\{s=s+n;\}}\, T \mid \texttt{n} \mapsto n, \sigma[\bot/\texttt{s}]\rangle, \\ &\{\langle T \mid \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i, \sigma\rangle\}) \mid \forall n, T, \sigma\}\end{aligned}$$

We have to prove $S \subseteq \text{valid}_R$. Note that this specification is more general than the specifications in Sect. 2.2. Here, T represents the remainder of the code to be executed, while σ represents the remainder of the store, with $\sigma[\bot/\texttt{s}]$ as σ restricted to $Dom(\sigma)/\{\texttt{s}\}$. Thus, we write out the entire configuration here, which gives us freedom in expressing more complex specifications if needed.

Instead of proving this directly, we will prove two subclaims valid and connect them via sequential composition (Definition 1). First, we need the following:

Lemma 2. $S_1 \fatsemi S_2 \subseteq \text{valid}_R$ *if* $S_1 \subseteq \text{valid}_R$ *and* $S_2 \subseteq \text{valid}_R$.

As before, let

$$\begin{aligned}Q_n &\equiv \{\langle \texttt{loop;}\ T \mid \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i, \sigma\rangle \mid \forall n'\} \\ T_n &\equiv \{\langle T \mid \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i\rangle\}\end{aligned}$$

and define

$$\begin{aligned}S_1 &\equiv \{(\langle \texttt{s=0; loop;}\ T \mid \texttt{n} \mapsto n, \sigma[\bot/\texttt{s}]\rangle, Q_n) \mid \forall n, T, \sigma\} \\ S_2 &\equiv \{(\langle \texttt{loop;}\ T \mid \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i, \sigma\rangle, T_n) \mid \forall n, n', T, \sigma\}\end{aligned}$$

Since $S \subseteq S_1 \fatsemi S_2$ (by Q_n), it suffices to show $S_1 \cup S_2 \subseteq \text{valid}_R$. To prove $S_1 \subseteq \text{valid}_R$, by Lemma 1 we show $S_1 \subseteq \text{step}_R(\text{step}_R^*(S_1))$. Regardless of the employed executable semantics, this should hold:

$$\forall n, T, \sigma.\, \langle \texttt{s=0; loop;}\ T \mid \texttt{n} \mapsto n, \sigma[\bot/\texttt{s}] \rangle \rightarrow_R \langle \texttt{loop;}\ T \mid \texttt{n} \mapsto n, \texttt{s} \mapsto 0, \sigma \rangle$$

Choosing the second case of the disjunction in step_R with d matching this step, it suffices to show

$$\{(\langle \texttt{loop;}\ T \mid \texttt{n} \mapsto n, \texttt{s} \mapsto 0, \sigma \rangle, Q_n) \mid \forall n, T, \sigma\} \subseteq \text{step}_R^*(S_1)$$

Note that we can unfold any fixpoint $F^*(S)$ to get the following two equations:

$$F(F^*(S)) \subseteq F(F^*(S)) \cup S = F^*(S) \qquad S \subseteq F(F^*(S)) \cup S = F^*(S) \tag{3}$$

We use the first equation to expose an application of step_R on the right hand side, so it suffices to show the above is a subset of $\text{step}_R(\text{step}_R^*(S))$. We then use the first case of the disjunction (showing $c \in P$) in step_R, and instantiating n' to n proves this goal, since $\sum_{i=n}^{n-1} i = 0$. Thus $S_1 \subseteq \text{valid}_R$.

Now we prove $S_2 \subseteq \text{valid}_R$, or $S_2 \subseteq \text{step}_R(\text{step}_R^*(S_2))$. First, note the operational semantics of IMP rewrites while loops to if statements. Then, by the definition of step_R, it suffices to show that

$$\{(\langle \texttt{if(--n)\{s=s+n;loop\};}\ T \mid \texttt{n} \mapsto n', \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i, \sigma \rangle, T_n) \mid \forall n, n', T, \sigma\} \subseteq \text{step}_R^*(S_2)$$

Using the first unfolding from (3), it suffices to show the above is a subset of $\text{step}_R(\text{step}_R^*(S_2))$, i.e. we expose an application of step_R on the right hand side. The definition of step_R thus allows the left hand side to continue taking execution steps, as long as we keep unfolding the fixpoint. Continuing this, the if condition becomes a single, but symbolic, boolean value. Specifically, it suffices to show:

$$\{(\langle \texttt{if(}n'\texttt{-1} \neq 0\texttt{)\{s=s+n;loop\};}\ T \mid \texttt{n} \mapsto n'\texttt{-1}, \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i, \sigma \rangle, T_n) \mid \forall n, n', T, \sigma\} \subseteq \text{step}_R^*(S_2)$$

Further progress requires making a case distinction on whether $n' - 1 = 0$. A case distinction corresponds to observing that $A \cup B \subseteq X$ if both $A \subseteq X$ and $B \subseteq X$. Here we split the current set of claims into those with $n' - 1 = 0$ and $n' - 1 \neq 0$, and separately establish the following inclusions:

$$\{(\langle \texttt{if(false)\{s=s+n;loop\};}\ T \mid \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i, \sigma \rangle, T_n) \mid \forall n, T, \sigma\} \subseteq \text{step}_R^*(S_2)$$

$$\{(\langle \texttt{if(true)\{s=s+n;loop\};}\ T \mid \texttt{n} \mapsto n'\texttt{-1}, \texttt{s} \mapsto \textstyle\sum_{i=n'}^{n-1} i, \sigma \rangle, T_n) \mid \forall n, n' \neq 1, T, \sigma\} \subseteq \text{step}_R^*(S_2)$$

Continuing symbolic execution and using $\sum_{i=n'}^{n-1} i + (n' - 1) = \sum_{i=n'-1}^{n-1} i$, we get

$$\{(\langle T \mid \texttt{n} \mapsto 0, \texttt{s} \mapsto \textstyle\sum_{i=1}^{n-1} i, \sigma \rangle, T_n) \mid \forall n, T, \sigma\} \subseteq \text{step}_R^*(S_2)$$

$$\{(\langle \texttt{loop;}\ T \mid \texttt{n} \mapsto n' - 1, \texttt{s} \mapsto \textstyle\sum_{i=n'-1}^{n-1} i, \sigma \rangle, T_n) \mid \forall n, n', T, \sigma, n' - 1 \neq 0\} \subseteq \text{step}_R^*(S_2)$$

In the $n' - 1 = 0$ case, the current configuration is already in the corresponding target set. To conclude, we expose another application of step_R as before, but use the clause $c \in P$ of the disjunction in step_R to leave the trivial goal $\forall n, T, \sigma.\, \langle T \mid \texttt{n} \mapsto 0, \texttt{s} \mapsto \frac{n(n-1)}{2}, \sigma \rangle \in \{\langle T \mid \texttt{n} \mapsto 0, \texttt{s} \mapsto \frac{n(n-1)}{2}, \sigma \rangle\}$. For the $n' - 1 \neq 0$ case,

we have a set of claims that are contained in the initial specification S_2. We conclude by showing $S_2 \subseteq \text{step}_R^*(S_2)$ from the second equation in (3) by noting that $S \subseteq F^*(S)$ for any F. So this set of claims is contained in S_2 by instantiating the universally quantified variable n' in the definition of S_2 with $n' - 1$. Thus it is contained in $\text{step}_R^*(S_2)$ and thus it is a subset of valid_R.

3.3 Example Proof: Reverse

Consider now the following program to reverse a linked list, written in the HIMP language (Fig. 5a). We will discuss HIMP in more detail Sect. 4.

```
decl p; decl y; p := 0;
while (x<>0) { y := (x+1); *(x+1) := p; p := x; x := y; }
```

Call the above code `rev` and the loop `rev-loop`. We prove this program is correct following intuitions from separation logic [19,20] but using the exact same coinductive technical machinery as before. Assuming we have a predicate that matches a heap containing only a linked list starting at address x and representing the list l (which we will see in Sect. 4.2), our specification becomes:

$$S \equiv \{(\langle \texttt{rev}; T \mid \text{list}(l, x)\rangle, \{\langle T \mid \lambda r.\text{list}(rev(l), r)\rangle\}) \mid \forall l, x, T\}$$

where rev is the mathematical list reverse. We proceed as in the previous example, first using lemma then stepping with the semantics, but with Q_n as

$$\{\langle \texttt{rev-loop}; T \mid \text{list}(A, x) * \text{list}(B, p) * \texttt{x} \mapsto x * \texttt{p} \mapsto p * \texttt{y} \mapsto y * \lambda r.\text{list}(B\texttt{++}A, r)\rangle \mid \forall A, B, p, y\}$$

where `++` is list append. We continue as before to prove our original specification. S_1 and S_2 follow from our choice for Q_n, our "loop invariant." Specifically,

$$S_1 \equiv \{(\langle \texttt{rev}; T \mid \text{list}(l, x)\rangle, \{\langle \texttt{rev-loop}; T \mid \text{list}(A, x) * \text{list}(B, p) * \texttt{x} \mapsto x * \texttt{p} \mapsto p * \texttt{y} \mapsto y * \lambda r.\text{list}(B\texttt{++}A, r)\rangle \mid \forall A, B, p, y\}) \mid \forall l, x, T\}$$

$$S_2 \equiv \{(\langle \texttt{rev-loop}; T \mid \text{list}(A, x) * \text{list}(B, p) * \texttt{x} \mapsto x * \texttt{p} \mapsto p * \texttt{y} \mapsto y * \lambda r.\text{list}(B\texttt{++}A, r)\rangle, \{\langle T \mid \lambda r.\text{list}(rev(l), r)\rangle\}) \mid \forall A, B, p, y, l, x, T\}$$

Then, the individual proofs for these specifications closely follow the same flavor as in the previous example: use step_R to execute the program via the operational semantics, use unions to case split as needed, and finish when we reach something in the target set or that was previously in our specification. The inherent similarity between these two examples hints that automation should not be too difficult. We go into detail regarding such automation in Sect. 4.

Reasoning with fixpoints and functions like step_R can be thought of as reasoning with proof rules, but ones which interact with the target programming language only through its operational semantics. The step_R operation corresponds, conceptually, to two such proof rules: taking an execution step and

HIMP

```
append(x, y)
  decl p;
  if (!x) return y;
  p := x;
  while(*(p+1)<>0) p := *(p+1);
  *(p+1) := y;
  return x;
```

Stack

```
: append over if over begin
1+ dup @ dup while nip repeat
drop ! else nip then ;
```

Lambda

```
(λ (λ IfNil 1 0
 ((λ (λ 0 0)
     (λ 1 (λ 1 1 0))) (λ
  (λ (λ (λ 0 1)) (Deref 0)
    (λ IfNil (Cdr 0)
      ((λ 5) (Assign 0
        (Cons (Car 0) 3)))
      (2 (Cdr 0)))))
 1)))
```

Fig. 4. Destructive list append in three languages.

showing that the current configuration is in the target set. Sequential composition and the trans rule corresponds to a transitivity rule used to chain together separate proofs. Unions correspond to case analysis. The fixpoint in the closure definition corresponds to iterative uses of these proof rules or to referring back to claims in the original specification.

4 Experiments

Now that we have proved the correctness of our coinductive verification approach and have seen some simple examples, we must consider the following pragmatic question: "Can this simple approach really work?". We have implemented it in Coq, and specified and verified programs in a variety of languages, each language being defined as an operational semantics [16]. We show not only that coinductive program verification is feasible and versatile, but also that it is amenable to highly effective proof automation. The simplifications in the manual proof, such as taking many execution steps at once, translate easily into proof tactics.

We first discuss the example languages and programs, and the reusable elements in specifications, especially an effective style of representation predicates for heap-allocated data structures. Then we show how we wrote specifications for example programs. Next we describe our proof automation, which was based on an overall heuristic applied unchanged for each language, though parameterized over subroutines which required somewhat more customization. Finally, we conclude with discussion of our verification of the Schorr-Waite graph-marking example and a discussion of our support for verification of divergent programs.

4.1 Languages

We discuss three languages following different paradigms, each defined operationally. Many language semantics are available with the distributions of K [9],

PLT-Redex [10], and Ott [11], e.g., but we believe these three languages are sufficient to illustrate the language-independence of our approach. Figure 4 shows a destructive linked list append function in each of the three languages.

HIMP (IMP with Heap) is an imperative language with (recursive) functions and a heap. The heap addresses are integers, to demonstrate reasoning about low-level representations, and memory allocation/deallocation are primitives. The configuration is a 5-tuple of current code, local variable environment mapping identifiers to values, call stack with frames as pairs of code and environment, heap, and a collection of functions as a map from function name to definition.

Stack is a Forth-like stack based language, though, unlike in Forth, we do make control structures part of the grammar. A shared data stack is used both for local state and to communicate between function invocations, eliminating the store, formal parameters on function declarations, and the environment of stack frames. Stack's configuration is also a 5-tuple, but instead of a current environment there is a stack of values, and stack frames do not store an environment.

Lambda is a call-by-value lambda calculus, extended with primitive integers, pair and nil values, and primitive operations for heap access. Fixpoint combinators enable recursive definitions without relying on primitive support for named functions. We use De Bruijn indices instead of named variables. The semantics is based on a CEK/CESK machine [21,22], extended with a heap. Lambda's configuration is a 4-tuple: current expression, environment, heap, continuation.

Pgm ::= *FunDef**

FunDef ::=
  Id `(` *Id*$^*_,$ `)` `{` *Stmt* `}`

Exp ::= *Id* `(` *Exp*$^*_,$ `)`
  | `alloc` | `load` *Exp*
  | *Exp* `.` *Id*
  | `build` *Map*
  | ...

Stmt ::= `*` *Exp* `:=` *Exp*
  | `dealloc` *Exp*
  | *Id* `(` *Exp*$^*_,$ `)` | `decl` *Id*
  | `return` *Exp* `;`
  | `return` `;`
  | ...

(a) **HIMP** syntax, extending the **IMP** syntax

Pgm ::= *FunDef**

FunDef ::=
  name `:` *Inst**

Inst ::= `Dup` n
  | `Roll` n
  | `Pop` | `Push` z
  | `BinOp` f
  | `Load` | `Store`
  | `Call` name | `Ret`
  | `If` *Inst** *Inst**
  | `While`
    Inst* *Inst**

(b) **Stack** syntax

Pgm ::= *Val*

Val ::= *Nat* | `Inc` | `Dec` | `Add`
  | `Add1` *Nat* | `Eq` | `Eq` *Val*
  | `Nil` | `Cons` | `Cons1` *Val*
  | `Car` | `Cdr`
  | `Closure` (*Exp*, *Env*)
  | `Pair` (*Val*, *Val*)

Exp ::= *Exp* *Exp* | λ *Exp*
  | `Var` *Nat*
  | `if` *Exp* `then` *Exp* `else` *Exp*
  | *Exp* `;` *Exp* | `Deref` *Exp*
  | `&` *Exp* | `*` *Exp* | *Exp* := *Exp*

Env ::= *Val**

(c) **Lambda** syntax

Fig. 5. Syntax of **HIMP**, **Stack**, and **Lambda**

4.2 Specifying Data Structures

Our coinductive verification approach is agnostic to how claims in $C \times \mathcal{P}(C)$ are specified. In Coq, we can specify sets using any definable predicates. Within this design space, we chose matching logic [23] for our experiments, which introduces patterns that concisely generalize the formulae of first order logic (FOL) and separation logic, as well as term unification. Symbols apply on patterns to build other patterns, just like terms, and patterns can be combined using FOL connectives, just like formulae. E.g., pattern $P \wedge Q$ matches a value if P and Q both match it, $[t]$ matches only the value t, $\exists x.P$ matches if there is any assignment of x under which P matches, and $[\![\varphi]\!]$ where φ is a FOL formula matches any value if φ holds, and no values otherwise (in [23] neither $[t]$ nor $[\![\varphi]\!]$ require a visible marker, but in Coq patterns are a distinct type, requiring explicit injections).

To specify programs manipulating heap data structures we use patterns matching subheaps that contain a data structure representing an abstract value. Following [24], we define representation predicates for data structures as functions from abstract values to more primitive patterns. The basic ingredients are primitive map patterns: pattern **emp** for the empty map, $k \mapsto v$ for the singleton map binding key k to value v, and $P * Q$ for maps which are a disjoint union of submaps matching P and, resp., Q. We use abbreviation $\langle\varphi\rangle \equiv [\![\varphi]\!] \wedge \mathbf{emp}$ to facilitate inline assertions, and $p \mapsto \{v_0, \ldots, v_i\} \equiv p \mapsto v_0 * \ldots * (p+i) \mapsto v_i$ to describe values at contiguous addresses. A heap pattern for a linked list starting at address p and holding list l is defined recursively by

$$\begin{aligned} \mathrm{list}(\mathrm{nil}, p) &= \langle p = 0 \rangle \\ \mathrm{list}(x : l, p) &= \langle p \neq 0 \rangle * \exists p_l \,.\, p \mapsto \{x, p_l\} * \mathrm{list}(l, p_l) \end{aligned}$$

We also define $\mathrm{list\_seg}(l, e, p)$ for list segments, useful in algorithms using pointers to the middle of a list, by generalizing the constant 0 (the pointer to the end of the list) to the trailing pointer parameter e. Also, simple binary trees:

$$\begin{aligned} \mathrm{tree}(\mathrm{leaf}, p) &= \langle p = 0 \rangle \\ \mathrm{tree}(\mathrm{node}(x, l, r), p) &= \langle p \neq 0 \rangle * \exists p_l, p_r . p \mapsto \{x, lp, rp\} * \mathrm{tree}(l, lp) * \mathrm{tree}(r, rp) \end{aligned}$$

Given such patterns, specifications and proofs can be done in terms of the abstract values represented in memory. Moreover, such primitive patterns are widely reusable across different languages, and so is our proof automation that deals with primitive patterns. Specifically, our proof scripting specific to such pattern definitions is concerned exclusively with unfolding the definition when allowed, deciding what abstract value, if any, is represented at a given address in a partially unfolded heap. This is further used to decide how another claim applies to the current state when attempting a transitivity step.

4.3 Specifying Reachability Claims

As mentioned, claims in $C \times \mathcal{P}(C)$ can be specified using any logical formalism, here the full power of Coq. An explicit specification can be verbose and low-level,

Table 1. Example list specifications

$$
\begin{array}{l}
\mathit{call}(\text{Head}, [x], [H] \wedge \text{list}(v : l, x), \lambda r.\langle r = v\rangle * [H]) \\
\mathit{call}(\text{Tail}, [x], [H] \wedge \text{list}(v : l, x), \lambda r.[H] \wedge \_ * \text{list}(l, r)) \\
\mathit{call}(\text{Add}, [y, x], \text{list}(l, x), \lambda r.\text{list}(y : l, r)) \\
\mathit{call}(\text{Add}', [y, x], [H] \wedge \text{list}(l, x), \lambda r.\text{list\_seg}([y], x, r) * [H]) \\
\mathit{call}(\text{Swap}, [x], \text{list}(a : b : l, x), \lambda r.\text{list}(b : a : l, x)) \\
\mathit{call}(\text{Dealloc}, [x], \text{list}(l, x), \lambda r.\mathbf{emp}) \\
\mathit{call}(\text{Length}, [x], [H] \wedge \text{list}(l, x), \lambda r.\langle r = \mathit{len}(l)\rangle * [H]) \\
\mathit{call}(\text{Sum}, [x], [H] \wedge \text{list}(l, x), \lambda r.\langle r = \mathit{sum}(l)\rangle\rangle * [H]) \\
\mathit{call}(\text{Reverse}, [x], \text{list}(l, x), \lambda r.\text{list}(\mathit{rev}(l), r)) \\
\mathit{call}(\text{Append}, [x, y], \text{list}(a, x) * \text{list}(b, y), \lambda r.\text{list}(a{+\!\!+}b, r)) \\
\mathit{call}(\text{Copy}, [x], [H] \wedge \text{list}(l, x), \lambda r.\text{list}(l, r) * [H]) \\
\mathit{call}(\text{Delete}, [v, x], \text{list}(l, x), \lambda r.\text{list}(\mathit{delete}(v, l), r))
\end{array}
$$

especially when many semantic components in the configuration stay unchanged. However, any reasonable logic allows making definitions to reduce verbosity and redundancy. Our use of matching logic particularly facilitates framing conditions, allowing us to regain the compactness and elegance of Hoare logic or separation logic specifications with definable syntactic sugar. For example, defining

$$
\begin{array}{l}
\mathit{call}(f(\mathit{formals})\{\mathit{body}\}, \mathit{args}, P_{in}, P_{out}) = \\
\{(\langle f(\mathit{args}) \curvearrowright \mathit{rest}, \mathit{env}, \mathit{stk}, \mathit{heap}, \mathit{funs}\rangle,\ \{\langle r \curvearrowright \mathit{rest}, \mathit{env}, \mathit{stk}, \mathit{heap}', \mathit{funs}\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \forall r, \mathit{heap}'.\ \mathit{heap}' \vDash P_{out}(r) * [H_f]\}) \\
\mid \forall \mathit{rest}, \mathit{env}, \mathit{stk}, \mathit{heap}, H_f, \mathit{funs}.\ \mathit{heap} \vDash P_{in} * [H_f] \wedge f{\mapsto}f(\mathit{formals})\{\mathit{body}\} \in \mathit{funs}\}
\end{array}
$$

gives the equivalent of the usual Hoare pre-/post-condition on function calls, including heap framing (in separation logic style). The notation $x \curvearrowright y$ represents the order of evaluation: evaluate x first followed by y. This is often used when y can depend on the value x takes after evaluation.

The first parameter is the function definition. The second is the arguments. The heap effect is described as a pattern P_{in} for the allowable initial states of the heap and function P_{out} from returned values to corresponding heap patterns. For example, we specify the definition D of append in Fig. 4 by writing $\mathit{call}(D, [x, y], (\text{list}(a, x) * \text{list}(b, y)), (\lambda r.\text{list}(a{+\!\!+}b, r)))$, which is as compact and elegant as it can be. More specifications are given in Table 1. A number of specifications assert that part of the heap is left entirely unchanged by writing $[H] \wedge \ldots$ in the precondition to bind a variable H to a specific heap, and using the variable in the postcondition (just repeating a representation predicate might permit a function to reallocate internal nodes in a data structure to different addresses). The specifications Add and Add' show that it can be a bit more complicated to assert that an input list is used undisturbed as a suffix of a result list. Specifications such as Length, Append, and Delete are written in

terms of corresponding mathematical functions on the lists represented in the heap, separating those functional descriptions from details of memory layout.

When a function contains loops, proving that it meets a specification often requires making some additional claims about configurations which are just about to enter loops, as we saw in Sect. 2.2. We support this with another pattern that takes the current code at an intermediate point in the execution of a function, and a description of the environment:

$$\begin{aligned} &\mathit{stmt}(\mathit{code}, \mathit{env}, P_{in}, P_{out}) = \\ &\{(\langle \mathit{code}, (\mathit{env}, e_f), \mathit{stk}, \mathit{heap}, \mathit{funs}\rangle, \{\langle \texttt{return}\ r \curvearrowright \mathit{rest}, \mathit{env}', \mathit{stk}, \mathit{heap}', \mathit{funs}\rangle \\ &\qquad\qquad \mid \forall r, \mathit{rest}, \mathit{env}', \mathit{heap}'.\mathit{heap}' \vDash P_{out}(r) * [H_f]\}) \\ &\qquad \mid \forall e_f, \mathit{stk}, \mathit{heap}, H_f, \mathit{funs}\ .\ \mathit{heap} \vDash P_{in} * [H_f]\} \end{aligned}$$

Verifying the definition of append in Fig. 4 meets the call specification above requires an auxiliary claim about the loop, which can be written using *stmt* as

$$\begin{aligned} &\mathit{stmt}(\texttt{while (*(p+1)<>0)} \ldots, (\texttt{x} \mapsto x, \texttt{y} \mapsto y, \texttt{p} \mapsto p), \\ &\quad (\mathrm{list\_seg}(l_x, p, x) * \mathrm{list}(l_p, p) * \mathrm{list}(l_y, y)), (\lambda r.\mathrm{list}(l_x +\!\!+ l_p +\!\!+ l_y, r))) \end{aligned}$$

The patterns above were described using HIMP's configurations; we defined similar ones for Stack and Lambda also.

4.4 Proofs and Automation

The basic heuristic in our proofs, which is also the basis of our proof automation, is to attack a goal by preferring to prove that the current configuration is in the target set if possible, then trying to use claims in the specification by transitivity, and only last resorting to taking execution steps according to the operational semantics or making case distinctions. Each of these operations begins, as in the example proofs, with certain manipulations of the definitions and fixpoints in the language-independent core. Our heuristic is reusable, as a proof tactic parameterized over sub-tactics for the more specific operations. A prelude to the main loop begins by applying the main theorem to move from claiming validity to showing a coinduction-style inclusion, and breaking down a specification with several classes of claims into a separate proof goal for each family of claims.

Additionally, our automation leverages support offered by the proof assistant, such as handling conjuncts by trying to prove each case, existentials by introducing a unification variable, equalities by unification, and so on. Moreover, we added tactics for map equalities and numerical formulae, which are shared among all languages involving maps and integers. The current proof goal after each step is always a reachability claim. So even in proofs which are not completely automatic, the proof automation can give up by leaving subgoals for the user, who can reinvoke the proof automation after making some proof steps of their own as long as they leave a proof goal in the same form.

Proving the properties in Table 1 sometimes required making additional claims about while loops or auxiliary recursive functions. All but the last four were proved automatically by invoking (an instance of) our heuristic proof tactic:

```
Proof. list_solver. Qed.
```

Append and copy needed to make use of associativity of list append. Reverse used a loop reversing the input list element by element onto an output list, which required relating the tail recursive $rev\_app(x : l, y) = rev\_app(l, x : y)$ with the Coq standard library definition $rev(x : l) = rev(l)\mathbin{+\!\!+}[x]$. Manually applying these lemmas merely modified the proof scripts to

```
list_solver. rewrite app_ass in * |- . list_run.
list_solver. rewrite <- rev_alt in * |- . list_run.
```

These proofs were used verbatim in each of our example languages. The only exceptions were append and copy for Lambda, for which the `app_ass` lemma was not necessary. For Delete, simple reasoning about $delete(v, l)$ when v is and is not at the head of the list is required, though the actual reasoning in Coq varies between our example languages. No additional lemmas or tactics equivalent to Hoare rules are needed in any of these proofs.

4.5 Other Data Structures

Matching logic allows us to concisely define many other important data structures. Besides lists, we also have proofs in Coq with trees, graphs, and stacks [16]. These data structures are all used for proving properties about the Schorr-Waite algorithm. In the next section we go into more detail about these data structures and how they are used in proving the Schorr-Waite algorithm.

4.6 Schorr-Waite

Our experiments so far demonstrate that our coinductive verification approach applies across languages in different paradigms, and can handle usual heap programs with a high degree of automation. Here we show that we can also handle the famous Schorr-Waite graph marking algorithm [25], which is a well-known verification challenge, "The Schorr-Waite algorithm is the first mountain that any formalism for pointer aliasing should climb" [26]. To give the reader a feel for what it takes to mechanically verify such an algorithm, previous proofs in [27] and [28] required manually produced proof scripts of about 470 and, respectively, over 1400 lines and they both used conventional Hoare logic. In comparison our proof is 514 lines. Line counts are a crude measure, but we can at least conclude that the language independence and generality of our approach did not impose any great cost compared to using language-specific program logics.

The version of Schorr-Waite that we verified is based on [29]. First, however, we verify a simpler property of the algorithm, showing that the given code correctly marks a tree, in the absence of sharing or cycles. Then we prove the same

code works on general graphs by considering the tree resulting from a depth first traversal. We define graphs by extending the definition of trees to allow a child of a node in an abstract tree to be a reference back to some existing node, in addition to an explicit subtree or a null pointer for a leaf. To specify that graph nodes are at their original addresses after marking, we include an address along with the mark flag in the abstract data structure in the pattern

$$\begin{aligned}
&\mathrm{grph}(\mathrm{leaf}, m, p') = \langle p' = 0 \rangle \\
&\mathrm{grph}(\mathrm{backref}(p), m, p') = \langle p' = p \rangle \\
&\mathrm{grph}(\mathrm{node}(p, l, r), m, p') = \langle p' {=} p \rangle * \exists p_l, p_r \,. \\
&\qquad p \mapsto \{m, p_l, p_r\} * \mathrm{grph}(l, m, p_l) * \mathrm{grph}(r, m, p_r)
\end{aligned}$$

The overall specification is $\mathit{call}(\mathit{Mark}, [p], \mathrm{grph}(G, 0, p), \lambda r.\mathrm{grph}(G, 3, p))$.

To describe the intermediate states in the algorithm, including the clever pointer-reversal trick used to encode a stack, we define another data structure for the context, in zipper style. A position into a tree is described by its immediate context, which is either the topmost context, or the point immediately left or right of a sibling tree, in a parent context. These are represented by nodes with intermediate values of the mark field, with one field pointing to the sibling subtree and the other pointing to the representation of the rest of the context.

$$\begin{aligned}
\mathrm{stack}(\mathrm{Top}, p) &= \langle p = 0 \rangle \\
\mathrm{stack}(\mathrm{LeftOf}(r, k), p) &= \exists p_r, p_k \,.\; p \mapsto \{1, p_r, p_k\} * \mathrm{grph}(r, 0, p_r) * \mathrm{stack}(k, p_k) \\
\mathrm{stack}(\mathrm{RightOf}(l, k), p) &= \exists p_l, p_k \,.\; p \mapsto \{2, p_k, p_l\} * \mathrm{stack}(k, p_k) * \mathrm{grph}(l, 3, p_l)
\end{aligned}$$

This is the second data structure needed to specify the main loop. When it is entered, there are only two live local variables, one pointing to the next address to visit and the other keeping context. The next node can either be the root of an unmarked subtree, with the context as stack, or the first node in the implicit stack when ascending after marking a tree, with the context pointing to the node that was just finished. For simplicity, we write a separate claim for each case.

$$\begin{aligned}
&\mathit{stmt}(\mathit{Loop}, (\mathtt{p} \mapsto p, \mathtt{q} \mapsto q), (\mathrm{grph}(G, 0, p) * \mathrm{stack}(S, q)), \lambda r.\mathrm{grph}(\mathit{plug}(G, S), 3)) \\
&\mathit{stmt}(\mathit{Loop}, (\mathtt{p} \mapsto p, \mathtt{q} \mapsto q), (\mathrm{stack}(S, p) * \mathrm{grph}(G, 3, q)), \lambda r.\mathrm{grph}(\mathit{plug}(G, S), 3))
\end{aligned}$$

The application of all the semantic steps was handled entirely automatically, the manual proof effort being entirely concerned with reasoning about the predicates above, for which no proof automation was developed.

4.7 Divergence

Our coinductive framework can also be used to verify a program is divergent. Such verification is often a topic that is given its own treatment, as in [30,31], though in our framework, no additional care is needed. To prove a program is divergent on all inputs, one verifies a set of claims of the form $(c, \emptyset)$, so that no

configuration can be determined valid by membership in the final set of states. We have verified the divergence of a simple program under each style of IMP semantics in Fig. 3, as well as programs in each language from Sect. 4.1. These program include the omega combinator and the `sum` program from Sect. 3.2 with `true` replacing the loop guard.

4.8 Summary of Experiments

Statistics are shown in Table 2. For each example, size shows the amount of code to be verified, the size of the specification, and the size of the proof script. If verifying an example required auxiliary definitions or lemmas specific to that example, the size of those definitions were counted with the specification or proof. Many examples were verified by a single invocation of our automatic proof tactic, giving 1-line proofs. Other small proofs required human assistance only in the form of applying lemmas about the domain. Proofs are generally smaller than the specifications, which are usually about as large as the code. This is similar to the results for Bedrock [32], and good for a foundational verification system.

Table 2. Proof statistics

	Size (lines)			Time (s)	
Example	Code	Spec	Proof	Prove	Check
Simple					
undefined	2	3	1	2.1	1.1
average3	2	5	1	2.3	0.8
min	3	4	2	2.1	0.7
max	3	4	2	2.1	0.7
multiply	9	6	1	7.2	1.4
sum(rec)	6	7	6	4.2	1.0
sum(iter)	6	11	8	6.0	1.0
Trees					
height	8	3	3	20.5	4.1
size	5	3	1	8.0	2.2
find	6	9	1	15.5	3.1
mirror	7	6	1	19.0	4.2
dealloc	15	7	1	19.6	4.1
flatten(rec)	12	10	1	30.9	6.8
flatten(iter)	24	17	4	150.3	22.8

	Size (lines)			Time (s)	
Example	Code	Spec	Proof	Prove	Check
Lists: head	2	4	1	2.1	0.8
tail	2	4	1	2.2	0.9
add	4	4	1	4.8	1.2
swap	6	4	1	19.6	3.6
dealloc	6	4	1	6.3	1.3
length(rec)	4	4	1	4.8	1.4
length(iter)	4	8	1	7.2	1.5
sum(rec)	4	4	1	8.2	2.0
sum(iter)	4	8	1	9.11	1.7
reverse	8	5	3	15.0	2.2
append	7	9	3	19.4	3.6
copy	14	11	3	55.0	9.3
delete	16	18	9	44.6	6.0
Schorr-Waite					
tree	14	91	116	60.1	7.6
graph	14	91	203	133.6	18.2

The reported "Proof" time is the time for Coq to process the proof script, which includes running proof tactics and proof searches to construct a complete proof. If this run succeeds, it produces a proof certificate file which can be rechecked without that overhead. For an initial comparison with Bedrock we timed their `SinglyLinkedList.v` example, which verifies length, reverse,

and append functions that closely resemble our example code. The total time to run the Bedrock proof script was 93 s, and 31 s to recheck the proof certificate, distinctly slower than our times in Table 2. To more precisely match the Bedrock examples we modified our programs to represent lists nodes with fields at successive addresses rather than using HIMP's records, but this only improved performance, down to 20 s to run the proof scripts, and 4 s to check the certificates.

5 Subsuming Reachability Logic

Reachability logic [33] is a closely related approach to program verification using operational semantics. In fact, our coinductive approach came about when trying to distill reachability logic into its mathematical essence. The practicality of reachability logic has recently been demonstrated, as the reachability logic proof system has been shown to work with several independently developed semantics of real-world languages, such as C, Java, and JavaScript [15].

5.1 Advantages of Coinduction

A mechanical proof of our soundness theorem gives a more usable verification framework, since reachability logic requires operational semantics to be given as a set of rewrite rules, while our approach does not. Further, reachability logic fixes a set of syntactic proof rules, while in our approach the mathematical fixpoints and functions act as proof rules without explicitly requiring any. In fact, the generality of our approach allows introductions of other derived rules that do not compromise the soundness result. Similarly, the generality allows higher-order verification, which reachability logic cannot handle.

Further, we saw in Sect. 3 that the general proof of our theorem is entirely mathematical. We instantiate it with the step_R function to get a program verification framework. However, if we instantiate it with other functions, we could get frameworks for proving different properties, such as all-path validity or the "until" notion of validity previously mentioned. Reachability logic does not support any other notion of validity without changes to its proof system, which then require new proofs of soundness and relative

Axiom :

$$\frac{\varphi \Rightarrow \varphi' \ \in \mathcal{A}}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'}$$

Reflexivity :

$$\mathcal{A} \vdash \varphi \Rightarrow \varphi$$

Transitivity :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^{+} \varphi_2 \qquad \mathcal{A} \cup \mathcal{C} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi_3}$$

Logic Framing :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi' \qquad \psi \text{ is a FOL formula}}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

Consequence :

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \quad \mathcal{A} \vdash_{\mathcal{C}} \varphi_1' \Rightarrow \varphi_2' \quad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi_2}$$

Case Analysis :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash_{\mathcal{C}} \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

Abstraction :

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi' \qquad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{A} \vdash_{\mathcal{C}} \exists X \ \varphi \Rightarrow \varphi'}$$

Circularity :

$$\frac{\mathcal{A} \vdash_{\mathcal{C} \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'}$$

Fig. 6. Reachability Logic proof system. Sequent $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ is a shorthand for $\mathcal{A} \vdash_{\emptyset} \varphi \Rightarrow \varphi'$.

completeness. For our framework, the proof of the main theorem does not need to be modified at all, and one only needs to prove that all-path validity is a greatest fixpoint (see Sect. 3). The same is true for any property. In this sense, this coinduction framework is much more general than the reachability logic proof system presented in [34].

5.2 Reachability Logic Proof System

The key construct in reachability logic is the notion of circularity. Circularities, represented as $\mathcal{C}$ in Fig. 6, intuitively represent claims that are conjectured to be true but have not yet been proved true. These claims are proved using the Circularity rule, which is analogous in our coinductive framework to referring back to claims previously seen. Most of the other rules in Fig. 6 are not as interesting. Transitivity requires progress before the circularities are flushed as axioms. This corresponds to the outer step_R in our coinductive framework.

Clearly, there are obvious parallels between the Reachability Logic proof system and our coinductive framework. We have formalized and mechanically verified a detailed proof that reachability logic is an instance of our coinductive verification framework. One can refer to [16] for full details, but we briefly discuss the nature of the proof below.

5.3 Reachability Logic is Coinduction

To formalize what it means for reachability logic to be an instance of coinduction, we first need some definitions. First, we need a translation from a reachability rule to a set of coinductive claims. In a reachability rule $\varphi \Rightarrow \varphi'$, both φ and φ' are patterns which respectively describe (symbolically) the starting and the reached configurations. Both φ and φ' can have free variables. Let Var be the set of variables. Then, we define the set of claims

$$S_{\varphi \Rightarrow \varphi'} \equiv \{(c, \overline{\rho}(\varphi')) \mid c \in \overline{\rho}(\varphi),\ \forall \rho : Var \to Cfg\}$$

where Cfg is the model of configurations and $\overline{\rho}(\cdot)$ is the extension of the valuation ρ to patterns [15]. Also, let the claims derived from a set of reachability rules $X = \{\varphi_1 \Rightarrow \varphi'_1, \ldots, \varphi_n \Rightarrow \varphi'_n\}$ be:

$$\overline{X} \equiv \bigcup_{\varphi_i \Rightarrow \varphi'_i \in X} S_{\varphi_i \Rightarrow \varphi'_i}$$

In reachability logic, programming language semantics are defined as *theories*, that is, as sets of (one-step) reachability rules $\mathcal{A}$ with patterns over a given signature of symbols. Each theory $\mathcal{A}$ defines a transition relation over the configurations in Cfg, say $R_{\mathcal{A}}$, which is then used to define the semantic validity in reachability logic, $\mathcal{A} \models \varphi \Rightarrow \varphi'$. It is possible and easier to prove our main theorem more generally, for any transition relation R that satisfies $R \vDash^+ \mathcal{A}$:

$$R \vDash^+ \mathcal{A} \text{ if } R \vDash^+ \varphi \Rightarrow \varphi' \text{ for each } \varphi \Rightarrow \varphi' \in \mathcal{A}$$

where $R \vDash^{+} \varphi \Rightarrow \varphi'$ if for each $\rho : Var \to Cfg$ and $\gamma : Cfg$ such that $(\rho, \gamma) \vDash \varphi$ [33], there is a γ' such that $\gamma \to_R \gamma'$ and $(\gamma', \overline{\rho}(\varphi'))$ is a valid reachability claim.

Lemma 3. *$R_{\mathcal{A}} \vDash^{+} \mathcal{A}$ and if $S_{\varphi \Rightarrow \varphi'} \subseteq \text{valid}_{R_{\mathcal{A}}}$ then $\mathcal{A} \vDash \varphi \Rightarrow \varphi'$.*

This lemma suggests what to do: take any reachability logic proof of $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ and any transition relation R such that $R \vDash^{+} \mathcal{A}$, and produce a coinductive proof of $S_{\varphi \Rightarrow \varphi'} \subseteq \text{valid}_R$. This gives us not only a procedure to associate coinductive proofs to reachability logic proofs, but also an alternative method to prove the soundness of reachability logic. This is what we do below:

Theorem 2. *If there is a reachability logic proof derivation for $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ and a transition relation R such that $R \vDash^{+} \mathcal{A}$, then $S_{\varphi \Rightarrow \varphi'} \subseteq \text{valid}_R$, and in particular this holds by applying Theorem 1 to an inclusion $\overline{\mathcal{C}} \subseteq \text{step}_R(\text{derived}_R^*(\overline{\mathcal{C}}))$. Here, derived_R is a particular function satisfying the conditions for G in Theorem 1 (see [16] for more details), and $\mathcal{C}$ is a set of reachability rules consisting of $\varphi \Rightarrow \varphi'$ along with those reachability rules which appear as conclusions of instances of the Circularity proof rule in the proof tree of $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$.*

To prove Theorem 2, we apply the Set Circularity theorem of reachability logic [35], which states that any reachability logic claim $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ is provable iff there is some set of claims $\mathcal{C}$ such that $\varphi \Rightarrow \varphi' \in \mathcal{C}$ and for each $\varphi_i \Rightarrow \varphi_i' \in \mathcal{C}$ there is a proof of $\mathcal{A} \vdash_{\mathcal{C}} \varphi_i \Rightarrow \varphi_i'$ which does not use the Circularity proof rule. In the forward direction, we can take $\mathcal{C}$ as defined in the statement of Theorem 2. The main idea is to convert proof trees into inclusions of sets of claims:

Lemma 4. *Given a proof derivation of $\mathcal{A} \vdash_{\mathcal{C}} \varphi_a \Rightarrow \varphi_b$ which does not use the Circularity proof rule (last rule in Fig. 6), if $R \vDash^{+} \mathcal{A}$ and $\mathcal{C}$ is nonempty then $S_{\varphi_a \Rightarrow \varphi_b} \subseteq \text{step}_R(\text{derived}_R^*(\overline{\mathcal{C}}))$.*

This lemma is proven by strengthening the inclusion into one that can be proven by structural induction over the Reachability Logic proof rules besides Circularity.

Combining this lemma with Set Circularity shows that $\overline{\mathcal{C}} = \cup_i S_{\varphi_i \Rightarrow \varphi_i'} \subseteq \text{valid}_R$ which implies that $S_{\varphi \Rightarrow \varphi'} \subseteq \text{valid}_R$ exactly as desired. We have mechanized the proofs of Lemmas 3 and 4 in Coq [16]. This is a major result, constituting an independent soundness proof for Reachability Logic, and helps demonstrate the strength of our coinductive framework, despite its simplicity. Moreover, this allows proofs done using reachability logic as in [15] to be translated to mechanically verified proofs in Coq, immediately allowing foundational verification of programs written in *any language*.

6 Other Related Work

Here we discuss work other than reachability logic that is related to our coinductive verification system. We discuss commonly used program verifiers, including approaches based on operational semantics and Iris [36], an approach with some language independence. We also discuss related coinduction schemata.

6.1 Current Verification Tools

A number of prominent tools such as Why [37], Boogie [38,39], and Bedrock [24,32] provide program verification for a fixed language, and support other languages by translation if at all. For example, Frama-C and Krakatoa, respectively, attempt to verify C and Java by translation through Why. Also, Spec# and Havoc, respectively, verify C# and C by translation through Boogie. We are not aware of soundness proofs for these translations. Such proofs would be highly non-trivial, requiring formal semantics of both source and target languages.

All of these systems are based on a verification condition (VC) generator for their programming language. Bedrock is closest in architecture and guarantees to our system, as it is implemented in Coq and verification results in a Coq proof certificate that the specification is sound with respect to a semantics of the object language. Bedrock supports dynamically created code, and modular verification of higher-order functions, for which our framework has preliminary support. Bedrock also makes more aggressive attempts at complete automation, which costs increased runtime. Most fundamentally, Bedrock is built around a VC generator for a fixed target language.

In sharp contrast to the above approaches, we demonstrated that a small-step operational semantics suffices for program verification, without a need to define any other semantics, or verification condition generators, for the same language. A language-independent, sound and (relatively) complete coinductive proof method then allows us to verify properties of programs using directly the operational semantics. As seen in Sect. 4.8 this language independence does not compromise other desirable properties. The required human effort and the performance of the verification task compare well with foundational program verifiers such as Bedrock, and we provide the same high confidence in correctness: the trust base consists of the operational semantics only.

6.2 Operational Semantics Based Approaches

Verifiable C [40] is a program verification tool for the C programming language based on an operational semantics for C defined in Coq. Hoare triples are then proved as lemmas about the operational semantics. However, in this approach and other similar approaches, it is necessary to prove such lemmas. Without them, verification of any nontrivial C program would be nearly impossible. In our approach, while we can also define and prove Hoare triples as lemmas, doing so is not needed to make program verification feasible, as demonstrated in the previous sections. We only need some additional domain reasoning in Coq, which logics like Verifiable C require *in addition* to Hoare logic reasoning. Thus, our approach automatically yields a program verification tool for any language with minimal additional reasoning, while approaches such as Verifiable C need over 40,000 lines of Coq to define the program logic. We believe this is completely unnecessary, and hope our coinductive framework will be the first step in eliminating such superfluous logics.

The work by the FLINT group [41–43] is another approach to program verification based on operational semantics. Languages developed use shallowly embedded state predicates in Coq, and inference rules are derived directly from the operational semantics. However, their work is not generic over operational semantics. For example, [43] is developed in the context of a particular machine model, with a fixed memory representation and register file. Even simple changes such as adding registers require updating soundness proofs. Our approach has a single soundness theorem that can be instantiated for *any* language.

Iris [36] is a concurrent separation logic that has language independence, with operational semantics formalized in Coq. Iris adds monoids and invariants to the program logic in order to facilitate verification. It also derives some Hoare-style rules for verification from the semantics of a language. However, there are still structural Hoare rules that depend on the language that must be added manually. Additionally, once proof rules are generated, they are specialized to that particular language. Further, the verification in the paper relies on Hoare style reasoning, while in our approach, we do not assume any such verification style, as we work directly with the mathematical specifications. Finally, the monoids used are not generated and are specific to the program language used.

6.3 Other Coinduction Schemata

A categorical generalization of our key theorem was presented as a recursion scheme in [12,13]. The titular result of the former is the dual of the λ-coiteration scheme of the latter, which specializes to preorder categories to give our Theorem 1. A more recent and more general result is [14], which also generalized other recent work on coinductive proofs such as [44]. Unlike these approaches, which were presented for showing bisimilarity, the novelty of our approach stems in the use of these techniques directly to show Hoare-style functional correctness claims, and in the development of the afferent machinery and automation that makes it work with a variety of languages, and not in advancing the already solid mathematical foundations of coinduction. Various weaker coinduction schemes are folklore, such as Isabelle/HOL's standard library's lemma `coinduct3`: $mono(f) \wedge A \subseteq f(\mu x.\, f(x) \cup A \cup \nu f) \implies A \subseteq \nu(f)$.

7 Conclusion and Future Work

We presented a language-independent program verification framework. Proofs can be as simple as with a custom Hoare logic, but only an operational semantics of the target language is required. We have mechanized a proof of the correctness of our approach in Coq. Combining this with a coinductive proof thus produces a Coq proof certificate concluding that the program meets the specification according to the provided semantics. Our approach is amenable to proof automation. Further automation may improve convenience and cannot compromise soundness of the proof system. A language designer need only give an authoritative

semantics to enable program verification for a new language, rather than needing to have the experience and invest the effort to design and prove the soundness of a custom program logic.

One opportunity for future work is using our approach to provide proof certificates for reachability logic program verifiers such as K [9]. The K prover was used to verify programs in several real programming languages [15]. While the proof system is sound, trusting the results of these tools requires trusting the implementation of the K system. Our translation in Sect. 5 will allow us to produce proof objects in Coq for proofs done in K's backend, which will make it sufficient to trust only Coq's proof checker to rely on the results from K's prover.

Another area for future work is verifying programs with higher-order specifications, where a specification can make reachability claims about values quantified over in the specification. This allows higher-order functions to have specifications that require functional arguments to themselves satisfy some specification. We have begun preliminary work on proving validity of such specifications using the notions of compatibility up-to presented in [14]. Combining this with more general forms of claims may allow modular verification of concurrent programs, as in RGsep [45]. See [16] for initial work in these areas.

Other areas for future work are evaluating the reusability of proof automation between languages, and using the ability to easily verify programs under a modified semantics, e.g. adding time costs to allow proving real-time properties.

References

1. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
2. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of C. In: PLDI, pp. 336–345. ACM (2015). https://doi.org/10.1145/2737924.2737979
3. Bogdănaş, D., Roşu, G.: K-Java: a complete semantics of Java. In: POPL, pp. 445–456. ACM (2015). https://doi.org/10.1145/2676726.2676982
4. Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised JavaScript specification. In: POPL, pp. 87–100. ACM (2014). https://doi.org/10.1145/2535838.2535876
5. Park, D., Ştefănescu, A., Roşu, G.: KJS: a complete formal semantics of Javascript. In: PLDI, pp. 346–356. ACM (2015). https://doi.org/10.1145/2737924.2737991
6. Politz, J.G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., Krishnamurthi, S.: Python: the full monty. In: OOPSLA, pp. 217–232. ACM (2013). https://doi.org/10.1145/2509136.2509536
7. Filaretti, D., Maffeis, S.: An executable formal semantics of PHP. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 567–592. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_23
8. Owens, S.: A sound semantics for OCaml$_{light}$. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 1–15. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78739-6_1
9. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. LAP **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012

10. Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J.A., Rafkind, J., Tobin-Hochstadt, S., Findler, R.B.: Run your research: on the effectiveness of lightweight mechanization. In: POPL, pp. 285–296. ACM (2012). https://doi.org/10.1145/2103656.2103691
11. Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strnisa, R.: Ott: effective tool support for the working semanticist. In: ICFP. ACM (2007). https://doi.org/10.1017/S0956796809990293
12. Uustalu, T., Vene, V., Pardo, A.: Recursion schemes from comonads. Nord. J. Comput. **8**(3), 366–390 (2001)
13. Bartels, F.: On generalised coinduction and probabilistic specification formats: distributive laws in coalgebraic modelling. Ph.D. thesis, Vrije Universiteit Amsterdam (2004)
14. Pous, D.: Coinduction all the way up. In: LICS, pp. 307–316. IEEE (2016). https://doi.org/10.1145/2933575.2934564
15. Ştefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics-based program verifiers for all languages. In: OOPSLA, pp. 74–91. ACM (2016). https://doi.org/10.1145/2983990.2984027
16. Moore, B., Peña, L., Rosu, G.: GitHub repository (2017). https://github.com/Formal-Systems-Laboratory/coinduction. Source code
17. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1), 38–94 (1992). https://doi.org/10.1006/inco.1994.1093
18. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebraic Program. **60–61**, 17–139 (2004). https://doi.org/10.1016/j.jlap.2004.05.001
19. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE (2002). https://doi.org/10.1109/LICS.2002.1029817
20. O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. Bull. Symbolic Log. **5**(2), 215–244 (1999). https://doi.org/10.2307/421090
21. Felleisen, M., Friedman, D.P.: A calculus for assignments in higher-order languages. In: POPL, p. 314. ACM (1987). https://doi.org/10.1145/41625.41654
22. Felleisen, M.: The calculi of Lambda-ν-cs conversion: a syntactic theory of control and state in imperative higher-order programming languages. Ph.D. thesis, Indiana University (1987)
23. Roşu, G.: Matching logic – extended abstract. In: RTA, LIPIcs, pp. 5–21. Schloss Dagstuhl-LZ I (2015). https://doi.org/10.4230/LIPIcs.RTA.2015.5
24. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: PLDI, pp. 234–245. ACM (2011). https://doi.org/10.1145/1993498.1993526
25. Schorr, H., Waite, W.M.: An efficient machine-independent procedure for garbage collection in various list structures. Commun. ACM **10**(8), 501–506 (1967). https://doi.org/10.1145/363534.363554
26. Bornat, R.: Proving pointer programs in Hoare logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000). https://doi.org/10.1007/10722010_8
27. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Inf. Comput. **199**(1–2), 200–227 (2005). https://doi.org/10.1016/j.ic.2004.10.007
28. Hubert, T., Marche, C.: A case study of C source code verification: the Schorr-Waite algorithm. In: SEFM, pp. 190–199. IEEE (2005). https://doi.org/10.1109/SEFM.2005.1
29. Gries, D.: The Schorr-Waite graph marking algorithm. Acta Informatica **11**(3), 223–232 (1979). https://doi.org/10.1007/BF00289068

30. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL, pp. 147–158. ACM (2008). https://doi.org/10.1145/1328438.1328459
31. Chen, H.-Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.: Proving nontermination via safety. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 156–171. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_11
32. Chlipala, A.: The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In: ICFP, pp. 391–402. ACM (2013). https://doi.org/10.1145/2500365.2500592
33. Roşu, G., Ştefănescu, A., Ciobâcă, Ş., Moore, B.M.: One-path reachability logic. In: LICS, pp. 358–367. IEEE (2013). https://doi.org/10.1109/LICS.2013.42
34. Roşu, G., Ştefănescu, A.: From Hoare logic to matching logic reachability. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 387–402. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_32
35. Roşu, G., Ştefănescu, A., Ciobâcă, c., Moore, B.M.: Reachability logic. Technical report, University of Illinois, July 2012. http://hdl.handle.net/2142/32952
36. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL, pp. 637–650. ACM (2015). https://doi.org/10.1145/2775051.2676980
37. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
38. Leino, K.R.M.: This is Boogie 2. Technical report, Microsoft Research, June 2008
39. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
40. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: Program Logics for Certified Compilers. Cambridge University Press, New York (2014)
41. Yu, D., Shao, Z.: Verification of safety properties for concurrent assembly code. In: ICFP, pp. 175–188. ACM (2004). https://doi.org/10.1145/1016850.1016875
42. Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. In: PLDI, pp. 401–414. ACM (2006). https://doi.org/10.1145/1133981.1134028
43. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Combining domain-specific and foundational logics to verify complete software systems. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 54–69. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87873-5_8
44. Hur, C.K., Neis, G., Dreyer, D., Vafeiadis, V.: The power of parameterization in coinductive proof. In: POPL, pp. 193–206. ACM (2013)
45. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge (2008)

Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq

Vincent Rahli(✉), Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo

SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
{vincent.rahli,ivana.vukotic,marcus.voelp,paulo.verissimo}@uni.lu

Abstract. Our increasing dependence on complex and critical information infrastructures and the emerging threat of sophisticated attacks, ask for extended efforts to ensure the correctness and security of these systems. Byzantine fault-tolerant state-machine replication (BFT-SMR) provides a way to harden such systems. It ensures that they maintain correctness and availability in an application-agnostic way, provided that the replication protocol is correct and at least $n - f$ out of n replicas survive arbitrary faults. This paper presents Velisarios, a logic-of-events based framework implemented in Coq, which we developed to implement and reason about BFT-SMR protocols. As a case study, we present the first machine-checked proof of a crucial safety property of an implementation of the area's reference protocol: PBFT.

Keywords: Byzantine faults · State machine replication
Formal verification · Coq

1 Introduction

Critical information infrastructures such as the power grid or water supply systems assume an unprecedented role in our society. On one hand, our lives depend on the correctness of these systems. On the other hand, their complexity has grown beyond manageability. One state of the art technique to harden such critical systems is Byzantine fault-tolerant state-machine replication (BFT-SMR). It is a generic technique that is used to turn any service into one that can tolerate *arbitrary* faults, by extensively replicating the service to mask the behavior of a minority of possibly faulty replicas behind a majority of healthy replicas, operating in consensus.[1] The total number of replicas n is a parameter over the maximum number of faulty replicas f, which the system is configured to tolerate

This work is partially supported by the Fonds National de la Recherche Luxembourg (FNR) through PEARL grant FNR/P14/8149128.

[1] For such techniques to be useful and in order to avoid persistent and shared vulnerabilities, replicas need to be rejuvenated periodically [17,76], they need to be diverse enough [43], and ideally they need to be physically far apart. Diversity and rejuvenation are not covered here.

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 619–650, 2018.
https://doi.org/10.1007/978-3-319-89884-1_22

at any point in time. Typically, $n = 3f + 1$ for classical protocols such as in [16], and $n = 2f + 1$ for protocols that rely on tamper-proof components such as in [82]. Because such protocols tolerate arbitrary faults, a faulty replica is one that does not behave according to its specification. For example it can be one that is controlled by an attacker, or simply one that contains a bug.

Ideally, we should guarantee the correctness and security of such replicated and distributed, hardened systems to the highest standards known to mankind today. That is, the proof of their correctness should be checked by a machine and their model refined down to machine code. Unfortunately, as pointed out in [29], most distributed algorithms, including BFT protocols, are published in pseudo-code or, in the best case, a formal but not executable specification, leaving their safety and liveness questionable. Moreover, Lamport, Shostak, and Pease wrote about such programs: "We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm." [54]. Therefore, we focus here on developing a generic and extensible formal verification framework for systematically supporting the mechanical verification of BFT protocols and their implementations.[2]

Our framework provides, among other things, a model that captures the idea of arbitrary/Byzantine faults; a collection of standard assumptions to reason about systems with faulty components; proof tactics that capture common reasoning patterns; as well as a general library of distributed knowledge. All these parts can be reused to reason about any BFT protocol. For example, most BFT protocols share the same high-level structure (they essentially disseminate knowledge and vote on the knowledge they gathered), which we capture in our knowledge theory. We have successfully used this framework to prove a crucial safety property of an implementation of a complex BFT-SMR protocol called PBFT [14–16]. We handle all the functionalities of the base protocol, including garbage collection and view change, which are essential in practical protocols. Garbage collection is used to bound message logs and buffers. The view change procedure enables BFT protocols to make progress in case the *primary*—a distinguished replica used in some fault-tolerant protocols to coordinate votes—becomes faulty.

Contributions. Our contributions are as follows: (1) Section 3 presents Velisarios, our continuing effort towards a generic and extensible logic-of-events based framework for verifying implementations of BFT-SMR protocols using Coq [25]. (2) As discussed in Sect. 4, our framework relies on a library to reason about *distributed epistemic knowledge*. (3) We implemented Castro's landmark PBFT protocol, and proved its agreement safety property (see Sect. 5). (4) We implemented a runtime environment to run the OCaml code we extract from Coq (see Sect. 6). (5) We released Velisarios and our PBFT safety proof under an open source licence.[3]

[2] Ideally, both (1) the replication mechanism and (2) the instances of the replicated service should be verified. However, we focus here on (1), which has to be done only once, while (2) needs to be done for every service and for every replica instance.

[3] Available at: https://github.com/vrahli/Velisarios.

Why PBFT? We have chosen PBFT because several BFT-SMR protocols designed since then either use (part of) PBFT as one of their main building blocks, or are inspired by it, such as [6,8,26,45,46,82], to cite only a few. Therefore, a bug in PBFT could imply bugs in those protocols too. Castro provided a thorough study of PBFT: he described the protocol in [16], studied how to proactively rejuvenate replicas in [14], and provided a pen-and-paper proof of PBFT's safety in [15,17]. Even though we use a different model—Castro used I/O automata (see Sect. 7.1), while we use a logic-of-events model (see Sect. 3)—our mechanical proof builts on top of his pen-and-paper proof. One major difference is that here we verify actual running code, which we obtain thanks to Coq's extraction mechanism.

2 PBFT Recap

This section provides a rundown of PBFT [14–16], which we use as running example to illustrate our model of BFT-SMR protocols presented in Sect. 3.

2.1 Overview of the Protocol

We describe here the public-key based version of PBFT, for which Castro provides a formal pen-and-paper proof of its safety. PBFT is considered the first practical BFT-SMR protocol. Compared to its predecessors, it is more efficient and it does not rely on unrealistic assumptions. It works with asynchronous, unreliable networks (i.e., messages can be dropped, altered, delayed, duplicated, or delivered out of order), and it tolerates independent network failures. To achieve this, PBFT assumes strong cryptography in the form of collision-resistant digests, and an existentially unforgeable signature scheme. It supports any deterministic state machine. Each state machine replica maintains the service state and implements the service operations. Clients send requests to all replicas and await $f + 1$ matching replies from different replicas. PBFT ensures that healthy replicas execute the same operations in the same order.

To tolerate up to f faults, PBFT requires $|R| = 3f+1$ replicas. Replicas move trough a succession of configurations called *views*. In each view v, one replica ($p = v \ mod \ |R|$) assumes the role of *primary* and the others become *backups*. The primary coordinates the votes, i.e., it picks the order in which client requests are executed. When a backup suspects the primary to be faulty, it requests a view-change to select another replica as new primary.

Normal-Case. During normal-case operation, i.e., when the primary is not suspected to be faulty by a majority of replicas, clients send requests to be executed, which trigger agreement among the replicas. Various kinds of messages have to be sent among clients and replicas before a client knows its request has been executed. Figure 1 shows the resulting message patterns for PBFT's normal-case operation and view-change protocol. Let us discuss here normal-case operation:

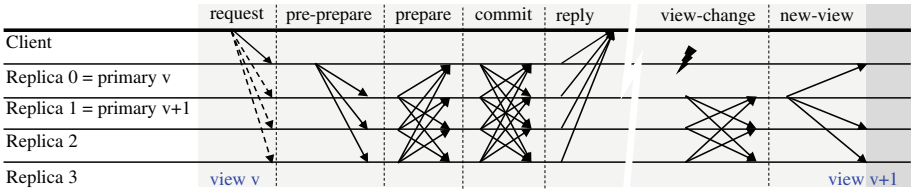


Fig. 1. PBFT normal-case (left) and view-change (right) operations

1. *Request:* To initiate agreement, a client c sends a request of the form $\langle \texttt{REQUEST}, o, t, c \rangle_{\sigma_c}$ to the primary, but is also prepared to broadcast it to all replicas if replies are late or primaries change. $\langle \texttt{REQUEST}, o, t, c \rangle_{\sigma_c}$ specifies the operation to execute o and a timestamp t that orders requests of the same client. Replicas will not re-execute requests with a lower timestamp than the last one processed for this client, but are prepared to resend recent replies.
2. *Pre-prepare:* The primary of view v puts the pending requests in a total order and initiates agreement by sending $\langle \texttt{PRE-PREPARE}, v, n, m \rangle_{\sigma_p}$ to all the backups, where m should be the n^{th} executed request. The strictly monotonically increasing and contiguous sequence number n ensures preservation of this order despite message reordering.
3. *Prepare:* Backup i acknowledges the receipt of a pre-prepare message by sending the digest d of the client's request in $\langle \texttt{PREPARE}, v, n, d, i \rangle_{\sigma_i}$ to all replicas.
4. *Commit:* Replica i acknowledges the reception of $2f$ prepares matching a valid pre-prepare by broadcasting $\langle \texttt{COMMIT}, v, n, d, i \rangle_{\sigma_i}$. In this case, we say that the message is *prepared* at i.
5. *Execution & Reply:* Replicas execute client operations after receiving $2f + 1$ matching commits, and follow the order of sequence numbers for this execution. Once replica i has executed the operation o requested by client c, it sends $\langle \texttt{REPLY}, v, t, c, i, r \rangle_{\sigma_i}$ to c, where r is the result of applying o to the service state. Client c accepts r if it receives $f+1$ matching replies from different replicas.

Client and replica authenticity, and message integrity are ensured through signatures of the form $\langle m \rangle_{\sigma_i}$. A replica accepts a message m only if: (1) m's signature is correct, (2) m's view number matches the current view, and (3) the sequence number of m is in the water mark interval (see below).

PBFT buffers pending client requests, processing them later in batches. Moreover, it makes use of checkpoints and water marks (which delimit sequence number intervals) to limit the size of all message logs and to prevent replicas from exhausting the sequence number space.

Garbage Collection. Replicas store all correct messages that were created or received in a log. Checkpoints are used to limit the number of logged messages by removing the ones that the protocol no longer needs. A replica starts checkpointing after executing a request with a sequence number divisible by some predefined constant, by multicasting the message $\langle \texttt{CHECKPOINT}, v, n, d, i \rangle_{\sigma_i}$ to all

other replicas. Here n is the sequence number of the last executed request and d is the digest of the state. Once a replica received $f + 1$ different checkpoint messages[4] (possibly including its own) for the same n and d, it holds a proof of correctness of the log corresponding to d, which includes messages up to sequence number n. The checkpoint is then called *stable* and all messages lower than n (except view-change messages) are pruned from the log.

View Change. The view change procedure ensures progress by allowing replicas to change the leader so as to not wait indefinitely for a faulty primary. Each backup starts a timer when it receives a request and stops it after the request has been executed. Expired timers cause the backup to suspect the leader and request a view change. It then stops receiving normal-case messages, and multicasts $\langle \texttt{VIEW-CHANGE}, v+1, n, s, C, P, i \rangle_{\sigma_i}$, reporting the sequence number n of the last stable checkpoint s, its proof of correctness C, and the set of messages P with sequence numbers greater than n that backup i prepared since then. When the new primary p receives $2f+1$ view-change messages, it multicasts $\langle \texttt{NEW-VIEW}, v+1, V, O, N \rangle_{\sigma_p}$, where V is the set of $2f + 1$ valid view-change messages that p received; O is the set of messages prepared since the latest checkpoint reported in V; and N contains only the special *null* request for which the execution is a no-op. N is added to the O set to ensure that there are no gaps between the sequence numbers of prepared messages sent by the new primary. Upon receiving this new-view message, replicas enter view $v + 1$ and re-execute the normal-case protocol for all messages in $O \cup N$.

We have proved a critical safety property of PBFT, including its garbage collection and view change procedures, which are essential in practical protocols. However, we have not yet developed generic abstractions to specifically reason about garbage collection and view changes, that can be reused in other protocols, which we leave as future work.

2.2 Properties

PBFT with $|R| = 3f+1$ replicas is safe and live. Its safety boils down to linearizability [42], i.e., the replicated service behaves like a centralized implementation that executes operations atomically one at a time. Castro used a modified version of linearizability in [14] to deal with faulty clients. As presented in Sect. 5, we proved the crux of this property, namely the agreement property (we leave linearizability for future work).

As informally explained by Castro [14], assuming weak synchrony (which constrains message transmission delays), PBFT is live, i.e., clients will eventually receive replies to their requests. In the future, we plan to extend Velisarios to support liveness and mechanize PBFT's liveness proof.

[4] Castro first required $2f + 1$ checkpoint messages [16] but relaxed this requirement in [14].

2.3 Differences with Castro's Implementation

As mentioned above, besides the normal-case operation, our Coq implementation of PBFT handles garbage collection, view changes and request batching. However, we slightly deviated from Castro's implementation [14], primarily in the way checkpoints are handled: we always work around sending messages that are not between the water marks, and a replica always requires its own checkpoint before clearing its log. Assuming the reader is familiar with PBFT, we now detail these deviations and refer the reader to [14] for comparison.

(1) To the best of our knowledge, to ensure liveness, Castro's implementation requires replicas to resend prepare messages below the low water mark when adopting a new-view message and processing the pre-prepares in $O \cup N$. In contrast, our implementation never sends messages with sequence numbers lower than the low water mark. This liveness issue can be resolved by bringing late replicas up to date through a state transfer.
(2) We require a new leader to send its own view-change message updated with its latest checkpoint as part of its new-view message. If not, it may happen that a checkpoint stabilizes after the view-change message is sent and before the new-view message is prepared. This might result in a new leader sending messages in $O \cup N$ with a sequence number below its low water mark, which it avoids by updating its own view-change message to contain its latest checkpoint.
(3) We require replicas to wait for their own checkpoint message before stabilizing a checkpoint and garbage collecting logs. This avoids stabilizing a checkpoint that has not been computed locally. Otherwise, a replica could lose track of the last executed request if its sequence number is superseded by the one in the checkpoint. Once proven, a state transfer of the latest checkpoint state and an update of the last executed request would also resolve this point.

We slightly deviated from Castro's protocol to make our proofs go through. We leave it for future work to formally study whether we could do without these changes, or whether they are due to shortcomings of the original specification.

3 Velisarios Model

Using PBFT as a running example, we now present our Coq model for Byzantine fault-tolerant distributed systems, which relies on a logic of events—Fig. 2 outlines our formalization.

3.1 The Logic of Events

We adapt the Logic of Events (LoE) we used in EventML [9,11,71] to not only deal with crash faults, but arbitrary faults in general (including malicious

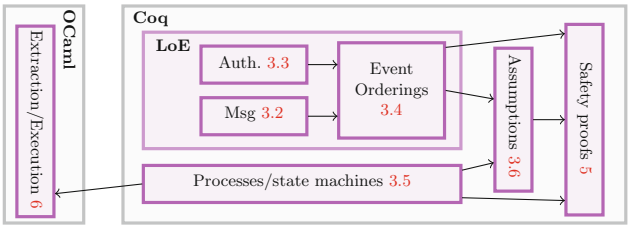


Fig. 2. Outline of formalization

faults). LoE, related to Lamport's notion of causal order [53] and to event structures [60,65], was developed to reason about events occurring in the execution of a distributed system. LoE has recently been used to verify consensus protocols [71,73] and cyber-physical systems [3]. Another standard model of distributed computing is Chandy and Lamport's *global state semantics* [19], where a distributed system is modeled as a single state machine: a state is the collection of all processes at a given time, and a transition takes a message in flight and delivers it to its recipient (a process in the collection). Each of these two models has advantages and disadvantages over the other. We chose LoE because in our experience it corresponds more closely to the way distributed system researchers and developers reason about protocols. As such, it provides a convenient communication medium between distributed systems and verification experts.

In LoE, an event is an abstract entity that corresponds either (1) to the handling of a received message, or (2) to some arbitrary activity about which no information is provided (see the discussion about trigger in Sect. 3.4). We use those arbitrary events to model arbitrary/Byzantine faults. An event happens at a specific point in space/time: the space coordinate of an event is called its location, and the time coordinate is given by a well-founded ordering on events that totally orders all events at the same location. Processes react to the messages that triggered the events happening at their locations one at a time, by transitioning through their states and creating messages to send out, which in turn might trigger other events. In order to reason about distributed systems, we use the notion of *event orderings* (see Sect. 3.4), which essentially are collections of ordered events and represent runs of a system. They are abstract entities that are never instantiated. Rather, when proving a property about a distributed system, one has to prove that the property holds for all event orderings corresponding to all possible runs of the system (see Sects. 3.5 and 5 for examples). Some runs/event orderings are not possible and therefore excluded through assumptions, such as the ones described in Sect. 3.6. For example, exists_at_most_f_faulty excludes event orderings where more than f out of n nodes could be faulty.

In the next few sections, we explain the different components (messages, authentication, event orderings, state machines, and correct traces) of Velisarios, and their use in our PBFT case study. Those components are parameterized by abstract types (parameters include the type of messages and the kind of authentication schemes), which we later have to instantiate in order to reason

about a given protocol, e.g. PBFT, and to obtain running code. The choices we made when designing Velisarios were driven by our goal to generate running code. For example, we model cryptographic primitives to reason about authentication.

3.2 Messages

Model. Some events are caused by messages of type msg, which is a parameter of our model. Processes react to messages to produce message/destinations pairs (of type DirectedMsg), called *directed messages*. A directed message is typically handled by a message outbox, which sends the message to the listed destinations.[5] A destination is the name (of type name, which is a parameter of our model) of a node participating in the protocol.

PBFT. In our PBFT implementation, we instantiate the msg type using the following datatype (we only show some of the normal-case operation messages, leaving out for example the more involved pre-prepare messages—see Sect. 2.1):

```
Inductive PBFTmsg :=                Inductive Bare_Prepare :=
| REQUEST (r : Request)             | bare_prepare (v : View) (n : SeqNum) (d : digest) (i : Rep).
| PREPARE (p : Prepare)             Inductive Prepare :=
| REPLY (r : Reply) ...             | prepare (b : Bare_Prepare) (a : list Token).
```

As for prepares, all messages are defined as follows: we first define bare messages that do not contain authentication tokens (see Sect. 3.3), and then authenticated messages as pairs of a bare message and an authentication token. Views and sequence numbers are nats, while digests are parameters of the specification. PBFT involves two types of nodes: replicas of the form PBFTreplica(r), where r is of type Rep; and clients of the form PBFTclient(c), where c is of type Client. Both Rep and Client are parameters of our formalization, such that Rep is of arity 3f+1, where f is a parameter that stands for the number of tolerated faults.

3.3 Authentication

Model. Our model relies on an abstract concept of keys, which we use to implement and reason about authenticated communication. Capturing authenticity at the level of keys allows us to talk about impersonation through key leakage. Keys are divided into *sending keys* (of type sending_key) to authenticate a message for a target node, and *receiving keys* (of type receiving_key) to check the validity of a received message. Both sending_key and receiving_key are parameters of our model.[6] Each node maintains *local keys* (of type local_keys), which consists of two lists of *directed keys*: one for sending keys and one for receiving keys. Directed keys are pairs of a key and a list of node names identifying the processes that the holder of the key can communicate with.

[5] Message inboxes/outboxes are part of the runtime environment but not part of the model.

[6] Sending and receiving keys must be different when using asymmetric cryptography, and can be the same when using symmetric cryptography.

Sending keys are used to create *authentication tokens* of type Token, which we use to authenticate messages. Tokens are parameters of our model and abstract away from concrete concepts such as digital signatures or MACs. Typically, a message consists of some data plus some tokens that authenticates the data. Therefore, we introduce the following parameters: (1) the type data, for the kind of data that can be authenticated; (2) a create function to authenticate some data by generating authentication tokens using the sending keys; and (3) a verify function to verify the authenticity of some data by checking that it corresponds to some token using the receiving keys.

Once some data has been authenticated, it is typically sent over the network to other nodes, which in turn need to check the authenticity of the data. Typically, when a process sends an authenticated message to another process it includes its identity somewhere in the message. This identity is used to select the corresponding receiving key to check the authenticity of the data using verify. To extract this claimed identity we require users to provide a data_sender function.

It often happens in practice that a message contains more than one piece of authenticated data (e.g., in PBFT, pre-prepare messages contain authenticated client requests). Therefore, we require users to provide a get_contained_auth_data function that extracts all authenticated pieces of data contained in a message. Because we sometimes want to use different tokens to authenticate some data (e.g., when using MACs), an authenticated piece of data of type auth_data is defined as a pair of: (1) a piece of data, and (2) a list of tokens.

PBFT. Our PBFT implementation leaves keys and authentication tokens abstract because our safety proof is agnostic to the kinds of these elements. However, we turn them into actual asymmetric keys when extracting OCaml code (see Sect. 6 for more details). The create and verify functions are also left abstract until we extract the code to OCaml. Finally, we instantiate the data (the objects that can be authenticated, i.e., bare messages here), data_sender, and get_contained_auth_data parameters using:

```
Inductive PBFTdata := | PBFTdata_request (r : Bare_Request)
  | PBFTdata_prepare (p : Bare_Prepare) | PBFTdata_reply (r : Bare_Reply) ...

Definition PBFTdata_sender (m : data) : option name := match m with
  | PBFTdata_request (bare_request o t c) ⇒ Some (PBFTclient c)
  | PBFTdata_prepare (bare_prepare v n d i) ⇒ Some (PBFTreplica i)
  | PBFTdata_reply (bare_reply v t c i r) ⇒ Some (PBFTreplica i) ...

Definition PBFTget_contained_auth_data (m : msg) : list auth_data := match m with
  | REQUEST (request b a) ⇒ [(PBFTdata_request b,a)]
  | PREPARE (prepare b a) ⇒ [(PBFTdata_prepare b,a)]
  | REPLY (reply b a) ⇒ [(PBFTdata_reply b,a)] ...
```

3.4 Event Orderings

A typical way to reason about a distributed system is to reason about its possible runs, which are sometimes modeled as execution traces [72], and which are captured in LoE using *event orderings*. An *event ordering* is an abstract representation of a run of a distributed system; it provides a formal definition of a *message sequence diagram* as used by system designers (see for example Fig. 1). As opposed to [72], a trace here is not just one sequence of events but instead can be seen as a collection of local traces (one local trace per sequential process), where a local trace is a collection of events all happening at the same location and ordered in time, and such that some events of different local traces are causally ordered. Event orderings are never instantiated. Instead, we express system properties as predicates on event orderings. A system satisfies such a property if every possible execution of the system satisfies the predicate. We first formally define the components of an event ordering, and then present the axioms that these components have to satisfy.

Components. An event ordering is formally defined as the tuple:[7]

```
Class EventOrdering :=
  { Event : Type;                          happenedBefore : Event → Event → Prop;
    loc : Event → name;                    direct_pred : Event → option Event;
    trigger : Event → option msg;          keys : Event → local_keys; }
```

where (1) Event is an abstract type of events; (2) happenedBefore is an ordering relation on events; (3) loc returns the location at which events happen; (4) direct_pred returns the direct local predecessor of an event when one exists, i.e., for all events except initial events; (5) given an event e, trigger either returns the message that triggered e, or it returns None to indicate that no information is available regarding the action that triggered the event (see below); (6) keys returns the keys a node can use at a given event to communicate with other nodes. The event orderings presented here are similar to the ones used in [3,71], which we adapted to handle Byzantine faults by modifying the type of trigger so that events can be triggered by arbitrary actions and not necessarily by the receipt of a message, and by adding support for authentication through keys.

The trigger function returns None to capture the fact that nodes can sometimes behave arbitrarily. This includes processes behaving correctly, i.e., according to their specifications; as well as (possibly malicious) processes deviating from their specifications. Note that this does not preclude from capturing the behavior of correct processes because for all event orderings where trigger returns None for an event where the node behaved correctly, there is a similar event ordering, where trigger returns the triggering message at that event. To model that at most f nodes out of n can be faulty we use the exists_at_most_f_faulty assumption, which enforces that trigger returns None at most f nodes.

Moreover, even though non-syntactically valid messages do not trigger events because they are discarded by message boxes, a triggering message could be

[7] A Coq type class is essentially a dependent record.

syntactically valid, but have an invalid signature. Therefore, it is up to the programmer to ensure that processes only react to messages with valid signatures using the verify function. Our authenticated_messages_were_sent_non_byz and exists_at_most_f_faulty assumptions presented in Sect. 3.6 are there to constrain trigger to ensure that at most f nodes out of n can diverge from their specifications, for example, by producing valid signatures even though they are not the nodes they claim to be (using leaked keys of other nodes).

Axioms. The following axioms characterize the behavior of these components:

1. Equality between events is decidable. Events are abstract entities that correspond to points in space/time that can be seen as pairs of numbers (one for the space coordinate and one for the time coordinate), for which equality is decidable.
2. The happened before relation is transitive and well-founded. This allows us to prove properties by induction on causal time. We assume here that it is not possible to infinitely go back in time, i.e., that there is a beginning of (causal) time, typically corresponding to the time a system started.
3. The direct predecessor e_2 of e_1 happens at the same location and before e_1. This makes local orderings sub-orderings of the happenedBefore ordering.
4. If an event e does not have a direct predecessor (i.e., e is an initial event) then there is no event happening locally before e.
5. The direct predecessor function is injective, i.e., two different events cannot have the same direct predecessor.
6. If an event e_1 happens locally before e_2 and e is the direct predecessor of e_2, then either $e = e_1$ or e_1 happens before e. From this, it follows that the direct predecessor function can give us the complete local history of an event.

Notation. We use $a \prec b$ to stand for (happenedBefore a b); $a \preceq b$ to stand for ($a \prec b$ or $a{=}b$); and $a \sqsubseteq b$ to stand for ($a \preceq b$ and loc a=loc b). We also sometimes write EO instead of EventOrdering.

Some functions take an event ordering as a parameter. For readability, we sometimes omit those when they can be inferred from the context. Similarly, we will often omit type declarations of the form (T : Type).

Correct Behavior. To prove properties about distributed systems, one only reasons about processes that have a correct behavior. To do so we only reason about events in event orderings that are correct in the sense that they were triggered by some message:

```
Definition isCorrect (e : Event) := match trigger e with Some m ⇒ True | None ⇒ False end.
Definition arbitrary (e : Event) := ∼ isCorrect e.
```

Next, we characterize correct replica histories as follows: (1) First we say that an event e has a correct trace if all local events prior to e are correct. (2) Then, we say that a node i has a correct trace before some event e, not necessarily happening at i, if all events happening before e at i have a correct trace:

```
Definition has_correct_bounded_trace (e : Event) := forall e', e' ⊑ e → isCorrect e'.
Definition has_correct_trace_before (e : Event) (i : name) :=
  forall e', e' ≺ e → loc e' = i → has_correct_bounded_trace e'.
```

3.5 Computational Model

Model. We now present our computational model, which we use when extracting OCaml programs. Unlike in EventML [71] where systems are first specified as *event observers* (abstract processes), and then later refined to executable code, we skip here event observers, and directly specify systems using executable state machines, which essentially consist of an update function and a current state. We define a system of distributed state machines as a function that maps names to state machines. Systems are parametrized by a function that associates state types with names in order to allow for different nodes to run different machines.

```
Definition Update S I O := S → I → (option S * O).
Record StateMachine S I O := MkSM { halted : bool; update : Update S I O; state : S }.
Definition System (F : name → Type) I O := forall (i : name), StateMachine (F i) I O.
```

where S is the type of the machine's state, I/O are the input/output types, and halted indicates whether the state machine is still running or not.

Let us now discuss how we relate state machines and events. We define state_sm_before_event and state_sm_after_event that compute a machine's state before and after a given event e. These states are computed by extracting the local history of events up to e using direct_pred, and then updating the state machine by running it on the triggering messages of those events. These functions return None if some arbitrary event occurs or the machine halts sometime along the way. Otherwise they return Some s, where s is the state of the machine updated according to the events. Therefore, assuming they return Some amounts to assuming that all events prior to e are correct, i.e., we can prove that if state_sm_after_event sm e = Some s then has_correct_trace_before e (loc e). As illustrated below, we use these functions to adopt a Hoare-like reasoning style by stating pre/post-conditions on the state of a process prior and after some event.

PBFT. We implement PBFT replicas as state machines, which we derive from an update function that dispatches input messages to the corresponding handlers. Finally, we define PBFTsys as the function that associates PBFTsm with replicas and a halted machine with clients (because we do not reason here about clients).

```
Definition PBFTupdate (i : Rep) := fun state msg ⇒ match msg with
  | REQUEST r ⇒ PBFThandle_request i state r
  | PREPARE p ⇒ PBFThandle_prepare i state p ...
Definition PBFTsm (i : Rep) := MkSM false (PBFTupdate i) (initial_state i).
Definition PBFTsys := fun name ⇒ match name with
  | PBFTreplica i ⇒ PBFTsm i | PBFTclient c ⇒ haltedSM end.
```

Let us illustrate how we reason about state machines through a simple example that shows that they maintain a view that only increases over time. It shows a local property, while Sect. 5 presents the distributed agreement property that makes use of the assumptions presented in Sect. 3.6. As mentioned above we prove such properties for all possible event orderings, which means that they are true for all possible runs of the system. In this lemma, *s1* is the state prior to the event *e*, and *s2* is the state after handling *e*. It does not have pre-conditions, and its post-condition states that the view in *s1* is smaller than the view in *s2*.

```
Lemma current_view_increases : forall (eo : EO) (e : Event) i s1 s2,
    state_sm_before_event (PBFTsm i) e = Some s1
    → state_sm_after_event (PPBFTsm i) e = Some s2
    → current_view s1 ≤ current_view s2.
```

3.6 Assumptions

Model. Let us now turn to the assumptions we make regarding the network and the behavior of correct and faulty nodes.

Assumption 1. Proving safety properties of crash fault-tolerant protocols that only require reasoning about past events, such as agreement, does not require reasoning about faults and faulty replicas. To prove such properties, one merely has to follow the causal chains of events back in time, and if a message is received by a node then it must have been sent by some node that had not crashed at that time. The state of affairs is different when dealing with Byzantine faults.

One issue it that Byzantine nodes can deviate from their specifications or impersonate other nodes. However, BFT protocols are designed in such a way that nodes only react to collections of messages, called *certificates*, that are larger than the number of faults. This means that there is always at least one correct node that can be used to track down causal chains of events.

A second issue is that, in general, we cannot assume that some received message was sent as such by the designated (correct) sender of the message because messages can be manipulated while in flight. As captured by the authenticated_messages_were_sent_or_byz predicate defined below,[8] we can only assume that the authenticated parts of the received message were actually sent by the designated senders, possibly inside larger messages, provided the senders did not leak their keys. As usual, we assume that attackers cannot break the cryptographic primitives, i.e., that they cannot authenticate messages without the proper keys [14].

```
1.Definition authenticated_messages_were_sent_or_byz (P : AbsProcess) :=
2. forall e (a : auth_data),
3.   In a (bind_op_list get_contained_auth_data (trigger e))
4.   → verify_auth_data (loc e) a (keys e) = true
```

[8] For readability, we show a slightly simplified version of this axiom. The full axiom can be found in https://github.com/vrahli/Velisarios/blob/master/model/EventOrdering.v.

```
5.  → exists e', e' ≺ e ∧ am_auth a = authenticate (am_data a) (keys e')
6.    ∧ ( (exists dst m,
7.           In a (get_contained_auth_data m) ∧ In (m,dst) (P eo e')
8.           ∧ data_sender (loc e) (am_data a) = Some (loc e'))
9.         ∨
10.        (exists e",
11.          e" ⪯ e' ∧ arbitrary e' ∧ arbitrary e" ∧ got_key_for (loc e) (keys e") (keys e')
12.          ∧ data_sender (loc e) (am_data a) = Some (loc e")) ).
```

This assumption says that if the authenticated piece of data a is part of the message that triggered some event e (L.3), and a is verified (L.4), then there exists a prior event e' such that the data was authenticated while handling e' using the keys available at that time (L.5). Moreover, (1) either the sender of the data was correct while handling e' and sent the data as part of a message following the process described by P (L.6–8); or (2) the node at which e' occurred was Byzantine at that time, and either it generated the data itself (e.g. when $e"$=e'), or it impersonated some other replica (by obtaining the keys that some node leaked at event $e"$) (L.10–12).

We used a few undefined abstractions in this predicate: An AbsProcess is an abstraction of a process, i.e., a function that returns the collection of messages generated while handling a given event: (`forall` (*eo* : EO) (*e* : Event), list DirectedMsg). The bind_op_list function is wrapped around get_contained_auth_data to handle the fact that trigger might return None, in which case bind_op_list returns nil. The verify_auth_data function takes an authenticated message a and some keys and: (1) invokes data_sender (defined in Sect. 3.3) to extract the expected sender s of a; (2) searches among its keys for a receiving_key that it can use to verify that s indeed authenticated a; and (3) finally verifies the authenticity of a using that key and the verify function. The authenticate function simply calls create and uses the sending keys to create tokens. The got_key_for function takes a name i and two local_keys *lk1* and *lk2*, and states that the sending keys for i in *lk1* are all included in *lk2*.

However, it turns out that because we never reason about faulty nodes, we never have to deal with the right disjunct of the above formula. Therefore, this assumption about received messages can be greatly simplified when we know that the sender is a correct replica, which is always the case when we use this assumption because BFT protocols as designed so that there is always a correct node that can be used to track down causal chains of events. We now define the following simpler assumption, which we have proved to be a consequence of authenticated_messages_were_sent_or_byz:

```
Definition authenticated_messages_were_sent_non_byz (P : AbsProcess) :=
  forall (e : Event) (a : auth_data) (c : name),
    In a (bind_op_list get_contained_auth_data (trigger e))
    → has_correct_trace_before e c
    → verify_auth_data (loc e) a (keys e) = true
    → data_sender (loc e) (am_data a) = Some c
    → exists e' dst m, e' ≺ e ∧ loc e' = c.
        ∧ am_auth a = authenticate (am_data a) (keys e')
        ∧ In a (get_contained_auth_data m)
        ∧ In (m,dst) (P eo e')
```

As opposed to the previous formula, this one assumes that the authenticated data was sent by a correct replica, which has a correct trace prior to the event e—the event when the message containing a was handled.

Assumption 2. Because processes need to store their keys to sign and verify messages, we must connect those keys to the ones in the model. We do this through the correct_keys assumption, which states that for each event e, if a process has a correct trace up to e, then the keys (keys e) from the model are the same as the ones stored in its state (which are computed using state_sm_before_event).

Assumption 3. Finally, we present our assumption regarding the number of faulty nodes. There are several ways to state that there can be at most f faulty nodes. One simple definition is (where node is a subset of name as discussed in Sect. 4.2):

```
Definition exists_at_most_f_faulty (E : list Event) (f : nat) :=
  exists (faulty : list node), length faulty ≤ f
    ∧ forall e1 e2, In e2 E → e1 ≼ e2 → ∼ In (loc e1) faulty
          → has_correct_bounded_trace e1.
```

This assumption says that at most f nodes can be faulty by stating that the events happening at nodes that are not in the list of faulty nodes *faulty*, of length f, are correct up to some point characterized by the partial cut E of a given event ordering (i.e., the collection of events happening before those in E).

PBFT Assumption 4. In addition to the ones above, we made further assumptions about PBFT. Replicas sometimes send message hashes instead of sending the entire messages. For example, pre-prepare messages contain client requests, but prepare and commit messages simply contain digests of client requests. Consequently, our PBFT formalization is parametrized by the following *create* and *verify* functions, and we assume that the create function is collision resistant:[9]

```
Class PBFThash := MkPBFThash {
  create_hash : list PBFTmsg → digest; verify_hash : list PBFTmsg → digest → bool; }.
Class PBFThash_axioms := MkPBFThash_axioms {
  create_hash_collision_resistant :
    forall msgs1 msgs2, create_hash msgs1 = create_hash msgs2 → msgs1 = msgs2; }.
```

The version of PBFT, called PBFT-PK in [14], that we implemented relies on digital signatures. However, we did not have to make any more assumptions regarding the cryptographic primitives than the ones presented above, and in particular we did not assume anything that is true about digital signatures and false about MACs. Therefore, our safety proof works when using either digital signatures or MAC vectors. As discussed below, this is true because we adapted the way messages are verified (we have not verified the MAC version of PBFT but a slight variant of PBFT-PK) and because we do not deal with liveness.

[9] Note that our current collision resistant assumption is too strong because it is always possible to find two distinct messages that are hashed to the same hash. We leave it to future work to turn it into a more realistic probabilistic assumption.

As Castro showed [14, Chap. 3], PBFT-PK has to be adapted when digital signatures are replaced by MAC vectors. Among other things, it requires "significant and subtle changes to the view change protocol" [14, Sect. 3.2]. Also, to the best of our knowledge, in PBFT-PK backups do not check the authenticity of requests upon receipt of pre-prepares. They only check the authenticity of requests before executing them [14, p. 42]. This works when using digital signatures but not when using MACs: one backup might not execute the request because its part of the MAC vector does not check out, while another backup executes the request because its part of the MAC vector checks out, which would lead to inconsistent states and break safety. Castro lists other problems related to liveness.

Instead, as in the MAC version of PBFT [14, p. 42], in our implementation we always check requests' validity when checking the validity of a pre-prepare. If we were to check the validity of requests only before executing them, we would have to assume that two correct replicas would either both be able to verify the data, or both would not be able to do so. This assumption holds for digital signatures but not for MAC vectors.

4 Methodology

Because distributed systems are all about exchanging information among nodes, we have developed a theory that captures abstractions and reasoning patterns to deal with knowledge dissemination (see Sect. 4.4). In the presence of faulty nodes, one has to ensure that this knowledge is reliable. Fault-tolerant state-machine replication protocols provide such guarantees by relying on certificates, which ensure that we can always get hold of a correct node to trace back information through the system. This requires reasoning about the past, i.e., reasoning by induction on causal time using the happenedBefore relation.

4.1 Automated Inductive Reasoning

We use induction on causal time to prove both distributed and local properties. As discussed here, we automated the typical reasoning pattern we use to prove local properties. As an example, in our PBFT formalization, we proved the following local property: if a replica has a prepare message in its log, then it either received or generated it. Moreover, as for any kinds of programs, using Velisarios we prove local properties about processes by reasoning about all possible paths they can take when reacting upon messages. Thus, a typical proof of such a lemma using Velisarios goes as follows: (1) we go by induction on events; (2) we split the code of a process into all possible execution paths; (3) we prune the paths that could not happen because they invalidate some hypotheses of the lemma being proved; and (4) we automatically prove some other cases by induction hypothesis. We packaged this reasoning as a Coq tactic, which in practice can significantly reduce the number of cases to prove, and used this automation

technique to prove local properties of PBFT, such as Castro's A.1.2 local invariants [14]. Because of PBFT's complexity, our Coq tactic typically reduces the number of cases to prove from between 50 to 60 cases down to around 7 cases, sometimes less, as we show in this histogram of goals left to interactively prove after automation:

# of goals left to prove	0	1	2	3	4	5	6	7	8
# of lemmas	8	1	5	4	4	2	9	17	3

4.2 Quorums

As usual, we use quorum theory to trace back correct information between nodes. A (Byzantine) quorum w.r.t. a given set of nodes N, is a subset Q of N, such that $f + 1 \leq (2 * |Q|) - |N|$ (where $|X|$ is the size of X), i.e. every two quorums intersect [59,83] in sufficiently many replicas.[10] Typically, a quorum corresponds to a majority of nodes that agree on some property. In case of state machine replication, quorums are used to ensure that a majority of nodes agree to update the state using the same operation. If we know that two quorums intersect, then we know that both quorums agree, and therefore that the states cannot diverge. In order to reason about quorums, we have proved the following general lemma:[11]

```
Lemma overlapping_quorums :
  forall (l1 l2 : NRlist node), exists Correct,
    (length l1 + length l2) - num_nodes ≤ length Correct
    ∧ subset Correct l1 ∧ subset Correct l2 ∧ no_repeats Correct.
```

This lemma implies that if we have two sets of nodes *l1* and *l2* (NRlist ensures that the sets have no repeats), such that the sum of their length is greater than the total number of nodes (num_nodes), there must exist an overlapping subset of nodes (*Correct*). We use this result below in Sect. 4.4.

The node type parameter is the collection of nodes that can participate in quorums. For example, PBFT replicas can participate in quorums but clients cannot. This type comes with a node2name function to convert nodes into names.

4.3 Certificates

Lemmas that require reasoning about several replicas are much more complex than local properties. They typically require reasoning about some information computed by a collection of replicas (such as quorums) that vouch for the information. In PBFT, a collection of $2f + 1$ messages from different replicas is called

[10] We use here Castro's notation where quorums are *majority* quorums [79] (also called *write quorums*) that require intersections to be non-empty, as opposed to *read quorums* that are only required to intersect with write quorums [36].

[11] We present here a simplified version for readability.

a *strong (or quorum) certificate*, and a collection of $f+1$ messages from different replicas is called a *weak certificate*.

When working with strong certificates, one typically reasons as follows: (1) Because PBFT requires $3f+1$ replicas, two certificates of size $2f+1$ always intersect in $f+1$ replicas. (2) One message among those $f+1$ messages must be from a correct replica because at most f replicas can be faulty. (3) This correct replica can vouch for the information of both quorums—we use that replica to trace back the corresponding information to the point in space/time where/when it was generated. We will get back to this in Sect. 4.4.

When working with weak certificates, one typically reasons as follows: Because, the certificate has size $f+1$ and there are at most f faulty nodes, there must be one correct replica that can vouch for the information of the certificate.

4.4 Knowledge Theory

Model. Let us now present an excerpt of our distributed epistemic knowledge library. Knowledge is a widely studied concept [10,30,31,37–39,70]. It is often captured using possible-worlds models, which rely on Kripke structures: an agent knows a fact if that fact is true in all possible worlds. For distributed systems, agents are nodes and a possible world at a given node is essentially one that has the same local history as the one of the current world, i.e., it captures the current state of the node. As Halpern stresses, e.g. in [37], such a definition of knowledge is *external* in the sense that it cannot necessarily be computed, though some work has been done towards deriving programs from knowledge-based specifications [10]. We follow a different, more pragmatic and computational approach, and say that a node knows some piece of data if it is stored locally, as opposed to the external and logical notion of knowing facts mentioned above. This computational notion of knowledge relies on exchanging messages to propagate it, which is what is required to derive programs from knowledge-based specifications (i.e., to compute that some knowledge is gained [20,37]).

We now extend the model presented in Sect. 3 with two epistemic modal operators *know* and *learn* that express what it means for a process to know and learn some information, and which bear some resemblance with the *fact discovery* and *fact publication* notions discussed in [38]. Formally, we extend our model with the following parameters, which can be instantiated as many times as needed for all the pieces of known/learned data that one wants to reason about—see below for examples:

```
Class LearnAndKnow := MkLearnAndKnow {
    lak_data : Type;          lak_data2info : lak_data → lak_info;
    lak_info : Type;          lak_know : lak_data → lak_memory → Prop;
    lak_memory : Type;        lak_data2owner : lak_data → node;
                              lak_data2auth : lak_data → auth_data; }.
```

The lak_data type is the type of "raw" data that we have knowledge of; while lak_info is some distinct information that might be shared by different pieces

of data. For example, PBFT replicas collect batches of $2f + 1$ (pre-)prepare messages from different replicas, that share the same view, sequence number, and digest. In that case, the (pre-)prepare messages are the raw data that contain the common information consisting of a view, a sequence number, and a digest. The lak_memory type is the type of objects used to store one's knowledge, such as a state machine state. One has to provide a lak_data2info function to extract the information embedded in some piece of data. The lak_know predicate explains what it means to know some piece of data. The lak_data2owner function extracts the "owner" of some piece of data, typically the node that generated the data. In order to authenticate pieces of data, the lak_data2auth function extracts some piece of authenticated data from some piece of raw data. For convenience, we define the following wrapper around lak_data2owner:

```
Definition lak_data2node (d : lak_data) : name := node2name (lak_data2owner d).
```

Let us now turn to the two main components of our theory, namely the know and learn epistemic modal operators. These operators provide an abstraction barrier: they allow us to abstract away from *how* knowledge is stored and computed, in order to focus on the mere *fact* that we have that knowledge.

```
Definition know (sm : node → StateMachine lak_memory) (e : Event) (d : lak_data) :=
  exists mem i, loc e = node2name i
    ∧ state_sm_after_event (sm i) e = Some mem
    ∧ lak_know d mem.
```

where we simply write (StateMachine S) for a state machine with a state of type S, that takes messages as inputs, and outputs lists of directed messages. This states that the state machine (sm i) knows the data d at event e if its state is mem at e and (lak_know d mem) is true. We define learn as follows:

```
Definition learn (e : Event) (d : lak_data) :=
  exists i, loc e = node2name i
    ∧ In (lak_data2auth d) (bind_op_list get_contained_auth_data (trigger e))
    ∧ verify_auth_data (loc e) (lak_data2auth d) (keys e) = true.
```

This states that a node learns d at some event e, if e was triggered by a message that contains the data d. Moreover, because we deal with Byzantine faults, we require that to learn some data one has to be able to verify its authenticity.

Next, we define a few predicates that are useful to track down knowledge. The first one is a local predicate that says that for a state machine to know about a piece of information it has to either have learned it or generated it.

```
Definition learn_or_know (sm : node → StateMachine lak_memory) :=
  forall (d : lak_data) (e : Event),
    know sm e d → (exists e', e' ⊑ e ∧ learn e' d) ∨ lak_data2node d = loc e.
```

The next one is a distributed predicate that states that if one learns some piece of information that is owned by a correct node, then that correct node must have known that piece of information:

```
Definition learn_if_know (sm : node → StateMachine lak_memory) :=
  forall (d : lak_data) (e : Event),
    (learn e d ∧ has_correct_trace_before e (lak_data2node d))
    → exists e', e' ≺ e ∧ loc e' = lak_data2node d ∧ know sm e' d.
```

Using these two predicates, we have proved this general lemma about knowledge propagating through nodes:

```
Lemma know_propagates :
  forall (e : Event) (sm : node → StateMachine lak_memory) (d : lak_data),
    (learn_or_know sm ∧ learn_if_know sm)
    → (know sm e d ∧ has_correct_trace_before e (lak_data2node d))
    → exists e', e' ⪯ e ∧ loc e' = lak_data2node d ∧ know sm e' d.
```

This lemma says that, assuming learn_or_know and learn_if_know, if one knows at some event e some data d that is owned by a correct node, then that correct node must have known that data at a prior event e'. We use this lemma to track down information through correct nodes.

As mentioned in Sect. 4.3, when reasoning about distributed systems, one often needs to reason about certificates, i.e., about collections of messages from different sources. In order to capture this, we introduce the following know_certificate predicate, which says that the state machine sm knows the information i at event e if there exists a list l of pieces of data of length at least k (the certificate size) that come from different sources, and such that sm knows each of these pieces of data, and each piece of data carries the common information nfo:

```
Definition know_certificate (sm : node → StateMachine lak_memory)
           (e : Event) (k : nat) (nfo : lak_info) (P : list lak_data → Prop) :=
  exists (l : list lak_data),
    k ≤ length l ∧ no_repeats (map lak_data2owner l) ∧ P l
    ∧ forall d, In d l → (know sm e d ∧ nfo = lak_data2info d).
```

Using this predicate, we can then combine the quorum and knowledge theories to prove the following lemma, which captures the fact that if there are two quorums for information $nfo1$ (known at $e1$) and $nfo2$ (known at $e2$), and the intersection of the two quorums is guaranteed to contain a correct node, then there must be a correct node (at which $e1'$ and $e2'$ happen) that owns and knows both $nfo1$ and $nfo2$—this lemma follows from know_propagates and overlapping_quorums:

```
Lemma know_in_intersection :
  forall (sm : node → StateMachine lak_memory) (e1 e2 : Event) (nfo1 nfo2 : lak_info)
         (k f : nat) (P : list lak_data → Prop) (E : list Event),
    (learn_or_know sm ∧ learn_if_know sm)
    → (k ≤ num_nodes ∧ num_nodes + f < 2 * k)
    → (exists_at_most_f_faulty E f ∧ In e1 E ∧ In e2 E)
    → (know_certificate sm e1 k nfo1 P ∧ know_certificate sm e2 k nfo2 P)
    → exists e1' e2' d1 d2, loc e1' = loc e2' ∧ e1' ⪯ e1 ∧ e2' ⪯ e2
        ∧ loc e1' = lak_data2node d1 ∧ loc e2' = lak_data2node d2
        ∧ know sm e1' d1 ∧ know sm e2' d2
        ∧ i1 = lak_data2info d1 ∧ i2 = lak_data2info d2.
```

Similarly, we proved the following lemma, which captures the fact that there is always a correct replica that can vouch for the information of a weak certificate:

```
Lemma know_weak_certificate :
 forall (e : Event) (k f : nat) (nfo : lak_info) (P : list lak_data → Prop) (E : list Event),
  (f < k ∧ exists_at_most_f_faulty E f ∧ In e E ∧ know_certificate e k nfo P)
  → exists d, has_correct_trace_before e (node2node d) ∧ know e d ∧ nfo = lak_data2info d.
```

PBFT. One of the key lemmas to prove PBFT's safety says that if two correct replicas have prepared some requests with the same sequence and view numbers, then the requests must be the same [14, Inv.A.1.4]. As mentioned in Sect. 2.1, a replica has prepared a request if it received pre-prepare and prepare messages from a quorum of replicas. To prove this lemma, we instantiated LearnAndKnow as follows: lak_data can either be a pre-prepare or a prepare message; lak_info is the type of triples view/sequence number/digest; lak_memory is the type of states maintained by replicas; lak_data2info extracts the view, sequence number and digest contained in pre-prepare and prepare messages; lak_know states that the pre-prepare or prepare message is stored in the state; lak_data2owner extracts the sender of the message; and lak_data2auth is similar to the PBFTget_contained_auth_data function presented in Sect. 3.6. The two predicates learn_or_know and learn_if_know, which we proved using the tactic discussed in Sect. 4.1, are true about this instance of LearnAndKnow. Inv.A.1.4 is then a straightforward consequence of know_in_intersection applied to the two quorums.

5 Verification of PBFT

Agreement. Velisarios is designed as a general, reusable, and extensible framework that can be instantiated to prove the correctness of any BFT protocol. We demonstrated its usability by proving that our PBFT implementation satisfies the standard agreement property, which is the crux of linearizability (we leave linearizability for future work—see Sect. 2.2 for a high-level definition). Agreement states that, regardless of the view, any two replies sent by correct replicas *i1* and *i2* at events *e1* and *e2* for the same timestamp *ts* to the same client *c* contain the same replies. We proved that this is true in any event ordering that satisfies the assumptions from Sect. 3.6:[12]

```
Lemma agreement :
  forall (eo : EventOrdering) (e1 e2 : Event) (v1 v2 : View) (ts : Timestamp)
         (c : Client) (i1 i2 : Rep) (r1 r2 : Request) (a1 a2 : list Token),
  authenticated_messages_were_sent_or_byz_sys eo PBFTsys ∧ correct_keys eo
  → (exists_at_most_f_faulty [e1,e2] f ∧ loc e1 = PBFTreplica i1 ∧ loc e2 = PBFTreplica i2)
  → In (send_reply v1 ts c i1 r1 a1) (output_system_on_event PBFTsys e1)
  → In (send_reply v2 ts c i2 r2 a2) (output_system_on_event PBFTsys e2)
  → r1 = r2.
```

[12] See agreement in https://github.com/vrahli/Velisarios/blob/master/PBFT/PBFTagreement.v.

where Timestamps are nats; authenticated_messages_were_sent_or_byz_sys is defined on systems using authenticated_messages_were_sent_or_byz; the function output_system_on_event is similar to state_sm_after_event (see Sect. 3.5) but returns the outputs of a given state machine at a given event instead of returning its state; and send_reply builds a reply message. To prove this lemma, we proved most of the invariants stated by Castro in [14, Appendix A]. In addition, we proved that if the last executed sequence number of two correct replicas is the same, then these two replicas have, among other things, the same service state.[13]

As mentioned above, because our model is based on LoE, we only ever prove such properties by induction on causal time. Similarly, Castro proved most of his invariants by induction on the length of the executions. However, he used other induction principles to prove some lemmas, such as Inv.A.1.9, which he proved by induction on views [14, p. 151]. This invariant says that prepared requests have to be consistent with the requests sent in pre-prepare messages by the primary. A straightforward induction on causal time was more natural in our setting.

Castro used a simulation method to prove PBFT's safety: he first proved the safety of a version without garbage collection and then proved that the version with garbage collection implements the one without. This requires defining two versions of the protocol. Instead, we directly prove the safety of the one with garbage collection. This involved proving further invariants about stored, received and sent messages, essentially that they are always within the water marks.

Proof Effort. In terms of proof effort, developing Velisarios and verifying PBFT's agreement property took us around 1 person year. Our generic Velisarios framework consists of around 4000 lines of specifications and around 4000 lines of proofs. Our verified implementation of PBFT consists of around 20000 lines of specifications and around 22000 lines of proofs.

6 Extraction and Evaluation

Extraction. To evaluate our PBFT implementation (i.e., PBFTsys defined in Sect. 3.5—a collection of state machines), we generate OCaml code using Coq's extraction mechanism. Most parameters, such as the number of tolerated faults, are instantiated before extraction. Note that not all parameters need to be instantiated. For example, as mentioned in Sect. 3.1, neither do we instantiate event orderings, nor do we instantiate our assumptions (such as exists_at_most_f_faulty), because they are not used in the code but are only used to prove that properties are true about all possible runs. Also, keys, signatures, and digests are only instantiated by stubs in Coq. We replace those stubs when extracting OCaml code by implementations provided by the nocrypto [66] library, which is the cryptographic library we use to hash, sign, and verify messages (we use RSA).

[13] See same_states_if_same_next_to_execute in https://github.com/vrahli/Velisarios/blob/master/PBFT/PBFTsame_states.v.

Evaluation. To run the extracted code in a real distributed environment, we implemented a small trusted runtime environment in OCaml that uses the Async library [5] to handle sender/receiver threads. We show among other things here that the average latency of our implementation is acceptable compared to the state of the art BFT-SMaRt [8] library. Note that because we do not offer a new protocol, but essentially a re-implementation of PBFT, we expect that on average the scale will be similar in other execution scenarios such as the ones studied by Castro in [14]. We ran our experiments using desktops with 16 GB of memory, and 8 i7-6700 cores running at 3.40 GHz. We report some of our experiments where we used a single client, and a simple state machine where the state is a number, and an operation is either adding or subtracting some value.

We ran a local simulation to measure the performance of our PBFT implementation without network and signatures: when 1 client sends 1 million requests, it takes on average 27.6 µs for the client to receive $f + 1$ ($f = 1$) replies.

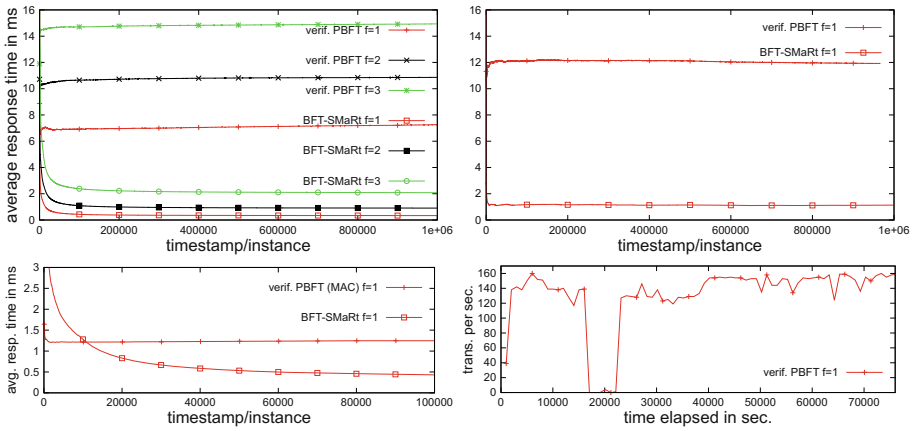


Fig. 3. (1) Single machine (top/left); (2) several machines (top/right); (3) single machine using MACs (bottom/left); (4) view change response time (bottom/right)

Top/left of Fig. 3 shows the experiment where we varied f from 1 to 3, and replicas sent messages, signed using RSA, through sockets, but on a single machine. As mentioned above, we implemented the digital signature-based version of PBFT, while BFT-SMaRt uses a more efficient MAC-based authentication scheme, which in part explains why BFT-SMaRt is around one order of magnitude faster than our implementation. As in [14, Table 8.9], we expect a similar improvement when using the more involved, and as of yet not formally verified, MAC-based version of PBFT (bottom/left of Fig. 3 shows the average response time when replacing digital signatures by MACs, without adapting the rest of the protocol). Top/right of Fig. 3 presents results when running our

version of PBFT and BFT-SMaRt on several machines, for $f = 1$. Finally, bottom/right of Fig. 3 shows the response time of our view-change protocol. In this experiment, we killed the primary after 16 s of execution, and it took around 7 s for the system to recover.

Trusted Computing Base. The TCB of our system includes: (1) the fact that our LoE model faithfully reflects the behavior of distributed systems (see Sect. 3.4); (2) the validity of our assumptions: authenticated_messages_were_sent_or_byz; exists_at_most_f_faulty; correct_keys; and create_hash_collision_resistant (Sect. 3.6); (3) Coq's logic and implementation; (4) OCaml and the nocrypto and Async libraries we use in our runtime environment, and the runtime environment itself (Sect. 6); (5) the hardware and software on which our framework is running.

7 Related Work

Our framework is not the first one for implementing and reasoning about the correctness of distributed systems (see Fig. 4). However, to the best of our knowledge, (1) it is the first theorem prover based tool for verifying the correctness of asynchronous Byzantine fault-tolerant protocols and their implementations; and (2) we provide the first mechanical proof of the safety of a PBFT implementation. Velisarios has evolved from our earlier EventML framework [71], primarily to reason about Byzantine faults and distributed epistemic knowledge.

	Running code	Byz. (synch.)	Byz. (asynch.)
IronFleet/EventML/Verdi/Disel/PSync	✓	✗	✗
HO-model/PVS	✗	✓	✗
Event-B	✓/✗	✓	✗
IOA/TLA$^+$/ByMC	✗	✓	✓
Velisarios	✓	✓	✓

Fig. 4. Comparison with related work

7.1 Logics and Models

IOA [33–35,78] is the model used by Castro [14] to prove PBFT's safety. It is a programming/specification language for describing asynchronous distributed systems as I/O automata [58] (labeled state transition systems) and stating their properties. While IOA is state-based, the logic we use in this paper is event-based. IOA can interact with a large range of tools such as type checkers, simulators, model checkers, theorem provers, and there is support for synthesis of Java code [78]. In contrast, our methodology allows us to both implement and verify protocols within the same tool, namely Coq.

TLA$^+$ [24,51] is a language for specifying and reasoning about systems. It combines: (1) TLA [52], which is a temporal logic for describing systems [51], and (2) set theory, to specify data structures. TLAPS [24] uses a collection of theorem provers, proof assistants, SMT solvers, and decision procedures to mechanically check TLA proofs. Model checker integration helps catch errors before verification attempts. TLA$^+$ has been used in a large number of projects (e.g., [12,18,44,56,63,64]) including proofs of safety and liveness of Multi-Paxos [18], and safety of a variant of an abstract model of PBFT [13]. To the best of our knowledge, TLA$^+$ does not perform program synthesis.

The Heard-Of (HO) Model [23] requires processes to execute in lock-step through rounds into which the distributed algorithms are divided. Asynchronous fault-tolerant systems are treated as synchronous systems with adversarial environments that cause messages to be dropped. The HO-model was implemented in Isabelle/HOL [22] and used, for example, to verify the EIGByz [7] Byzantine agreement algorithm for synchronous systems with reliable links. This formalization uses the notion of *global state of the system* [19], while our approach relies on Lamport's *happened before* relation [53], which does not require reasoning about a distributed system as a single entity (a global state). Model checking and the HO-model were also used in [21,80,81] for verifying the crash fault-tolerant consensus algorithms presented in [23]. To the best of our knowledge, there is no tool that allows generating code from algorithms specified using the HO-model.

Event-B [1] is a set-theory-based language for modeling reactive systems and for *refining* high-level abstract specifications into low-level concrete ones. It supports code generation [32,61], with some limitations (not all features are covered). The Rodin [2] platform for Event-B provides support for refinement, and automated and interactive theorem proving. Both have been used in a number of projects, such as: to prove the safety and liveness of self-$\star$ systems [4]; to prove the agreement and validity properties of the synchronous crash-tolerant Floodset consensus algorithm [57]; and to prove the agreement and validity of synchronous Byzantine agreement algorithms [50]. In [50], the authors assume that messages cannot be forged (using PBFT, at most f nodes can forge messages), and do not verify implementations of these algorithms.

7.2 Tools

Verdi [85,86] is a framework to develop and reason about distributed systems using Coq. As in our framework, Verdi leaves no gaps between verified and running code. Instead, OCaml code is extracted directly from the verified Coq implementation. Verdi provides a compositional way of specifying distributed systems. This is done by applying *verified system transformers*. For example, Raft [67]—an alternative to Paxos—transforms a distributed system into a crash-tolerant one. One difference between our respective methods is that they verify a system by reasoning about the evolution of its global state, while we use Lamport's happened before relation. Moreover, they do not deal with the full spectrum of arbitrary faults (e.g., malicious faults).

Disel [75,84] is a verification framework that implements a separation-style program logic, and that enables compositional verification of distributed systems.

IronFleet [40,41] is a framework for building and reasoning about distributed systems using Dafny [55] and the Z3 theorem prover [62]. Because systems are both implemented in and verified using Dafny, IronFleet also prevents gaps between running and verified code. It uses a combination of TLA-style state-machine refinements [51] to reason about the distributed aspects of protocols, and Floyd-Hoare-style imperative verification techniques to reason about local behavior. The authors have implemented, among other things, the Paxos-based state machine replication library IronRSL, and verified its safety and liveness.

PSync [28] is a domain specific language embedded in Scala, that enables executing and verifying fault-tolerant distributed algorithms in synchronous and partially asynchronous networks. PSync is based on the HO-model, and has been used to implement several crash fault-tolerant algorithms. Similar to the Verdi framework, PSync makes use of a notion of global state and supports reasoning based on the multi-sorted first-order *Consensus verification logic* (CL) [27]. To prove safety, users have to provide invariants, which CL checks for validity. Unlike Verdi, IronFleet and PSync, we focus on Byzantine faults.

ByMC is a model checker for verifying safety and liveness of fault-tolerant distributed algorithms [47–49]. It applies an automated method for model checking parametrized threshold-guarded distributed algorithms (e.g., processes waiting for messages from a majority of distinct senders). ByMC is based on a short counter-example property, which says that if a distributed algorithm violates a temporal specification then there is a counterexample whose length is bounded and independent of the parameters (e.g. the number of tolerated faults).

Ivy [69] allows debugging infinite-state systems using bounded verification, and formally verifying their safety by gradually building universally quantified inductive invariants. To the best of our knowledge, Ivy does not support faults.

Actor Services [77] allows verifying the distributed and functional properties of programs communicating via asynchronous message passing at the level of the source code (they use a simple Java-like language). It supports modular reasoning and proving liveness. To the best of our knowledge, it does not deal with faults.

PVS has been extensively used for verification of synchronous systems that tolerate malicious faults such as in [74], to the extent that its design was influenced by these verification efforts [68].

8 Conclusions and Future Work

We introduced Velisarios, a framework to implement and reason about BFT-SMR protocols using the Coq theorem prover, and described a methodology based on learn/know epistemic modal operators. We used this framework to

prove the safety of a complex system, namely Castro's PBFT protocol. In the future, we plan to also tackle liveness/timeliness. Indeed, proving the safety of a distributed system is far from being enough: a protocol that does not run (which is not live) is useless. Following the same line of reasoning, we want to tackle timeliness because, for real world systems, it is not enough to prove that a system will *eventually reply*. One often desires that the system replies in a timely fashion.

References

1. Abrial, J.-R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.-R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
3. Anand, A., Knepper, R.: ROSCoq: robots powered by constructive reals. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 34–50. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_3
4. Andriamiarina, M.B., Méry, D., Singh, N.K.: Analysis of self-$\star$ and P2P systems using refinement. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 117–123. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_9
5. Async. https://janestreet.github.io/guide-async.html
6. Aublin, P.-L., Mokhtar, S.B., Qéuma, V.: RBFT: redundant Byzantine fault tolerance. In: ICDCS 2013, pp. 297–306. IEEE Computer Society (2013)
7. Bar-Noy, A., Dolev, D., Dwork, C., Raymond Strong, H.: Shifting gears: changing algorithms on the fly to expedite Byzantine agreement. Inf. Comput. **97**(2), 205–233 (1992)
8. Bessani, A.N., Sousa, J., Alchieri, E.A.P.: State machine replication for the masses with BFT-SMART. In: DSN 2014, pp. 355–362. IEEE (2014)
9. Bickford, M.: Component specification using event classes. In: Lewis, G.A., Poernomo, I., Hofmeister, C. (eds.) CBSE 2009. LNCS, vol. 5582, pp. 140–155. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02414-6_9
10. Bickford, M., Constable, R.C., Halpern, J.Y., Petride, S.: Knowledge-based synthesis of distributed systems using event structures. In: Baader, F., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3452, pp. 449–465. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32275-7_30
11. Bickford, M., Constable, R.L., Rahli, V.: Logic of events, a framework to reason about distributed systems. In: Languages for Distributed Algorithms Workshop (2012)
12. Bolosky, W.J., Douceur, J.R., Howell, J.: The Farsite project: a retrospective. Oper. Syst. Rev. **41**(2), 17–26 (2007)
13. Mechanically Checked Safety Proof of a Byzantine Paxos Algorithm. http://lamport.azurewebsites.net/tla/byzpaxos.html
14. Castro, M.: Practical Byzantine Fault Tolerance. Also as Technical report MIT-LCS-TR-817. Ph.D. MIT, January 2001
15. Castro, M., Liskov, B.: A correctness proof for a practical Byzantine-fault-tolerant replication algorithm. Technical Memo MIT-LCS-TM-590. MIT, June 1999

16. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: OSDI 1999, pp. 173–186. USENIX Association (1999)
17. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (2002)
18. Chand, S., Liu, Y.A., Stoller, S.D.: Formal verification of multi-paxos for distributed consensus. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 119–136. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_8
19. Mani Chandy, K., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985)
20. Mani Chandy, K., Misra, J.: How processes learn. Distrib. Comput. **1**(1), 40–52 (1986)
21. Chaouch-Saad, M., Charron-Bost, B., Merz, S.: A reduction theorem for the verification of round-based distributed algorithms. In: Bournez, O., Potapov, I. (eds.) RP 2009. LNCS, vol. 5797, pp. 93–106. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04420-5_10
22. Charron-Bost, B., Debrat, H., Merz, S.: Formal verification of consensus algorithms tolerating malicious faults. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 120–134. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24550-3_11
23. Charron-Bost, B., Schiper, A.: The Heard-Of model: computing in distributed systems with benign faults. Distrib. Comput. **22**(1), 49–71 (2009)
24. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the TLA$^+$ proof system. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 142–148. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_12
25. The Coq Proof Assistant. http://coq.inria.fr/
26. Distler, T., Cachin, C., Kapitza, R.: Resource-efficient Byzantine fault tolerance. IEEE Trans. Comput. **65**(9), 2807–2819 (2016)
27. Drăgoi, C., Henzinger, T.A., Veith, H., Widder, J., Zufferey, D.: A logic-based framework for verifying consensus algorithms. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 161–181. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_10
28. Dragoi, C., Henzinger, T.A., Zufferey, D.: PSync: a partially synchronous language for fault-tolerant distributed algorithms. In: POPL 2016, pp. 400–415. ACM (2016)
29. Dragoi, C., Henzinger, T.A., Zufferey, D.: The need for language support for fault-tolerant distributed systems. In: SNAPL 2015. LIPIcs, vol. 32, pp. 90–102. Schloss Dagstuhl - Leibniz-Zentrum fuerInformatik (2015)
30. Dwork, C., Moses, Y.: Knowledge and common knowledge in a Byzantine environment: crash failures. Inf. Comput. **88**(2), 156–186 (1990)
31. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Knowledge-based programs. Distrib. Comput. **10**(4), 199–225 (1997)
32. Fürst, A., Hoang, T.S., Basin, D., Desai, K., Sato, N., Miyazaki, K.: Code generation for Event-B. In: Albert, E., Sekerinski, E. (eds.) IFM 2014. LNCS, vol. 8739, pp. 323–338. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10181-1_20
33. Garland, S., Lynch, N., Tauber, J., Vaziri, M.: IOA user guide and reference manual. Technical report MIT/LCS/TR-961. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA (2004)

34. Garland, S.J., Lynch, N.: Using I/O automata for developing distributed systems. In: Foundations of Component Based Systems, pp. 285–312. Cambridge University Press, New York (2000)
35. Georgiou, C., Lynch, N., Mavrommatis, P., Tauber, J.A.: Automated implementation of complex distributed algorithms specified in the IOA language. Int. J. Softw. Tools Technol. Transf. **11**, 153–171 (2009)
36. Gifford, D.K.: Weighted voting for replicated data. In: SOSP 1979, pp. 150–162. ACM (1979)
37. Halpern, J.Y.: Using reasoning about knowledge to analyze distributed systems. Ann. Rev. Comput. Sci. **2**(1), 37–68 (1987). https://doi.org/10.1146/annurev.cs.02.060187.000345
38. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. J. ACM **37**(3), 549–587 (1990)
39. Halpern, J.Y., Zuck, L.D.: A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols. J. ACM **39**(3), 449–478 (1992)
40. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: SOSP 2015, pp. 1–17. ACM (2015)
41. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving safety and liveness of practical distributed systems. Commun. ACM **60**(7), 83–92 (2017)
42. Herlihy, M., Wing, J.M.: Axioms for concurrent objects. In: POPL 1987, pp. 13–26. ACM Press (1987)
43. Jajodia, S., Ghosh, A.K., Swarup, V., Wang, C., Wang, X.S.: Moving Target Defense - Creating Asymmetric Uncertainty for Cyber Threats. Advances in Information Security, vol. 54. Springer, New York (2011). https://doi.org/10.1007/978-1-4614-0977-9
44. Joshi, R., Lamport, L., Matthews, J., Tasiran, S., Tuttle, M.R., Yuan, Y.: Checking cache-coherence protocols with TLA$^+$. Formal Methods Syst. Des. **22**(2), 125–131 (2003)
45. Kapitza, R., Behl, J., Cachin, C., Distler, T., Kuhnle, S., Mohammadi, S.V., Schröder-Preikschat, W., Stengel, K.: CheapBFT: resource-efficient Byzantine fault tolerance. In: EuroSys 2012, pp. 295–308. ACM (2012)
46. Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B.: Enhancing Bitcoin security and performance with strong consistency via collective signing. In: USENIX Security Symposium, pp. 279–296. USENIX Association (2016)
47. Konnov, I.V., Lazic, M., Veith, H., Widder, J.: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: POPL 2017, pp. 719–734. ACM (2017)
48. Konnov, I.V., Veith, H., Widder, J.: On the completeness of bounded model checking for threshold-based distributed algorithms: reachability. Inf. Comput. **252**, 95–109 (2017)
49. Konnov, I., Veith, H., Widder, J.: SMT and POR beat counter abstraction: parameterized model checking of threshold-based distributed algorithms. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 85–102. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_6
50. Krenický, R., Ulbrich, M.: Deductive verification of a Byzantine agreement protocol. Technical report 2010-7. Karlsruhe Institute of Technology, Department of Computer Science (2010)

51. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2004)
52. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
53. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
54. Lamport, L., Shostak, R.E., Pease, M.C.: The Byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)
55. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
56. Lu, T., Merz, S., Weidenbach, C.: Towards verification of the pastry protocol using TLA$^+$. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 244–258. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21461-5_16
57. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
58. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: PODC 1987, pp. 137–151. ACM (1987)
59. Malkhi, D., Reiter, M.K.: Byzantine quorum systems. In: STOC 1997, pp. 569–578. ACM (1997)
60. Mattern, F.: Virtual time and global states of distributed systems. In: Proceedings of the Workshop on Parallel and Distributed Algorithms, pp. 215–226. North-Holland/Elsevier (1989). Reprinted. In: Yang, Z., Marsland, T.A. (eds.) Global States and Time in Distributed Systems, pp. 123–133. IEEE (1994)
61. Méry, D., Singh, N.K.: Automatic code generation from event-B models. In: Symposium on Information and Communication Technology, SoICT 2011, pp. 179–188. ACM (2011)
62. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
63. Newcombe, C.: Why Amazon chose TLA$^+$. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 25–39. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_3
64. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
65. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri Nets, event structures and domains, Part I. Theor. Comput. Sci. **13**, 85–108 (1981)
66. nocrypto. https://github.com/mirleft/ocaml-nocrypto
67. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, 19–20 June 2014, pp. 305–319. USENIX Association (2014)
68. Owre, S., Rushby, J.M., Shankar, N., von Henke, F.W.: Formal verification for fault-tolerant architectures: prolegomena to the design of PVS. IEEE Trans. Softw. Eng. **21**(2), 107–125 (1995)
69. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: PLDI 2016, pp. 614–630. ACM (2016)
70. Panangaden, P., Taylor, K.: Concurrent common knowledge: defining agreement for asynchronous systems. Distrib. Comput. **6**(2), 73–93 (1992)

71. Rahli, V., Guaspari, D., Bickford, M., Constable, R.L.: EventML: Specification, verification, and implementation of crash-tolerant state machine replication systems. In: SCP (2017)
72. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River (1997)
73. Schiper, N., Rahli, V., van Renesse, R., Bickford, M., Constable, R.L.: Developing correctly replicated databases using formal tools. In: DSN 2014, pp. 395–406. IEEE (2014)
74. Schmid, U., Weiss, B., Rushby, J.M.: Formally verified Byzantine agreement in presence of link faults. In: ICDCS, pp. 608–616 (2002)
75. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. In: POPL 2018 (2018)
76. Sousa, P.: Proactive resilience. Ph.D. thesis. Faculty of Sciences, University of Lisbon, Lisbon, May 2007
77. Summers, A.J., Müller, P.: Actor services. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 699–726. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_27
78. Tauber, J.A.: Verifiable compilation of I/O automata without global synchronization. Ph.D. thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA (2004)
79. Thomas, R.H.: A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. Database Syst. **4**(2), 180–209 (1979)
80. Tsuchiya, T., Schiper, A.: Model checking of consensus algorithm. In: SRDS 2007, pp. 137–148. IEEE Computer Society (2007)
81. Tsuchiya, T., Schiper, A.: Using bounded model checking to verify consensus algorithms. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 466–480. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87779-0_32
82. Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C., Veríssimo, P.: Efficient Byzantine fault-tolerance. IEEE Trans. Comput. **62**(1), 16–30 (2013)
83. Vukolic, M.: The origin of quorum systems. Bull. EATCS **101**, 125–147 (2010)
84. Wilcox, J.R., Sergey, I., Tatlock, Z.: Programming language abstractions for modularly verified distributed systems. In: SNAPL 2017. LIPIcs, vol. 71, pp. 19:1–19:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
85. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI 2015, pp. 357–368. ACM (2015)
86. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.E.: Planning for change in a formal verification of the raft consensus protocol. In: CPP 2016, pp. 154–165. ACM (2016)

Program Analysis and Automated Verification

Evaluating Design Tradeoffs in Numeric Static Analysis for Java

Shiyi Wei[1(✉)], Piotr Mardziel[2], Andrew Ruef[3], Jeffrey S. Foster[3], and Michael Hicks[3]

[1] The University of Texas at Dallas, Richardson, USA
swei@utdallas.edu
[2] Carnegie Mellon University, Moffett Field, USA
piotrm@gmail.com
[3] University of Maryland, College Park, USA
{awruef,jfoster,mwh}@cs.umd.edu

Abstract. Numeric static analysis for Java has a broad range of potentially useful applications, including array bounds checking and resource usage estimation. However, designing a scalable numeric static analysis for real-world Java programs presents a multitude of design choices, each of which may interact with others. For example, an analysis could handle method calls via either a top-down or bottom-up interprocedural analysis. Moreover, this choice could interact with how we choose to represent aliasing in the heap and/or whether we use a relational numeric domain, e.g., convex polyhedra. In this paper, we present a family of abstract interpretation-based numeric static analyses for Java and systematically evaluate the impact of 162 analysis configurations on the DaCapo benchmark suite. Our experiment considered the precision and performance of the analyses for discharging array bounds checks. We found that top-down analysis is generally a better choice than bottom-up analysis, and that using access paths to describe heap objects is better than using summary objects corresponding to points-to analysis locations. Moreover, these two choices are the most significant, while choices about the numeric domain, representation of abstract objects, and context-sensitivity make much less difference to the precision/performance tradeoff.

1 Introduction

Static analysis of numeric program properties has a broad range of useful applications. Such analyses can potentially detect array bounds errors [50], analyze a program's resource usage [28,30], detect side channels [8,11], and discover vectors for denial of service attacks [10,26].

One of the major approaches to numeric static analysis is abstract interpretation [18], in which program statements are evaluated over an abstract domain until a fixed point is reached. Indeed, the first paper on abstract interpretation [18] used numeric intervals as one example abstract domain,

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 653–682, 2018.
https://doi.org/10.1007/978-3-319-89884-1_23

and many subsequent researchers have explored abstract interpretation-based numeric static analysis [13,22–25,31].

Despite this long history, applying abstract interpretation to real-world Java programs remains a challenge. Such programs are large, have many interacting methods, and make heavy use of heap-allocated objects. In considering how to build an analysis that aims to be sound but also precise, prior work has explored some of these challenges, but not all of them together. For example, several works have considered the impact of the choice of numeric domain (e.g., intervals vs. convex polyhedra) in trading off precision for performance but not considered other tradeoffs [24,38]. Other works have considered how to integrate a numeric domain with analysis of the heap, but unsoundly model method calls [25] and/or focus on very precise properties that do not scale beyond small programs [23,24]. Some scalability can be recovered by using programmer-specified pre- and post-conditions [22]. In all of these cases, there is a lack of consideration of the broader design space in which many implementation choices interact. (Sect. 7 considers prior work in detail.)

In this paper, we describe and then systematically explore a large design space of fully automated, abstract interpretation-based numeric static analyses for Java. Each analysis is identified by a choice of five configurable options—the numeric domain, the heap abstraction, the object representation, the interprocedural analysis order, and the level of context sensitivity. In total, we study 162 analysis configurations to asses both how individual configuration options perform overall and to study interactions between different options. To our knowledge, our basic analysis is one of the few fully automated numeric static analyses for Java, and we do not know of any prior work that has studied such a large static analysis design space.

We selected analysis configuration options that are well-known in the static analysis literature and that are key choices in designing a Java static analysis. For the numeric domain, we considered both intervals [17] and convex polyhedra [19], as these are popular and bookend the precision/performance spectrum. (See Sect. 2.)

Modeling the flow of data through the heap requires handling pointers and aliasing. We consider three different choices of *heap abstraction*: using *summary objects* [25,27], which are *weakly updated*, to summarize multiple heap locations; *access paths* [21,52], which are *strongly updated*; and a combination of the two.

To implement these abstractions, we use an ahead-of-time, global *points-to analysis* [44], which maps static/local variables and heap-allocated fields to abstract objects. We explore three variants of *abstract object representation*: the standard *allocation-site abstraction* (the most precise) in which each syntactic `new` in the program represents an abstract object; *class-based abstraction* (the least precise) in which each class represents all instances of that class; and a *smushed string abstraction* (intermediate precision) which is the same as allocation-site abstraction except strings are modeled using a class-based abstraction [9]. (See Sect. 3.)

We compare three choices in the *interprocedural analysis order* we use to model method calls: *top-down analysis*, which starts with `main` and analyzes callees as they are encountered; and *bottom-up analysis*, which starts at the leaves of the call tree and instantiates method summaries at call sites; and a hybrid analysis that is bottom-up for library methods and top-down for application code. In general, top-down analysis explores fewer methods, but it may analyze callees multiple times. Bottom-up analysis explores each method once but needs to create summaries, which can be expensive.

Finally, we compare three kinds of *context-sensitivity* in the points-to analysis: *context-insensitive* analysis, *1-CFA analysis* [46] in which one level of calling context is used to discriminate pointers, and *type-sensitive analysis* [49] in which the type of the receiver is the context. (See Sect. 4.)

We implemented our analysis using WALA [2] for its intermediate representation and points-to analyses and either APRON [33,41] or ELINA [47,48] for the interval or polyhedral, respectively, numeric domain. We then applied all 162 analysis configurations to the DaCapo benchmark suite [6], using the numeric analysis to try to prove array accesses are within bounds. We measured the analyses' performance and the number of array bounds checks they discharged. We analyzed our results by using a multiple linear regression over analysis features and outcomes, and by performing data visualizations.

We studied three research questions. First, we examined how analysis configuration affects performance. We found that using summary objects causes significant slowdowns, e.g., the vast majority of the analysis runs that timed out used summary objects. We also found that polyhedral analysis incurs a significant slowdown, but only half as much as summary objects. Surprisingly, bottom-up analysis provided little performance advantage generally, though it did provide some benefit for particular object representations. Finally, context-insensitive analysis is faster than context-sensitive analysis, as might be expected, but the difference is not great when combined with more approximate (class-based and smushed string) abstract object representations.

Second, we examined how analysis configuration affects precision. We found that using access paths is critical to precision. We also found that the bottom-up analysis has worse precision than top-down analysis, especially when using summary objects, and that using a more precise abstract object representation improves precision. But other traditional ways of improving precision do so only slightly (the polyhedral domain) or not significantly (context-sensitivity).

Finally, we looked at the precision/performance tradeoff for all programs. We found that using access paths is always a good idea, both for precision and performance, and top-down analysis works better than bottom-up. While summary objects, originally proposed by Fu [25], do help precision for some programs, the benefits are often marginal when considered as a percentage of all checks, so they tend not to outweigh their large performance disadvantage. Lastly, we found that the precision gains for more precise object representations and polyhedra are modest, and performance costs can be magnified by other analysis features.

Table 1. Analysis configuration options, and their possible settings.

Config. Option	Setting	Description
Numeric domain (ND)	INT	Intervals
	POL	Polyhedra
Heap abstraction (HA)	SO	Only summary objects
	AP	Only access paths
	AP+SO	Both access paths and summary objects
Abstract object representation (OR)	ALLO	Alloc-site abstraction
	CLAS	Class-based abstraction
	SMUS	Alloc-site except Strings
Inter-procedural analysis order (AO)	TD	Top-down
	BU	Bottom-up
	TD+BU	Hybrid top-down and bottom-up
Context sensitivity (CS)	CI	Context-insensitive
	1CFA	1-CFA
	1TYP	Type-sensitive

In summary, our empirical study provides a large, comprehensive evaluation of the effects of important numeric static analysis design choices on performance, precision, and their tradeoff; it is the first of its kind. Our code and data is available at https://github.com/plum-umd/JANA.

2 Numeric Static Analysis

A *numeric static analysis* is one that tracks numeric properties of memory locations, e.g., that $x \leqslant 5$ or $y > z$. A natural starting point for a numeric static analysis for Java programs is numeric abstract interpretation over program variables within a single procedure/method [18].

A standard abstract interpretation expresses numeric properties using a *numeric abstract domain*, of which the most common are *intervals* (also known as boxes) and *convex polyhedra*. Intervals [17] define abstract states using inequalities of the form p *relop* n where p is a variable, n is a constant integer, and *relop* is a relational operator such as $\leqslant$. A variable such as p is sometimes called a *dimension*, as it describes one axis of a numeric space. Convex polyhedra [19] define abstract states using linear relationships between variables and constants, e.g., of the form $3p_1 - p_2 \leqslant 5$. Intervals are less precise but more efficient than polyhedra. Operation on intervals have time complexity linear in the number of dimensions whereas the time complexity for polyhedra operations is exponential in the number of dimensions.[1]

[1] Further, the time complexity of join is $O(d \cdot c^{2^{d+1}})$ where c is the number of constraints, and d is the number of dimensions [47].

Numeric abstract interpretation, including our own analyses, are usually flow-sensitive, i.e., each program point has an associated abstract state characterizing properties that hold at that point. Variable assignments are *strong updates*, meaning information about the variable is replaced by information from the right-hand side of the assignment. At merge points (e.g., after the completion of a conditional), the abstract states of the possible prior states are *joined* to yield properties that hold regardless of the branch taken. Loop bodies are reanalyzed until their constituent statements' abstract states reach a fixed point. Reaching a fixed point is accelerated by applying the numeric domain's standard *widening* operator [4] in place of join after a fixed number of iterations.

Scaling a basic numeric abstract interpreter to full Java requires making many design choices. Table 1 summarizes the key choices we study in this paper. Each configuration option has a range of settings that potentially offer different precision/performance tradeoffs. Different options may interact with each other to affect the tradeoff. In total, we study five options with two or three settings each. We have already discussed the first option, the numeric domain (ND), for which we consider intervals (INT) and polyhedra (POL). The next two options consider the heap, and are discussed in the next section, and the last two options consider method calls, and are discussed in Sect. 4.

For space reasons, our paper presentation focuses on the high-level design and tradeoffs. Detailed algorithms are given formally in the technical report [51] for the heap and interprocedural analysis.

3 The Heap

The numeric analysis described so far is sufficient only for analyzing code with local, numeric variables. To analyze numeric properties of heap-manipulating programs, we must also consider heap locations $x.f$, where x is a reference to a heap-allocated object, and f is a numeric field.[2] To do so requires developing a *heap abstraction* (HA) that accounts for aliasing. In particular, when variables x and y may point to the same heap object, an assignment to $x.f$ could affect $y.f$. Moreover, the referent of a pointer may be uncertain, e.g., the true branch of a conditional could assign location o_1 to x, while the false branch could assign o_2 to x. This uncertainty must be reflected in subsequent reads of $x.f$.

We use a *points-to analysis* to reason about aliasing. A points-to analysis computes a mapping Pt from variables x and access paths $x.f$ to (one or more) *abstract objects* [44]. If Pt maps two variables/paths p_1 and p_2 to a common abstract object o then p_1 and p_2 *may alias*. We also use points-to analysis to determine the call graph, i.e., to determine what method may be called by an expression $x.m(\ldots)$ (discussed in Sect. 4).

[2] In our implementation, statements such as $z = x.f.g$ are decomposed so that paths are at most length one, e.g., $w = x.f; z = w.g$.

3.1 Summary Objects (SO)

The first heap abstraction we study is based on Fu [25]: use a *summary object* (SO) to abstract information about multiple heap locations as a single abstract state "variable" [27]. As an example, suppose that $Pt(x) = \{o\}$ and we encounter the assignment $x.f$:= 5. Then in this approach, we add a variable o_f to the abstract state, modeling the field f of object o, and we add constraint $o\_f = n$. Subsequent assignments to such summary objects must be *weak updates*, to respect the *may alias* semantics of the points-to analysis. For example, suppose $y.f$ may alias $x.f$, i.e., $o \in Pt(x) \cap Pt(y)$. Then after a later assignment $y.f$:= 7 the analysis would weakly update o_f with 7, producing constraints $5 \leqslant o\_f \leqslant 7$ in the abstract state. These constraints conservatively model that either $o\_f = 5$ or $o\_f = 7$, since the assignment to $y.f$ may or may not affect $x.f$.

In general, weak updates are more expensive than strong updates, and reading a summary object is more expensive than reading a variable. A strong update to x is implemented by *forgetting* x in the abstract state,[3] and then re-adding it to be equal to the assigned-to value. Note that x cannot appear in the assigned-to value because programs are converted into static single assignment form (Sect. 5). A weak update—which is not directly supported in the numeric domain libraries we use—is implemented by copying the abstract state, strongly updating x in the copy, and then joining the two abstract states. Reading from a summary object requires "expanding" the abstract state with a copy o'_f of the summary object and its constraints, creating a constraint on o'_f, and then forgetting o'_f. Doing this ensures that operations on a variable into which a summary object is read do not affect prior reads. A normal read just references the read variable.

Fu [25] argues that this basic approach is better than ignoring heap locations entirely by measuring how often field reads are not unconstrained, as would be the case for a heap-unaware analysis. However, it is unclear whether the approach is sufficiently precise for applications such as array-bounds check elimination. Using the polyhedra numeric domain should help. For example, a `Buffer` class might store an array in one field and a conservative bound on an array's length in another. The polyhedral domain will permit relating the latter to the former while the interval domain will not. But the slowdown due to the many added summary objects may be prohibitive.

3.2 Access Paths (AP)

An alternative heap abstraction we study is to treat *access paths* (AP) as if they are normal variables, while still accounting for possible aliasing [21,52]. In particular, a path $x.f$ is modeled as a variable x_f, and an assignment $x.f$:= n strongly updates x_f to be n. At the same time, if there exists another path $y.f$ and x and y may alias, then we must weakly update y_f as possibly containing n. In general, determining which paths must be weakly updated depends on the abstract object representation and context-sensitivity of the points-to analysis.

[3] Doing so has the effect of "connecting" constraints that are transitive via x. For example, given $y \leqslant x \leqslant 5$, forgetting x would yield constraint $y \leqslant 5$.

Two key benefits of AP over SO are that (1) AP supports strong updates to paths $x.f$, which are more precise and less expensive than weak updates, and (2) AP may require fewer variables to be tracked, since, in our design, access paths are mostly local to a method whereas points-to sets are computed across the entire program. On the other hand, SO can do better at summarizing invariants about heap locations pointed to by other heap locations, i.e., not necessarily via an access path. Especially when performing an interprocedural analysis, such information can add useful precision.

Combined (AP+SO). A natural third choice is to combine AP and SO. Doing so sums both the costs and benefits of the two approaches. An assignment $x.f$:= n strongly updates x_f and weakly updates o_f for each o in $Pt(x)$ and each y_f where $Pt(x) \cap Pt(y) \neq \varnothing$. Reading from $x.f$ when it has not been previously assigned to is just a normal read, after first strongly updating x_f to be the join of the summary read of o_f for each $o \in Pt(x)$.

3.3 Abstract Object Representation (OR)

Another key precision/performance tradeoff is the *abstract object representation* (OR) used by the points-to analysis. In particular, when $Pt(x) = \{o_1, ..., o_n\}$, where do the names $o_1, ..., o_n$ come from? The answer impacts the naming of summary objects, the granularity of alias checks for assignments to access paths, and the precision of the call-graph, which requires aliasing information to determine which methods are targeted by a dynamic dispatch $x.m(...)$.

As shown in the third row of Table 1, we explore three representations for abstract objects. The first choice names abstract objects according to their *allocation site* (ALLO)—all objects allocated at the same program point have the same name. This is precise but potentially expensive, since there are many possible allocation sites, and each path $x.f$ could be mapped to many abstract objects. We also consider representing abstract objects using *class names* (CLAS), where all objects of the same class share the same abstract name, and a hybrid *smushed string* (SMUS) approach, where every `String` object has the same abstract name but objects of other types have allocation-site names [9]. The class name approach is the least precise but potentially more efficient since there are fewer names to consider. The smushed string analysis is somewhere in between. The question is whether the reduction in names helps performance enough, without overly compromising precision.

4 Method Calls

So far we have considered the first three options of Table 1, which handle integer variables and the heap. This section considers the last two options—interprocedural analysis order (AO) and context sensitivity (CS).

4.1 Interprocedural Analysis Order (AO)

We implement three styles of interprocedural analysis: top-down (TD), bottom-up (BU), and their combination (TD+BU). The TD analysis starts at the program entry point and, as it encounters method calls, analyzes the body of the callee (memoizing duplicate calls). The BU analysis starts at the leaves of the call graph and analyzes each method in isolation, producing a summary of its behavior [29,53]. (We discuss call graph construction in the next subsection.) This summary is then instantiated at each method call. The hybrid analysis works top-down for application code but bottom-up for any code from the Java standard library.

Top-Down (TD). Assuming the analyzer knows the method being called, a simple approach to top-down analysis would be to transfer the caller's state to the beginning of callee, analyze the callee in that state, and then transfer the state at the end of the callee back to the caller. Unfortunately, this approach is prohibitively expensive because the abstract state would accumulate all local variables and access paths across all methods along the call-chain.

We avoid this blowup by analyzing a call to method m while considering only relevant local variables and heap abstractions. Ignoring the heap for the moment, the basic approach is as follows. First, we make a copy C_m of the caller's abstract state C. In C_m, we set variables for m's formal numeric arguments to the actual arguments and then forget (as defined in Sect. 3.1) the caller's local variables. Thus C_m will only contain the portion of C relevant to m. We analyze m's body, starting in C_m, to yield the final state C'_m. Lastly, we merge C and C'_m, strongly update the variable that receives the returned result, and forget the callee's local variables—thus avoiding adding the callee's locals to the caller's state.

Now consider the heap. If we are using summary objects, when we copy C to C_m we do not forget those objects that might be used by m (according to the points-to analysis). As m is analyzed, the summary objects will be weakly updated, ultimately yielding state C'_m at m's return. To merge C'_m with C, we first forget the summary objects in C not forgotten in C_m and then concatenate C'_m with C. The result is that updated summary objects from C'_m replace those that were in the original C.

If we are using access paths, then at the call we forget access paths in C because assignments in m's code might invalidate them. But if we have an access path $x.f$ in the caller and we pass x to m, then we retain $x.f$ in the callee but rename it to use m's parameter's name. For example, $x.f$ becomes $y.f$ if m's parameter is y. If y is never assigned to in m, we can map $y.f$ back to $x.f$ (in the caller) once m returns.[4] All other access paths in C_m are forgotten prior to concatenating with the caller's state.

Note that the above reasoning is only for numeric values. We take no particular steps for pointer values as the points-to analysis already tracks those across all methods.

[4] Assignments to $y.f$ in the callee are fine; only assignments to y are problematic.

Bottom Up (BU). In the BU analysis, we analyze a method m's body to produce a *method summary* and then instantiate the summary at calls to m. Ignoring the heap, producing a method summary for m is straightforward: start analyzing m in a state C_m in which its (numeric) parameters are unconstrained variables. When m returns, forget all variables in the final state except the parameters and return value, yielding a state C'_m that is the method summary. Then, when m is called, we concatenate C'_m with the current abstract state; add constraints between the parameters and their actual arguments; strongly update the variable receiving the result with the summary's returned value; and then forget those variables.

When using the polyhedral numeric domain, C'_m can express relationships between input and output parameters, e.g., `ret` $\leqslant$ z or `ret = x+y`. For the interval domain, which is non-relational, summaries are more limited, e.g., they can express `ret` $\leqslant 100$ but not `ret` $\leqslant$ `x`. As such, we expect bottom-up analysis to be far more useful with the polyhedral domain than the interval domain.

Summary Objects. Now consider the heap. Recall that when using summary objects in the TD analysis, reading a path $x.f$ into z "expands" each summary object o_f when $o \in Pt(x)$ and strongly updates z with the join of these expanded objects, before forgetting them. This expansion makes a copy of each summary object's constraints so that later use of z does not incorrectly impact the summary. However, when analyzing a method bottom-up, we may not yet know all of a summary object's constraints. For example, if x is passed into the current method, we will not (yet) know if o_f is assigned to a particular numeric range in the caller.

We solve this problem by allocating a fresh, unconstrained *placeholder object* at each read of $x.f$ and include it in the initialization of the assigned-to variable z. The placeholder is also retained in m's method summary. Then at a call to m, we instantiate each placeholder with the constraints in the caller involving the placeholder's summary location. We also create a fresh placeholder in the caller and weakly update it to the placeholder in the callee; doing so allows for further constraints to be added from calls further up the call chain.

Access Paths. If we are using access paths, we treat them just as in TD—each $x.f$ is allocated a special variable that is strongly updated when possible, according to the points-to analysis. These are not kept in method summaries. When also using summary objects, at the first read to $x.f$ we initialize it from the summary objects derived from x's points-to set, following the above expansion procedure. Otherwise $x.f$ will be unconstrained.

Hybrid (TD+BU). In addition to TD or BU analysis (only), we implemented a hybrid strategy that performs TD analysis for the application, but BU analysis for code from the Java standard library. Library methods are analyzed first, bottom-up. Application method calls are analyzed top-down. When an application method calls a library method, it applies the BU method call approach.

TD+BU could potentially be better than TD because library methods, which are likely called many times, only need to be analyzed once. TD+BU could similarly be better than BU because application methods, which are likely not called as many times as library methods, can use the lower-overhead TD analysis.

Now, consider the interaction between the heap abstraction and the analysis order. The use of access paths (only) does not greatly affect the normal TD/BU tradeoff: TD may yield greater precision by adding constraints from the caller when analyzing the callee, while BU's lower precision comes with the benefit of analyzing method bodies less often. Use of summary objects complicates this tradeoff. In the TD analysis, the use of summary objects adds a relatively stable overhead to all methods, since they are included in every method's abstract state. For the BU analysis, methods further down in the call chain will see fewer summary objects used, and method bodies may end up being analyzed less often than in the TD case. On the other hand, placeholder objects add more dimensions overall (one per read) and more work at call sites (to instantiate them). But, instantiating a summary may be cheaper than reanalyzing the method.

4.2 Context Sensitivity (CS)

The last design choice we considered was context-sensitivity. A *context-insensitive* (CI) analysis conflates information from different call sites of the same method. For example, two calls to method m in which the first passes x_1, y_1 and the second passes x_2, y_2 will be conflated such that within m we will only know that either x_1 or x_2 is the first parameter, and either y_1 or y_2 is the second; we will miss the correlation between parameters. A context sensitive analysis provides some distinction among different call sites. A *1-CFA analysis* [46] (1CFA) distinguishes based on one level of calling context, i.e., two calls originating from different program points will be distinguished, but two calls from the same point, but in a method called from two different points will not. A *type-sensitive analysis* [49] (1TYP) uses the type of the receiver as the context.

Context sensitivity in the points-to analysis affects alias checks, e.g., when determining whether an assignment to $x.f$ might affect $y.f$. It also affects the abstract object representation and call graph construction. Due to the latter, context sensitivity also affects our interprocedural numeric analysis. In a context-sensitive analysis, a single method is essentially treated as a family of methods indexed by a calling context. In particular, our analysis keeps track of the current context as a *frame*, and when considering a call to method `x.m()`, the target methods to which `m` may refer differ depending on the frame. This provides more precision than a context-insensitive (i.e., frame-less) approach, but the analysis may consider the same method code many times, which adds greater precision but also greater expense. This is true both for TD and BU, but is perhaps more detrimental to the latter since it reduces potential method summary reuse. On the other hand, more precise analysis may reduce unnecessary work by pruning infeasible call graph edges. For example, when a call might dynamically dispatch to several different methods, the analysis must consider them all, joining their abstract states. A more precise analysis may consider fewer target methods.

5 Implementation

We have implemented an analysis for Java with all of the options described in the previous two sections. Our implementation is based on the intermediate representation in the T. J. Watson Libraries for Analysis (WALA) version 1.3.10 [2], which converts a Java bytecode program into static single assignment (SSA) form [20], which is then analyzed. We use APRON [33,41] trunk revision 1096 (published on 2016/05/31) implementation of intervals, and ELINA [47,48], snapshot as of October 4, 2017, for convex polyhedra. Our current implementation supports all non-floating point numeric Java values and comprises 14 K lines of Scala code.

Next we discuss a few additional implementation details.

Preallocating Dimensions. In both APRON and ELINA, it is very expensive to perform join operations that combine abstract states with different variables. Thus, rather than add dimensions as they arise during abstract interpretation, we instead *preallocate* all necessary dimensions—including for local variables, access paths, and summary objects, when enabled—at the start of a method body. This ensures the abstract states have the same dimensions at each join point. We found that, even though this approach makes some states larger than they need to be, the overall performance savings is still substantial.

Arrays. Our analysis encodes an array as an object with two fields, `contents`, which represents the contents of the array, and `len`, representing the array's length. Each read/write from `a[i]` is modeled as a weak read/write of `contents` (because all array elements are represented with the same field), with an added check that `i` is between 0 and `len`. We treat `String`s as a special kind of array.

Widening. As is standard in abstract interpretation, our implementation performs widening to ensure termination when analyzing loops. In a pilot study, we compared widening after between one and ten iterations. We found that there was little added precision when applying widening after more than three iterations when trying to prove array indexes in bounds (our target application, discussed next). Thus we widen at that point in our implementation.

Limitations. Our implementation is sound with a few exceptions. In particular, it ignores calls to native methods and uses of reflection. It is also unsound in its handling of recursive method calls. If the return value of a recursive method is numeric, it is regarded as unconstrained. Potential side effects of the recursive calls are not modeled.

6 Evaluation

In this section, we present an empirical study of our family of analyses, focusing on the following research questions:

RQ1: Performance. How does the configuration affect analysis running time?
RQ2: Precision. How does the configuration affect analysis precision?
RQ3: Tradeoffs. How does the configuration affect precision and performance?

To answer these questions, we chose an important analysis client, array index out-of-bound analysis, and ran it on the DaCapo benchmark suite [6]. We vary each of the analysis features listed in Table 1, yielding 162 total configurations. To understand the impact of analysis features, we used multiple linear regression and logistic regression to model precision and performance (the dependent variables) in terms of analysis features and across programs (the independent variables). We also studied per-program data directly.

Overall, we found that using access paths is a significant boon to precision but costs little in performance, while using summary objects is the reverse, to the point that use of summary objects is a significant source of timeouts. Polyhedra add precision compared to intervals, and impose some performance cost, though only half as much as summary objects. Interestingly, when both summary objects and polyhedra together would result in a timeout, choosing the first tends to provide better precision over the second. Finally, bottom-up analysis harms precision compared to top-down analysis, especially when only summary objects are enabled, but yields little gain in performance.

6.1 Experimental Setup

We evaluated our analyses by using them to perform array index out of bounds analysis. More specifically, for each benchmark program, we counted how many array access instructions (`x[i]=y`, `y=x[i]`, etc.) an analysis configuration could verify were in bounds (i.e., `i<x.length`), and measured the time taken to perform the analysis.

Benchmarks. We analyzed all eleven programs from the DaCapo benchmark suite [6] version 2006-10-MR2. The first three columns of Table 2 list the programs' names, their size (number of IR instructions), and the number of array bounds checks they contain. The rest of the table indicates the fastest and most precise analysis configuration for each program; we discuss these results in Sect. 6.4. We ran each benchmark three times under each of the 162 analysis configurations. The experiments were performed on two 2.4 GHz single processor (with four logical cores) Intel Xeon E5-2609 servers, each with 128GB memory running Ubuntu 16.04 (LTS). On each server, we ran three analysis configurations in parallel, binding each process to a designated core.

Since many analysis configurations are time-intensive, we set a limit of 1 hour for running a benchmark under a particular configuration. All performance results reported are the median of the three runs. We also use the median precision result, though note the analyses are deterministic, so the precision does not vary except in the case of timeouts. Thus, we treat an analysis as not timing out as long as either two or three of the three runs completed, and otherwise it is a timeout. Among the 1782 median results (11 benchmarks, 162 configurations),

Table 2. Benchmarks and overall results.

		#	Best Performance			Best Precision		
Prog.	Size	Checks	Time (min)	# Checks	Percent	Time (min)	# Checks	Percent
			BU-AP-CI-CLAS-INT			TD-AP+SO-1TYP-CLAS-INT		
antlr	55734	1526	0.6	1176	77.1%	18.5	1306	85.6%
			BU-AP-CI-CLAS-INT			TD-AP-1TYP-SMUS-POL		
bloat	150197	4621	4.0	2538	54.9%	17.2	2795	60.5%
			BU-AP-CI-CLAS-INT			TD-AP-1TYP-SMUS-INT		
chart	167621	7965	3.3	5593	70.2%	7.7	5654	71.0%
			BU-AP-CI-ALLO-INT			TD-AP+SO-1TYP-SMUS-POL		
eclipse	18938	1043	0.2	896	85.9%	3.3	977	93.7%
			BU-AP-CI-CLAS-INT			TD-AP+SO-1CFA-SMUS-INT		
fop	33243	1337	0.4	998	74.6%	2.6	1137	85.0%
			BU-AP-CI-SMUS-INT			TD-AP+SO-CI-SMUS-INT		
hsqldb	19497	1020	0.3	911	89.3%	1.4	975	95.6%
			BU-AP-CI-SMUS-INT			TD-AP-1CFA-CLAS-POL		
jython	127661	4232	1.3	2667	63.0%	33.6	2919	69.0%
			BU-AP-CI-SMUS-INT			TD-AP+SO-1TYP-ALLO-INT		
luindex	69027	2764	1.8	1682	60.9%	46.8	2015	72.9%
			BU-AP-CI-CLAS-INT			TD-AP+SO-1CFA-ALLO-POL		
lusearch	20242	1062	0.2	912	85.9%	54.2	979	92.2%
			BU-AP-CI-CLAS-INT			TD-AP+SO-CI-CLAS-INT		
pmd	116422	4402	1.7	3153	71.6%	49.5	3301	75.0%
			BU-AP-CI-CLAS-INT			TD-AP+SO-1CFA-SMUS-POL		
xalan	20315	1043	0.2	912	87.4%	3.8	981	94.1%

667 of them (37%) timed out. The percentage of the configurations that timed out analyzing a program ranged from 0% (xalan) to 90% (chart).

Statistical Analysis. To answer RQ1 and RQ2, we constructed a model for each question using multiple linear regression. Roughly put, we attempt to produce a model of performance (RQ1) and precision (RQ2)—the *dependent variables*—in terms of a linear combination of analysis configuration options (i.e., one choice from each of the five categories given in Table 1) and the benchmark program (i.e., one of the eleven subjects from DaCapo)—the *independent variables*. We include the programs themselves as independent variables, which allows us to roughly factor out program-specific sources of performance or precision gain/loss (which might include size, complexity, etc.); this is standard in this sort of regression [45]. Our models also consider all two-way interactions among analysis options. In our scenario, a significant interaction between two option settings suggests that the combination of them has a different impact on the analysis precision and/or performance compared to their independent impact.

To obtain a model that best fits the data, we performed variable selection via the Akaike Information Criterion (AIC) [12], a standard measure of model quality. AIC drops insignificant independent variables to better estimate the impact of analysis options. The R^2 values for the models are good, with the lowest of any model being 0.71.

After performing the regression, we examine the results to discover potential trends. Then we draw plots to examine how those trends manifest in the different programs. This lets us study the whole distribution, including outliers and any non-linear behavior, in a way that would be difficult if we just looked at the regression model. At the same time, if we only looked at plots it would be hard to see general trends because there is so much data.

Threats to Validity. There are several potential threats to the validity of our study. First, the benchmark programs may not be representative of programs that analysis users are interested in. That said, the programs were drawn from a well-studied benchmark suite, so they should provide useful insights.

Second, the insights drawn from the results of the array index out-of-bound analysis may not reflect the trends of other analysis clients. We note that array bounds checking is a standard, widely used analysis.

Third, we examined a design space of 162 analysis configurations, but there are other design choices we did not explore. Thus, there may be other independent variables that have important effects. In addition, there may be limitations specific to our implementation, e.g., due to precisely how WALA implements points-to analysis. Even so, we relied on time-tested implementations as much as possible, and arrived at our choices of analysis features by studying the literature and conversing with experts. Thus, we believe our study has value even if further variables are worth studying.

Fourth, for our experiments we ran each analysis configuration three times, and thus performance variation may not be fully accounted for. While more trials would add greater statistical assurance, each trial takes about a week to run on our benchmark machines, and we observed no variation in precision across the trials. We did observe variations in performance, but they were small and did not affect the broader trends. In more detail, we computed the variance of the running time among a set of three runs of a configuration as `(max-min)/median` to calculate the variance. The average variance across all configurations is only 4.2%. The maximum total time difference (`max-min`) is 32 min, an outlier from eclipse. All the other time differences are within 4 min.

6.2 RQ1: Performance

Table 3 summarizes our regression model for performance. We measure performance as the time to run both the core analysis and perform array index out-of-bounds checking. If a configuration timed out while analyzing a program, we set its running time as one hour, the time limit (characterizing a lower bound on the configuration's performance impact). Another option would have been to

Table 3. Model of run-time performance in terms of analysis configuration options (Table 1), including two-way interactions. Independent variables for individual programs not shown. R^2 of 0.72.

Option	Setting	Est. (min)	CI	p-value
AO	TD	-	-	-
	BU	-1.98	[-6.3, 1.76]	0.336
	TD+BU	1.97	[-1.78, 6.87]	0.364
HA	AP+SO	-	-	-
	AP	-37.6	[-42.36, -32.84]	<0.001
	SO	0.15	[-4.60, 4.91]	0.949
CS	1TYP	-	-	-
	CI	-7.09	[-10.89, -3.28]	<0.001
	1CFA	1.62	[-2.19, 5.42]	0.405
OR	ALLO	-	-	-
	CLAS	-11.00	[-15.44, -6.56]	<0.001
	SMUS	-7.15	[-11.59, -2.70]	0.002
ND	POL	-	-	-
	INT	-16.51	[-19.56, -13.46]	<0.001
AO:HA	TD:AP+SO	-	-	-
	BU:AP	-5.31	[-9.35, -1.27]	0.01
	TD+BU:AP	-3.13	[-7.38, 1.12]	0.15
	BU:SO	0.11	[-3.92, 4.15]	0.956
	TD+BU:SO	-0.08	[-4.33, 4.17]	0.97
AO:OR	TD:ALLO	-	-	-
	BU:CLAS	-8.87	[-12.91, -4.83]	<0.001
	BU:SMUS	-4.23	[-8.27, -0.19]	0.04
	TD+BU:CLAS	-4.07	[-8.32, 0.19]	0.06
	TD+BU:SMUS	-2.52	[-6.77, 1.74]	0.247
AO:ND	TD:POL	-	-	-
	BU:INT	8.04	[4.73, 11.33]	<0.001
	TD+BU:INT	2.35	[-1.12, 5.82]	0.185
HA:CS	AP+SO:1TYP	-	-	-
	AP:1CFA	7.01	[2.83, 11.17]	<0.001
	AP:CI	3.38	[-0.79, 7.54]	0.112
	SO:CI	-0.20	[-4.37, 3.96]	0.924
	SO:1CFA	-0.21	[-4.37, 3.95]	0.921
HA:OR	AP+SO:ALLO	-	-	-
	AP:CLAS	9.55	[5.37, 13.71]	<0.001
	AP:SMUS	6.25	[2.08, 10.42]	<0.001
	SO:SMUS	0.07	[-4.09, 4.24]	0.973
	SO:CLAS	-0.43	[-4.59, 3.73]	0.839
HA:ND	AP+SO:POL	-	-	-
	AP:INT	6.94	[3.53, 10.34]	<0.001
	SO:INT	0.08	[-3.32, 3.48]	0.964
CS:OR	1TYP:ALLO	-	-	-
	CI:CLAS	4.76	[0.59, 8.93]	0.025
	CI:SMUS	4.02	[-0.15, 8.18]	0.05
	1CFA:CLAS	-3.09	[-7.25, 1.08]	0.147
	1CFA:SMUS	-0.52	[-4.68, 3.64]	0.807

leave the configuration out of the regression, but doing so would underrepresent the important negative contribution to performance.

In the top part of the table, the first column shows the independent variables and the second column shows a setting. One of the settings, identified by dashes in the remaining columns, is the baseline in the regression. We use the following settings as baselines: TD, AP+SO, 1TYP, ALLO, and POL. We chose the baseline according to what we expected to be the most precise settings. For

Table 4. Model of timeout in terms of analysis configuration options (Table 1). Independent variables for individual programs not shown. R^2 of 0.77.

Option	Setting	Coef.	CI	Exp(coef.)	p-value
AO	TD	-	-	-	-
	BU	-1.47	[-2.04, -0.92]	0.23	<0.001
	TD+BU	0.09	[-0.46, 0.65]	1.09	0.73
HA	AP+SO	-	-	-	-
	AP	-10.6	[-12.29, -9.05]	2.49E-5	<0.001
	SO	0.03	[-0.46, 0.53]	1.03	0.899
CS	1TYP	-	-	-	-
	CI	-0.89	[-1.46, -0.34]	0.41	0.002
	1CFA	0.94	[0.39, 1.49]	2.56	0.001
OR	ALLO	-	-	-	-
	CLAS	-3.84	[-4.59, -3.15]	0.02	<0.001
	SMUS	-1.78	[-2.36, -1.23]	0.17	<0.001
ND	POL	-	-	-	-
	INT	-3.73	[-4.40, -3.13]	0.02	<0.001

the other settings, the third column shows the estimated effect of that setting with all other settings (including the choice of program, each an independent variable) held fixed. For example, the fifth row of the table shows that AP (only) decreases overall analysis time by 37.6 min compared to AP+SO (and the other baseline settings). The fourth column shows the 95% confidence interval around the estimate, and the last column shows the p-value. As is standard, we consider p-values less than 0.05 (5%) significant; such rows are highlighted green.

The bottom part of the table shows the additional effects of two-way combinations of options compared to the baseline effects of each option. For example, the BU:CLAS row shows a coefficient of –8.87. We add this to the individual effects of BU (–1.98) and CLAS (–11.0) to compute that BU:CLAS is 21.9 min faster (since the number is negative) than the baseline pair of TD:ALLO. Not all interactions are shown, e.g., AO:CS is not in the table. Any interactions not included were deemed not to have meaningful effect and thus were dropped by the model generation process [12].

Setting the running time of a timed-out configuration as one hour in Table 3 may under-report a configuration's (negative) performance impact. For a more complete view, we follow the suggestion of Arcuri and Briand [3], and construct a model of success/failure using logistic regression. We consider "if a configuration timed out" as the categorical dependent variable, and the analysis configuration options and the benchmark programs as independent variables.

Table 4 summarizes our logistic regression model for timeout. The coefficients in the third column represent the change in log likelihood associated with each configuration setting, compared to the baseline setting. Negative coefficients indicate lower likelihood of timeout. The exponential of the coefficient, Exp(coef) in the fifth column, indicates roughly how strongly that configuration setting being turned on affects the likelihood relative to the baseline setting. For example, the third row of the table shows that BU is roughly 5 times less likely to time out compared to TD, a significant factor to the model.

Tables 3 and 4 present several interesting performance trends.

Summary Objects Incur a Significant Slowdown. Use of summary objects results in a very large slowdown, with high significance. We can see this in the AP row in Table 3. It indicates that using *only* AP results in an average 37.6-min speedup compared to the baseline AP+SO (while SO only had no significant difference from the baseline). We observed a similar trend in Table 4; use of summary objects has the largest effect, with high significance, on the likelihood of timeout. Indeed, 624 out of the 667 analyses that timed out had summary objects enabled (i.e., SO or AP+SO). We investigated further and found the slowdown from summary objects is mostly due to significantly larger number of dimensions included in the abstract state. For example, analyzing jython with AP-TD-CI-ALLO-INT has, on average, 11 numeric variables when analyzing a method, and the whole analysis finished in 15 min. Switching AP to SO resulted in, on average, 1473 variables per analyzed method and the analysis ultimately timed out.

The Polyhedral Domain is Slow, But Not as Slow as Summary Objects. Choosing INT over baseline POL nets a speedup of 16.51 min. This is the second-largest performance effect with high significance, though it is half as large as the effect of SO. Moreover, per Table 4, turning on POL is more likely to result in timeout; 409 out of 667 analyses that timed out used POL.

Heavyweight CS and OR Settings Hurt Performance, Particularly When Using Summary Objects. For CS settings, CI is faster than baseline 1TYP by 7.1 min, while there is not a statistically significant difference with 1CFA. For the OR settings, we see that the more lightweight representations CLAS and SMUS are faster than baseline ALLO by 11.00 and 7.15 min, respectively, when using baseline AP+SO. This makes sense because these representations have a direct effect on reducing the number of summary objects. Indeed, when summary objects are disabled, the performance benefit disappears: AP:CLAS and AP:SMUS add back 9.55 and 6.25 min, respectively.

Bottom-up Analysis Provides No Substantial Performance Advantage. Table 4 indicates that a BU analysis is less likely to time out than a TD analysis. However, the performance model in Table 3 does not show a performance advantage of bottom-up analysis: neither BU nor TD+BU provide a statistically significant impact on running time over baseline TD. Setting one hour for the configurations that timed out in the performance model might fail to capture the negative performance of top-down analysis. This observation underpins the utility of constructing a success/failure analysis to complement the performance model. In any case, we might have expected bottom-up analysis to provide a real performance advantage (Sect. 4.1), but that is not what we have observed.

6.3 RQ2: Precision

Table 5 summarizes our regression model for precision, using the same format as Table 3. We measure precision as the number of array indexes proven to be in

Table 5. Model of precision, measured as # of array indexes proved in bounds, in terms of analysis configuration options (Table 1), including two-way interactions. Independent variables for individual programs not shown. R^2 of 0.98.

Option	Setting	Est. (#)	CI	p-value
AO	TD	-	-	-
	TD+BU	-134.22	[-184.93, -83.50]	<0.001
	BU	-129.98	[-180.24, -79.73]	<0.001
HA	AP+SO	-	-	-
	SO	-94.46	[-166.79, -22.13]	0.011
	AP	-5.24	[-66.47, 55.99]	0.866
OR	ALLO	-	-	-
	CLAS	-90.15	[-138.80, -41.5]	<0.001
	SMUS	35.47	[-14.72, 85.67]	0.166
ND	POL	-	-	-
	INT	5.11	[-28.77, 38.99]	0.767
AO:HA	TD:AP+SO	-	-	-
	BU:SO	-686.79	[-741.82, -631.76]	<0.001
	TD+BU:SO	-630.99	[-687.41, -574.56]	<0.001
	TD+BU:AP	63.59	[14.71, 112.47]	0.011
	BU:AP	58.92	[11.75, 106.1]	0.014
AO:OR	TD:ALLO	-	-	-
	TD+BU:CLAS	156.31	[107.78, 204.83]	<0.001
	BU:CLAS	141.46	[94.13, 188.80]	<0.001
	BU:SMUS	-29.16	[-77.69, 19.37]	0.238
	TD+BU:SMUS	-29.25	[-79.23, 20.72]	0.251
HA:OR	AP+SO:ALLO	-	-	-
	SO:CLAS	-351.01	[-408.35, -293.67]	<0.001
	SO:SMUS	-72.23	[-131.99, -12.47]	0.017
	AP:SMUS	-16.88	[-67.20, 33.44]	0.51
	AP:CLAS	-8.81	[-57.84, 40.20]	0.724
HA:ND	AP+SO:POL	-	-	-
	AP:INT	-58.87	[-99.39, -18.35]	0.004
	SO:INT	-61.96	[-109.08, -14.84]	0.01

bounds. As recommended by Arcuri and Briand [3], we omit from the regression those configurations that timed out.[5] We see several interesting trends.

Access Paths are Critical to Precision. Removing access paths from the configuration, by switching from AP+SO to SO, yields significantly lower precision. We see this in the SO (only) row in the table, and in all of its interactions (i.e., SO:*opt* and *opt*:SO rows). In contrast, AP on its own is not statistically worse than AP+SO, indicating that summary objects often add little precision. This is unfortunate, given their high performance cost.

Bottom-up Analysis Harms Precision Overall, Especially for SO (Only). BU has a strongly negative effect on precision: 129.98 fewer checks compared to TD. Coupled with SO it fares even worse: BU:SO nets 686.79 fewer checks, and TD+BU:SO nets 630.99 fewer. For example, for xalan the most precise configuration, which uses TD and AP+SO, discharges 981 checks, while all configurations

[5] The alternative of setting precision to be 0 would misrepresent the general power of a configuration, particularly when combined with runs that did not time out. Fewer runs might reduce statistical power, however, which is captured in the model.

that instead use BU and SO on xalan discharge close to zero checks. The same basic trend holds for just about every program.

The Relational Domain Only Slightly Improves Precision. The row for INT is not statistically different from the baseline POL. This is a bit of a surprise, since by itself POL is strictly more precise than INT. In fact, it does improve precision empirically when coupled with either AP or SO—the interaction AP:INT and SO:INT reduces the number of checks. This sets up an interesting performance tradeoff that we explore in Sect. 6.4: using AP+SO with INT vs. using AP with POL.

More Precise Abstract Object Representation Improves Precision, But Context Sensitivity Does Not. The table shows CLAS discharges 90.15 fewer checks compared to ALLO. Examining the data in detail, we found this occurred because CLAS conflates all arrays of the same type as one abstract object, thus imprecisely approximating those arrays' lengths, in turn causing some checks to fail.

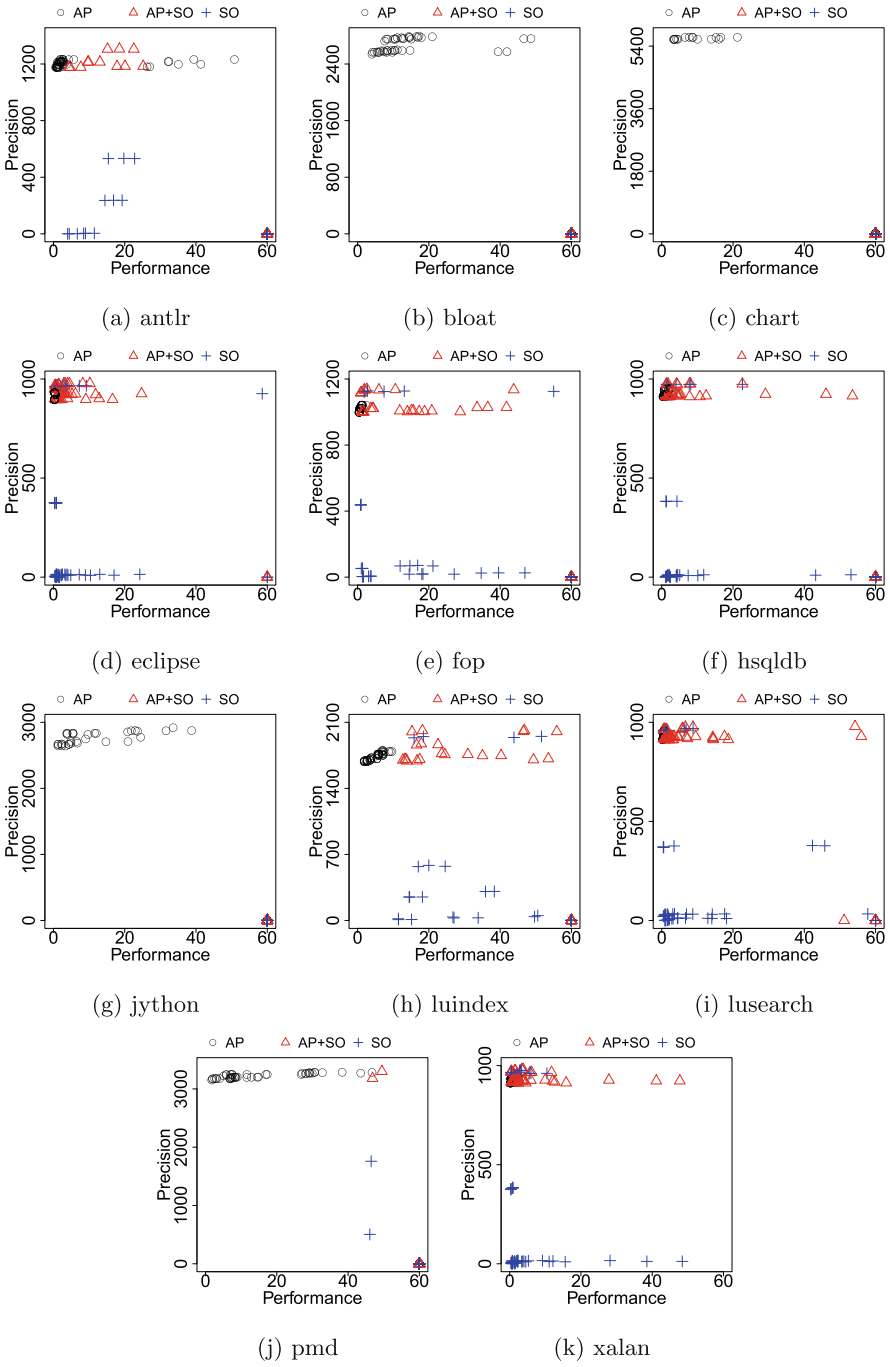
Also notice that context sensitivity (CS) does not appear in the model, meaning it does not significantly increase or decrease the precision of array bounds checking. This is interesting, because context-sensitivity is known to reduce points-to set size [35,49] (thus yielding more precise alias checks and dispatch targets). However, for our application this improvement has minimal impact.

6.4 RQ3: Tradeoffs

Finally, we examine how analysis settings affect the tradeoff between precision and performance. To begin out discussion, recall Table 2 (page 12), which shows the fastest configuration and the most precise configuration for each benchmark. Further, the table shows the configurations' running time, number of checks discharged, and percentage of checks discharged.

We see several interesting patterns in this table, though note the table shows just two data points and not the full distribution. First, the configurations in each column are remarkably consistent. The fastest configurations are all of the form BU-AP-CI-*-INT, only varying in the abstract object representation. The most precise configurations are more variable, but all include TD and some form of AP. The rest of the options differ somewhat, with different forms of precision benefiting different benchmarks. Finally, notice that, overall, the fastest configurations are much faster than the most precise configurations—often by an order of magnitude—but they are not that much less precise—typically by 5–10% points.

To delve further into the tradeoff, we examine, for each program, the overall performance and precision distribution for the analysis configurations, focusing on particular options (HA, AO, etc.). As settings of option HA have come up prominently in our discussion so far, we start with it and then move through the other options. Figure 1 gives per-benchmark scatter plots of this data. Each plotted point corresponds to one configuration, with its performance on the x-axis and number of discharged array bounds checks on the y-axis. We regard a configuration that times out as discharging no checks, so it is plotted at (60, 0).



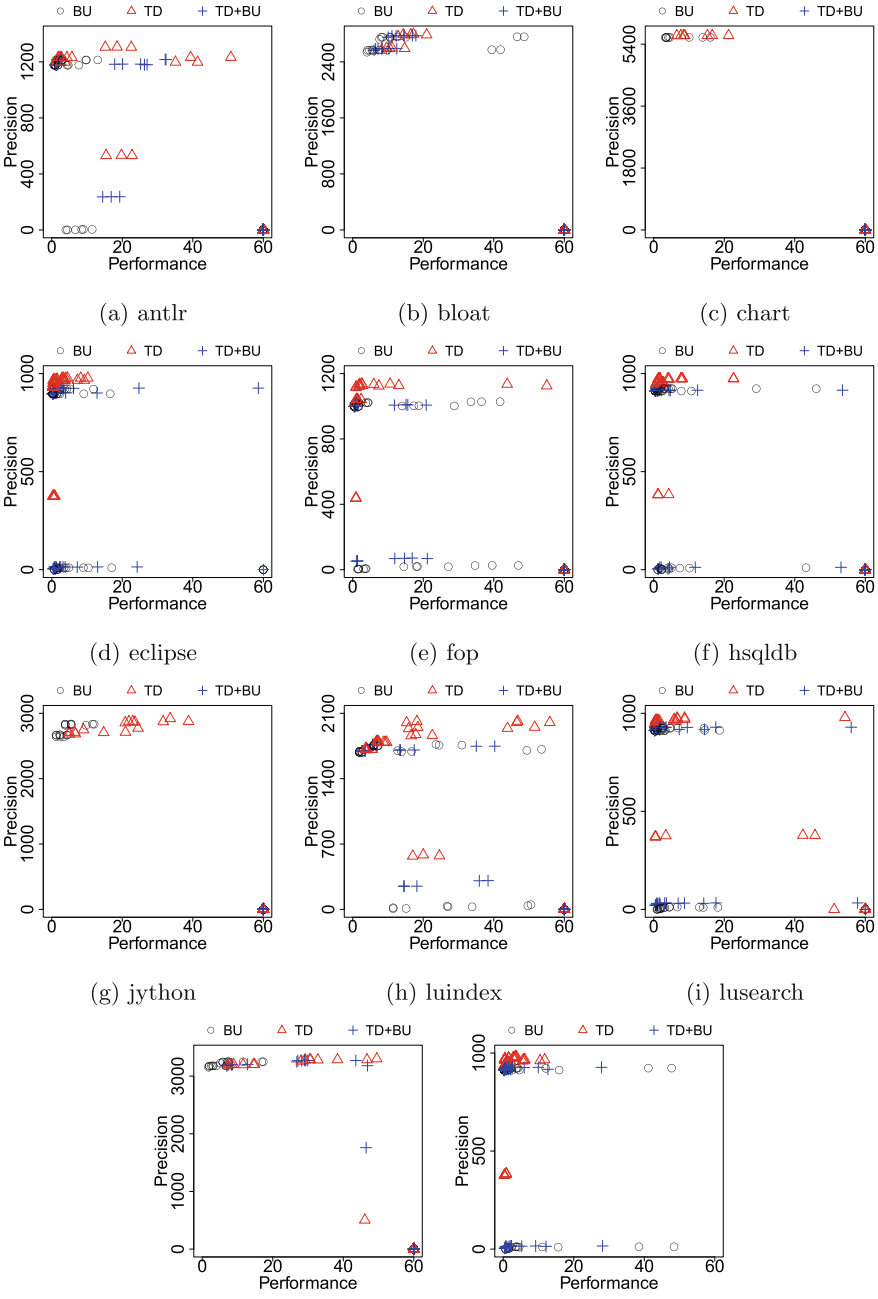
(a) antlr (b) bloat (c) chart

(d) eclipse (e) fop (f) hsqldb

(g) jython (h) luindex (i) lusearch

(j) pmd (k) xalan

Fig. 1. Tradeoffs: AP vs. SO vs. AP+SO.



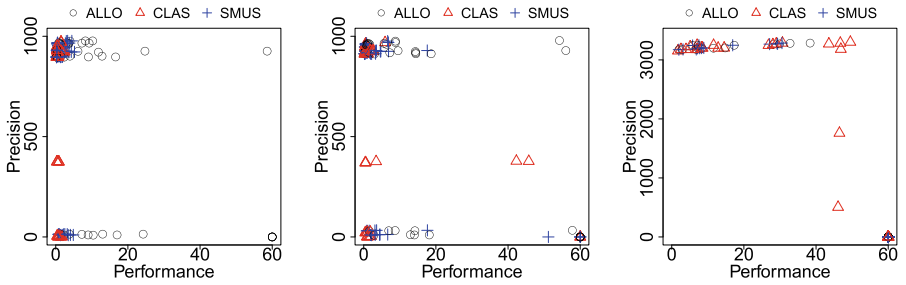
(a) antlr (b) bloat (c) chart

(d) eclipse (e) fop (f) hsqldb

(g) jython (h) luindex (i) lusearch

(j) pmd (k) xalan

Fig. 2. Tradeoffs: TD vs. BU vs. TD+BU.



(a) eclipse (b) lusearch (c) pmd

Fig. 3. Tradeoffs: ALLO vs. SMUS vs. CLAS.

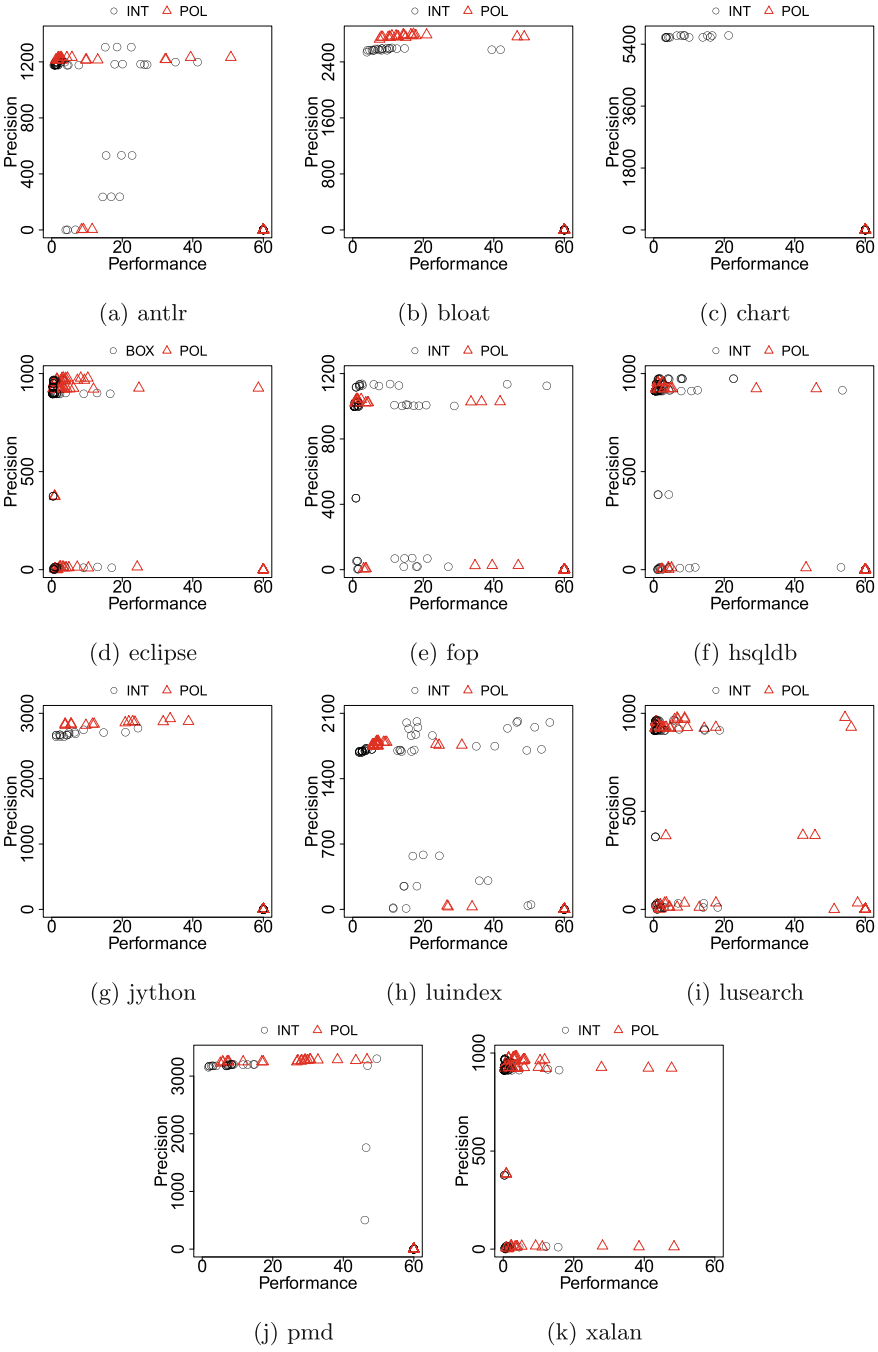
The shape of a point indicates the HA setting of the corresponding configuration: black circle for AP, red triangle for AP+SO, and blue cross for SO.

As a general trend, we see that *access paths improve precision and do little to harm performance; they should always be enabled.* More specifically, configurations using AP and AP+SO (when they do not time out) are always toward the top of the graph, meaning good precision. Moreover, the performance profile of SO and AP+SO is quite similar, as evidenced by related clusters in the graphs differing in the y-axis, but not the x-axis. In only one case did AP+SO time out when SO alone did not.[6]

On the flip side, *summary objects are a significant performance bottleneck for a small boost in precision.* On the graphs, we can see that the black AP circles are often among the most precise, while AP+SO tend to be the best (8/11 cases in Table 2). But AP are much faster. For example, for bloat, chart, and jython, only AP configurations complete before the timeout, and for pmd, all but four of the configurations that completed use AP.

Top-Down Analysis is Preferred: Bottom-up is less precise and does little to improve performance. Figure 2 shows a scatter plot of the precision/performance behavior of all configurations, distinguishing those with BU (black circles), TD (red triangles), and TD+BU (blue crosses). Here the trend is not as stark as with HA, but we can see that the mass of TD points is towards the upper-left of the plots, except for some timeouts, while BU and TD+BU have more configurations at the bottom, with low precision. By comparing the same (x,y) coordinate on a graph in this figure with the corresponding graph in the previous one, we can see options interacting. Observe that the cluster of black circles at the lower left for antlr in Fig. 2(a) correspond to SO-only configurations in Fig. 1(a), thus illustrating the strong negative interaction on precision of BU:SO we discussed in the previous subsection. The figures (and Table 2) also show that the best-performing configurations involve bottom-up analysis, but usually the

[6] In particular, for eclipse, configuration TD+BU-SO-1CFA-ALLO-POL finished at 59 min, while TD+BU-AP+SO-1CFA-ALLO-POL timed out.



(a) antlr (b) bloat (c) chart

(d) eclipse (e) fop (f) hsqldb

(g) jython (h) luindex (i) lusearch

(j) pmd (k) xalan

Fig. 4. Tradeoffs: INT vs. POL.

benefit is inconsistent and very small. And TD+BU does not seem to balance the precision/performance tradeoff particularly well.

Precise Object Representation Often Helps with Precision at a Modest Cost to Performance. Figure 3 shows a representative sample of scatter plots illustrating the tradeoff between ALLO, CLAS, and SMUS. In general, we see that the highest points tend to be ALLO, and these are more to the right of CLAS and SMUS. On the other hand, the precision gain of ALLO tends to be modest, and these usually occur (examining individual runs) when combining with AP+SO. However, summary objects and ALLO together greatly increase the risk of timeouts and low performance. For example, for eclipse the row of circles across the bottom are all SO-only.

The Precision Gains of POLY are More Modest than Gains Due to Using AP+SO (over AP). Figure 4 shows scatter plots comparing INT and POLY. We investigated several groupings in more detail and found an interesting interaction between the numeric domain and the heap abstraction: POLY is often better than INT for AP (only). For example, the points in the upper left of bloat use AP, and POLY is slightly better than INT. The same phenomenon occurs in luindex in the cluster of triangles and circles to the upper left. But INT does better further up and to the right in luindex. This is because these configurations use AP+SO, which times out when POLY is enabled. A similar phenomenon occurs for the two points in the upper right of pmd, and the most precise points for hsqldb. Indeed, when a configuration with AP+SO-INT terminates, it will be more precise than those with AP-POLY, but is likely slower. We manually inspected the cases where AP+SO-INT is more precise than AP-POLY, and found that it mostly is because of the limitation that access paths are dropped through method calls. AP+SO rarely terminates when coupled with POLY because of the very large number of dimensions added by summary objects.

7 Related Work

Our numeric analysis is novel in its focus on fully automatically identifying numeric invariants in real (heap-manipulating, method-calling) Java programs, while aiming to be sound. We know of no prior work that carefully studies precision and performance tradeoffs in this setting. Prior work tends to be much more imprecise and/or intentionally unsound, but scale better, or more precise, but not scale to programs as large as those in the DaCapo benchmark suite.

Numeric vs. Heap Analysis. Many abstract interpretation-based analyses focus on numeric properties or heap properties, but not both. For example, Calcagno et al. [13] uses separation logic to create a compositional, bottom-up heap analysis. Their client analysis for Java checks for NULL pointers [1], but not out-of-bounds array indexes. Conversely, the PAGAI analyzer [31] for LLVM explores abstract interpretation algorithms for precise invariants of numeric variables, but ignores the heap (soundly treating heap locations as $\top$).

Numeric Analysis in Heap-Manipulating Programs. Fu [25] first proposed the basic summary object heap abstraction we explore in this paper. The approach uses a points-to analysis [44] as the basis of generating abstract names for summary objects that are weakly updated [27]. The approach does not support strong updates to heap objects and ignores procedure calls, making unsound assumptions about effects of calls to or from the procedure being analyzed. Fu's evaluation on DaCapo only considered how often the analysis yields a non-$\top$ field, while ours considers how often the analysis can prove that an array index is in bounds, which is a more direct measure of utility. Our experiments strongly suggest that when modeled soundly and at scale, summary objects add enormous performance overhead while doing much less to assist precision when compared to strongly updatable access paths alone [21,52].

Some prior work focuses on inferring precise invariants about heap-allocated objects, e.g., relating the presence of an object in a collection to the value of one of the object's fields. Ferrera et al. [23,24] also propose a composed analysis for numeric properties of heap manipulating programs. Their approach is amenable to both points-to and shape analyses (e.g., TVLA [34]), supporting strong updates for the latter. DESKCHECK [39] and Chang and Rival [14,15] also aim to combine shape analysis and numeric analysis, in both cases requiring the analyst to specify predicates about the data structures of interest. Magill [37] automatically converts heap-manipulating programs into integer programs such that proving a numeric property of the latter implies a numeric shape property (e.g., a list's length) of the former. The systems just described support more precise invariants than our approach, but are less general or scalable: they tend to focus on much smaller programs, they do not support important language features (e.g., Ferrara's approach lacks procedures, DESKCHECK lacks loops), and may require manual annotation.

Clousot [22] also aims to check numeric invariants on real programs that use the heap. Methods are analyzed in isolation but require programmer-specified pre/post conditions and object invariants. In contrast, our interprocedural analysis is fully automated, requiring no annotations. Clousot's heap analysis makes local, optimistic (and unsound) assumptions about aliasing,[7] while our approach aims to be sound by using a global points-to analysis.

Measuring Analysis Parameter Tradeoffs. We are not aware of work exploring performance/precision tradeoffs of features in realistic abstract interpreters. Oftentimes, papers leave out important algorithmic details. The initial ASTRÉE paper [7] contains a wealth of ideas, but does not evaluate them systematically, instead reporting anecdotal observations about their particular analysis targets. More often, papers focus on one element of an analysis to evaluate, e.g., Logozzo [36] examines precision and performance tradeoffs useful for certain kinds of numeric analyses, and Ferrara [24] evaluates his technique using both intervals and octagons as the numeric domain. Regarding the latter, our paper shows that interactions with the heap abstraction can have a strong impact on

[7] Interestingly, Clousot's assumptions often, but not always, lead to sound results [16].

the numeric domain precision/performance tradeoff. Prior work by Smaragdakis et al. [49] investigates the performance/precision tradeoffs of various implementation decisions in points-to analysis. PADDLE [35] evaluates tradeoffs among different abstractions of heap allocation sites in a points-to analysis, but specifically only evaluates the heap analysis and not other analyses that use it.

8 Conclusion and Future Work

We presented a family of static numeric analyses for Java. These analyses implement a novel combination of techniques to handle method calls, heap-allocated objects, and numeric analysis. We ran the 162 resulting analysis configurations on the DaCapo benchmark suite, and measured performance and precision in proving array indexes in bounds. Using a combination of multiple linear regression and data visualization, we found several trends. Among others, we discovered that strongly updatable access paths are always a good idea, adding significant precision at very little performance cost. We also found that top-down analysis also tended to improve precision at little cost, compared to bottom-up analysis. On the other hand, while summary objects did add precision when combined with access paths, they also added significant performance overhead, often resulting in timeouts. The polyhedral numeric domain improved precision, but would time out when using a richer heap abstraction; intervals and a richer heap would work better.

The results of our study suggest several directions for future work. For example, for many programs, a much more expensive analysis often did not add much more in terms of precision; a pre-analysis that identifies the tradeoff would be worthwhile. Another direction is to investigate a more sparse representation of summary objects that retains their modest precision benefits, but avoids the overall blowup. We also plan to consider other analysis configuration options. Our current implementation uses an ahead-of-time points-to analysis to model the heap; an alternative solution is to analyze the heap along with the numeric analysis [43]. Concerning abstract object representation and context sensitivity, there are other potentially interesting choices, e.g., recency abstraction [5] and object sensitivity [40]. Other interesting dimensions to consider are field sensitivity [32] and widening, notably *widening with thresholds*. Finally, we plan to explore other effective ways to design hybrid top-down and bottom-up analysis [54], and investigate sparse inter-procedural analysis for better performance [42].

Acknowledgments. We thank Gagandeep Singh for his help in debugging ELINA. We thank Arlen Cox, Xavier Rival, and the anonymous reviewers for their detailed feedback and comments. This research was supported in part by DARPA under contracts FA8750-15-2-0104 and FA8750-16-C-0022.

References

1. Facebook Infer. http://fbinfer.com. Accessed 11 Nov 2016
2. Watson, T.J.: Libraries for Analysis (WALA). http://wala.sourceforge.net/, version 1.3
3. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: ICSE (2011)
4. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: Precise widening operators for convex polyhedra. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 337–354. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_19
5. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006). https://doi.org/10.1007/11823230_15
6. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: java benchmarking development and analysis. In: OOPSLA (2006)
7. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI (2003)
8. Bortz, A., Boneh, D.: Exposing private information by timing web applications. In: WWW (2007)
9. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: OOPSLA (2009)
10. Brodkin, J.: Huge portions of the web vulnerable to hashing denial-of-service attack (2011). http://arstechnica.com/business/2011/12/huge-portions-of-web-vulnerable-to-hashing-denial-of-service-attack/
11. Brumley, D., Boneh, D.: Remote timing attacks are practical. In: USENIX Security (2003)
12. Burnham, K.P., Anderson, D.R., Huyvaert, K.P.: AIC model selection and multimodel inference in behavioral ecology: some background, observations, and comparisons. Behav. Ecol. Sociobiol. **65**(1), 23–25 (2011)
13. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 1–66 (2011)
14. Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. In: POPL (2008)
15. Chang, B.Y.E., Rival, X.: Modular construction of shape-numeric analyzers. In: Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday (SAIRP) (2013)
16. Christakis, M., Müller, P., Wüstholz, V.: An experimental evaluation of deliberate unsoundness in a static program analyzer. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 336–354. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_19
17. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming (1976)
18. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
19. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL (1978)

20. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
21. De, A., D'Souza, D.: Scalable flow-sensitive pointer analysis for java with strong updates. In: ECOOP (2012)
22. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18070-5_2
23. Ferrara, P.: Generic combination of heap and value analyses in abstract interpretation. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 302–321. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_17
24. Ferrara, P., Müuller, P., Novacek, M.: Automatic inference of heap properties exploiting value domains. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 393–411. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_22
25. Fu, Z.: Modularly combining numeric abstract domains with points-to analysis, and a scalable static numeric analyzer for java. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 282–301. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_16
26. Goodin, D.: Long passwords are good, but too much length can be a dos hazard (2013). http://arstechnica.com/security/2013/09/long-passwords-are-good-but-too-much-length-can-be-bad-for-security/
27. 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
28. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI (2009)
29. Gulwani, S., Tiwari, A.: Computing procedure summaries for interprocedural analysis. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 253–267. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_18
30. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: PLDI (2010)
31. Henry, J., Monniaux, D., Moy, M.: Pagai: a path sensitive static analyser. Electron. Notes Theor. Comput. Sci. **289**, 15–25 (2012)
32. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: PASTE (2001)
33. 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
34. Lev-Ami, T., Sagiv, M.: TVLA: a system for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–301. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-45099-3_15
35. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Methodol. (TOSEM) **18**(1), 1–53 (2008)
36. Logozzo, F., Fähndrich, M.: Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In: SAC (2008)
37. Magill, S.: Instrumentation analysis: an automated method for producing numeric abstractions of heap-manipulating programs. Ph.D. thesis, School of Computer Science, Carnegie Mellon University (2010)

38. Mardziel, P., Magill, S., Hicks, M., Srivatsa, M.: Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. J. Comput. Secur. 21(4), 463–532 (2013)
39. McCloskey, B., Reps, T., Sagiv, M.: Statically inferring complex heap, array, and numeric invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 71–99. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_6
40. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. (TOSEM) **14**(1), 1–14 (2005)
41. Miné, A.: APRON numerical abstract domain library. http://apron.cri.ensmp.fr/library/
42. Oh, H., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for c-like languages. In: PLDI (2012)
43. Pioli, A., Hind, M.: Combining interprocedural pointer analysis and conditional constant propagation. Technical report, IBM T. J. Watson Research Center (1999)
44. Ryder, B.G.: Dimensions of precision in reference analysis of object-oriented programming languages. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 126–137. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36579-6_10
45. Seltman, H.: Experimental design and analysis (2015). http://www.stat.cmu.edu/~hseltman/309/Book/Book.pdf. e-book
46. Shivers, O.: Control-flow analysis of higher-order languages or taming lambda. Ph.D. thesis, School of Computer Science, Carnegie Mellon University (1991)
47. Singh, G., Püschel, M., Vechev, M.: ETH Library for Numerical Analysis. http://elina.ethz.ch and https://github.com/eth-srl/ELINA
48. Singh, G., Püschel, M., Vechev, M.T.: Fast polyhedra abstract domain. In: POPL (2017)
49. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: POPL (2011)
50. Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: NDSS (2000)
51. Wei, S., Mardziel, P., Ruef, A., Foster, J.S., Hicks, M.: Evaluating design tradeoffs in numeric static analysis for java (extended version). Technical report (2018). http://www.cs.umd.edu/~mwh/papers/jana-extended.pdf
52. Wei, S., Ryder, B.G.: State-sensitive points-to analysis for the dynamic behavior of javascript objects. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 1–26. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_1
53. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for java programs. In: OOPSLA (1999)
54. Zhang, X., Mangal, R., Naik, M., Yang, H.: Hybrid top-down and bottom-up interprocedural analysis. In: PLDI (2014)

An Abstract Interpretation Framework for Input Data Usage

Caterina Urban(✉) and Peter Müller

Department of Computer Science, ETH Zurich, Zurich, Switzerland
{caterina.urban,peter.mueller}@inf.ethz.ch

Abstract. Data science software plays an increasingly important role in critical decision making in fields ranging from economy and finance to biology and medicine. As a result, errors in data science applications can have severe consequences, especially when they lead to results that look plausible, but are incorrect. A common cause of such errors is when applications erroneously ignore some of their input data, for instance due to bugs in the code that reads, filters, or clusters it.

In this paper, we propose an abstract interpretation framework to automatically detect unused input data. We derive a program semantics that precisely captures data usage by abstraction of the program's operational trace semantics and express it in a constructive fixpoint form. Based on this semantics, we systematically derive static analyses that automatically detect unused input data by fixpoint approximation.

This clear design principle provides a framework that subsumes existing analyses; we show that secure information flow analyses and a form of live variables analysis can be used for data usage, with varying degrees of precision. Additionally, we derive a static analysis to detect single unused data inputs, which is similar to dependency analyses used in the context of backward program slicing. Finally, we demonstrate the value of expressing such analyses as abstract interpretation by combining them with an existing abstraction of compound data structures such as arrays and lists to detect unused chunks of the data.

1 Introduction

In the past few years, data science has grown considerably in importance and now heavily influences many domains, ranging from economy and finance to biology and medicine. As we rely more and more on data science for making decisions, we become increasingly vulnerable to programming errors.

Programming errors can cause frustration, especially when they lead to a program failure after hours of computation. However, programming errors that do not cause failures can have more serious consequences as code that produces an erroneous but plausible result gives no indication that something went wrong. A notable example is the paper "Growth in a Time of Debt" published in 2010 by economists Reinhart and Rogoff, which was widely cited in political debates and

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 683–710, 2018.
https://doi.org/10.1007/978-3-319-89884-1_24

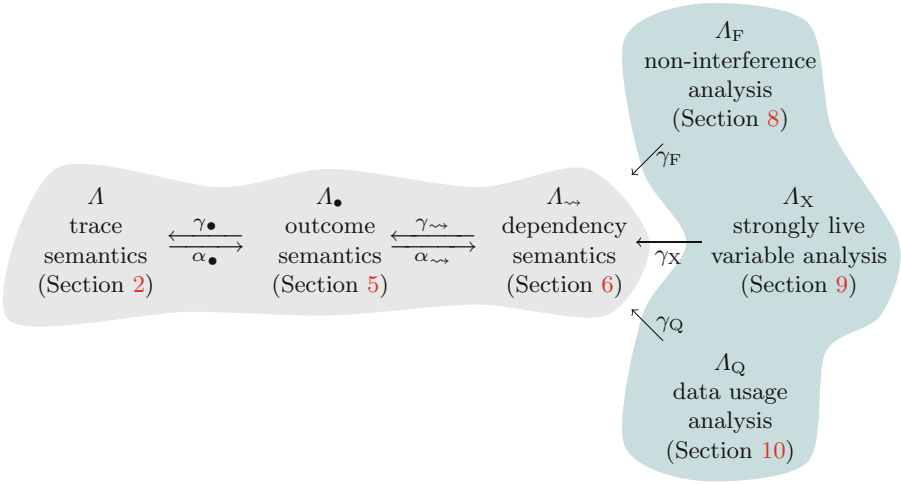


Fig. 1. Overview of the program semantics presented in the paper. The *dependency semantics*, derived by abstraction of the *trace semantics*, is sound and complete for data usage. Further sound but not complete abstractions are shown on the right.

was later demonstrated to be flawed. Notably, one of the flaws was a programming error, which *entirely excluded some data* from the analysis [23]. Its critics hold that this paper led to unjustified adoption of austerity policies for countries with various levels of public debt [30]. Programming errors in data analysis code for medical applications are even more critical [27]. It is thus paramount to achieve a high level of confidence in the correctness of data science code.

The likelihood that a programming error causes some input data to remain unused is particularly high for data science applications, where data goes through long pipelines of modules that acquire, filter, merge, and manipulate it. In this paper, we propose an abstract interpretation [14] framework to automatically detect *unused input data*. We characterize when a program uses (some of) its input data using the notion of *dependency* between the input data and the *outcome* of the program. Our notion of dependency accounts for non-determinism and non-termination. Thus, it encompasses notions of dependency that arise in many different contexts, such as secure information flow and program slicing [1], as well as provenance or lineage analysis [9], to name a few.

Following the theory of abstract interpretation [12], we systematically derive a new program semantics that precisely captures exactly the information needed to reason about input data usage, abstracting away from irrelevant details about the program behavior. Figure 1 gives an overview of our approach. The semantics is first expressed in a constructive fixpoint form over *sets of sets of traces*, by partitioning the operational trace semantics of a program based on its outcome (cf. *outcome semantics* in Fig. 1), and a further abstraction ignores intermediate state computations (cf. *dependency semantics* in Fig. 1). Starting the development of the semantics from the operational trace semantics enables a

uniform mathematical reasoning about programs semantics and program properties (Sect. 3). In particular, since input data usage is not a trace property or a subset-closed property [11] (Sect. 4), we show that a formulation of the semantics using sets of sets of traces is necessary for a sound validation of input data usage via fixpoint approximation [28].

This clear design principle provides a unifying framework for reasoning about existing analyses based on dependencies. We survey existing analyses and identify key design decisions that limit or facilitate their applicability to input data usage, and we assess their precision. We show that non-interference analyses [6] are sound for proving that a *terminating* program does not use *any* of its input data; although this is too strong a property in general. We prove that strongly live variable analysis [20] is sound for data usage even for non-terminating programs, albeit it is imprecise with respect to implicit dependencies between program variables. We then derive a more precise static analysis similar to dependency analyses used in the context of backward program slicing [37]. Finally, we demonstrate the value of expressing these analyses as abstract interpretations by combining them with an existing abstraction of compound data structures such as arrays and lists [16]. This allows us to detect unused chunks of the input data, and thus apply our work to realistic data science applications.

2 Trace Semantics

The *semantics* of a program is a mathematical characterization of its behavior when executed for all possible input data. We model the operational semantics of a program as a *transition system* $\langle \Sigma, \tau \rangle$ where Σ is a (potentially infinite) set of program states and the transition relation $\tau \subseteq \Sigma \times \Sigma$ describes the possible transitions between states [12,14]. Note that this model allows representing programs with (possibly unbounded) non-determinism. The set $\Omega \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \notin \tau\}$ is the set of *final states* of the program.

In the following, let $\Sigma^n \stackrel{\text{def}}{=} \{s_0 \cdots s_{n-1} \mid \forall i < n : s_i \in \Sigma\}$ be the set of all sequences of exactly n program states. We write ε to denote the empty sequence, i.e., $\Sigma^0 \stackrel{\text{def}}{=} \{\varepsilon\}$. Let $\Sigma^\star \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \Sigma^n$ be the set of all finite sequences, $\Sigma^+ \stackrel{\text{def}}{=} \Sigma^\star \setminus \Sigma^0$ be the set of all non-empty finite sequences, Σ^ω be the set of all infinite sequences, $\Sigma^{+\infty} \stackrel{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$ be the set of all non-empty finite or infinite sequences and $\Sigma^{\star\infty} \stackrel{\text{def}}{=} \Sigma^\star \cup \Sigma^\omega$ be the set of all finite or infinite sequences of program states. In the following, we write $\sigma\sigma'$ for the concatenation of two sequences $\sigma, \sigma' \in \Sigma^{\star\infty}$ (with $\sigma\varepsilon = \varepsilon\sigma = \sigma$, and $\sigma\sigma' = \sigma$ when $\sigma \in \Sigma^\omega$), $T^+ \stackrel{\text{def}}{=} T \cap \Sigma^+$ and $T^\omega \stackrel{\text{def}}{=} T \cap \Sigma^\omega$ for the selection of the non-empty finite sequences and the infinite sequences of $T \in \mathcal{P}(\Sigma^{\star\infty})$, and $T \;;\; T' \stackrel{\text{def}}{=} \{\sigma s \sigma' \mid s \in \Sigma \wedge \sigma s \in T \wedge s\sigma' \in T'\}$ for the merging of two sets of sequences $T \in \mathcal{P}(\Sigma^+)$ and $T' \in \mathcal{P}(\Sigma^{+\infty})$, when a finite sequence in T terminates with the initial state of a sequence in T'.

Given a transition system $\langle \Sigma, \tau \rangle$, a *trace* is a non-empty sequence of program states described by the transition relation τ, that is, $\langle s, s' \rangle \in \tau$ for each pair of

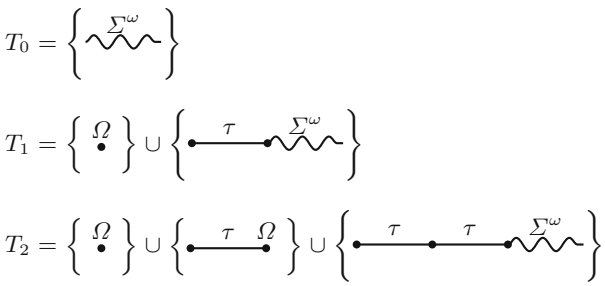


Fig. 2. First fixpoint iterates of the trace semantics Λ.

consecutive states $s, s' \in \Sigma$ in the sequence. The set of final states Ω and the transition relation τ can be understood as sets of traces of length one and length two, respectively. The *trace semantics* $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ generated by a transition system $\langle \Sigma, \tau \rangle$ is the union of all finite traces that are terminating with a final state in Ω, and all infinite traces. It can be expressed as a least fixpoint in the complete lattice $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+ \rangle$ [12]:

$$\begin{aligned} \Lambda &= \mathrm{lfp}^{\sqsubseteq}\, \Theta \\ \Theta(T) &\stackrel{\text{def}}{=} \Omega \cup (\tau \,;\, T) \end{aligned} \tag{1}$$

where the computational order is $T_1 \sqsubseteq T_2 \stackrel{\text{def}}{=} T_1^+ \subseteq T_2^+ \wedge T_1^\omega \supseteq T_2^\omega$. Figure 2 illustrates the first fixpoint iterates. The fixpoint iteration starts from the set of all infinite *sequences* of program states. At each iteration, the final program states in Ω are added to the set, and sequences already in the set are extended by prepending transitions to them. In this way, we *add* increasingly longer finite traces, and we *remove* infinite sequences of states with increasingly longer prefixes not forming traces. In particular, the i-th iterate builds all finite traces of length less than or equal to i, and selects all infinite sequences whose prefixes of length i form traces. At the limit we obtain all infinite traces and all finite traces that terminate in a final state in Ω. Note that Λ is *suffix-closed*.

The trace semantics Λ fully describes the behavior of a program. However, to reason about a particular property of a program, it is not necessary to consider all aspects of its behavior. In fact, reasoning is facilitated by the design of a semantics that abstracts away from irrelevant details about program executions. In the next sections, we define our property of interest and use abstract interpretation [14] to systematically derive, by successive abstractions of the trace semantics, a semantics that precisely captures such a property.

3 Input Data Usage

A *property* is specified by its extension, that is, the set of elements having such a property [14,15]. Thus, properties of program traces in $\Sigma^{+\infty}$ are sets of traces in

$\mathcal{P}(\Sigma^{+\infty})$, and properties of programs with trace semantics in $\mathcal{P}(\Sigma^{+\infty})$ are *sets of sets of traces* in $\mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$. Accordingly, a program P satisfies a property $\mathcal{H} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ if and only if its semantics $[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty})$ belongs to $\mathcal{H}$:

$$P \models \mathcal{H} \Leftrightarrow [\![P]\!] \in \mathcal{H} \tag{2}$$

Some program properties are defined in terms of individual program traces and can be equivalently expressed as trace properties. This is the case for the traditional safety [26] and liveness [4] properties of programs. In such a case, a program P satisfies a trace property $\mathcal{T}$ if and only if all traces in its semantics $[\![P]\!]$ belong to the property: $P \models \mathcal{T} \Leftrightarrow [\![P]\!] \subseteq \mathcal{T}$.

Program properties that establish a relation between different program traces cannot be expressed as trace properties [11]. Examples are security properties such as *non-interference* [21,35]. In this paper, we consider a closely related but more general property called *input data usage*, which expresses that *the outcome of a program does not depend on (some of) its input data*. The notion of *outcome* accounts for non-determinism as well as non-termination. Thus, our notion of dependency encompasses non-interference as well as notions of dependency that arise in many other contexts [1,9]. We further explore this in Sects. 8 to 10.

Let each program P with trace semantics $[\![P]\!]$ have a set I_P of input variables and a set O_P of output variables[1]. For simplicity, we can assume that these variables are all of the same type (e.g., boolean variables) and their values are all in a set V of possible values (e.g., $\mathrm{V} = \{\mathrm{T}, \mathrm{F}\}$ where T is the boolean value true and F is the boolean value false). Given a trace $\sigma \in [\![P]\!]$, we write $\sigma[0]$ to denote its initial state and $\sigma[\omega]$ to denote its outcome, that is, its final state if the trace is finite or $\perp$ if the trace is infinite. The input variables at the initial states of the traces of a program store the values of its input data: we write $\sigma[0](i)$ to denote the value of the input data stored in the input variable i at the initial state of the trace σ, and $\sigma_1[0] \neq_i \sigma_2[0]$ to denote that the initial states of two traces σ_1 and σ_2 disagree on the value of the input variable i but agree on the values of all other variables. The output variables at the final states of the finite traces of a program store its result: we write $\sigma[\omega](o)$ to denote the result stored in the output variable o at the final state of a finite trace σ. We can now formally define when an input variable $i \in \mathrm{I}_P$ is *unused* with respect to a program with trace semantics $[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty})$:

$$\begin{aligned} \textsc{unused}_i([\![P]\!]) \stackrel{\text{def}}{=}\ & \forall \sigma \in [\![P]\!], v \in \mathrm{V} : \sigma[0](i) \neq v \Rightarrow \\ & \exists \sigma' \in [\![P]\!] : \sigma'[0] \neq_i \sigma[0] \wedge \sigma'[0](i) = v \wedge \sigma[\omega] = \sigma'[\omega] \end{aligned} \tag{3}$$

Intuitively, an input variable i is unused if all feasible program outcomes (e.g., the outcome $\sigma[\omega]$ of a trace σ) are feasible from all possible initial values of i (i.e., for all possible initial values v of i that differ from the initial value of i in σ, there exists a trace with initial value v for i that has the same outcome $\sigma[\omega]$). In other words, the outcome of the program is the same independently of

[1] The approach can be easily extended to infinite inputs and/or outputs via abstractions such as the one later presented in Sect. 11.

```
1  english = input()
2  math = input()
3  science = input()
4  bonus = input()
5
6  passing = True
7  if not english: english = False              # english should be passing
8  if not math: passing = bonus
9  if not math: passing = bonus                 # math should be science
10
11 print(passing)
```

Fig. 3. Simple program to check if a student has passed three school subjects. The programmer has made two mistakes at line 7 and at line 9, which cause the input data stored in the variables `english` and `science` to be unused.

the initial value of the input variable i. Note that this definition accounts for non-determinism (since it considers each program outcome independently) and non-termination (since a program outcome can be $\perp$).

Example 1. Let us consider the simple program P in Fig. 3. Based on the input variables `english`, `math`, and `science` (cf. lines 1–3), the program is supposed to check if a student has passed all three considered school subjects and store the result in the output variable `passing` (cf. line 11). For mathematics and science, the student is allowed a bonus based on the input variable `bonus` (cf. line 8 and 9). However, the programmer has made two mistakes at line 7 and at line 9, which cause the input variables `english` and `science` to be unused.

Let us now consider the input variable `science`. The trace semantics of the program (simplified to consider only the variables `science` and `passing`) is:

$$[\![P]\!]_{\texttt{science}} = \{(\mathrm{T}\_)\ldots(\mathrm{TT}), (\mathrm{T}\_)\ldots(\mathrm{TF}), (\mathrm{F}\_)\ldots(\mathrm{FT}), (\mathrm{F}\_)\ldots(\mathrm{FF})\}$$

where each state $(v_1 v_2)$ shows the boolean value v_1 of `science` and v_2 of `passing`, and _ denotes any boolean value. We omitted the trace suffixes for brevity. The input variable `science` is *unused*, since each result value (T or F) for `passing` is feasible from all possible initial values of `science`. Note that all other outcomes of the program (i.e., non-termination) are not feasible.

Let us now consider the input variable `math`. The trace semantics of the program (now simplified to only consider `math` and `passing`) is the following:

$$[\![P]\!]_{\texttt{math}} = \{(\mathrm{T}\_)\ldots(\mathrm{TT}), (\mathrm{F}\_)\ldots(\mathrm{FT}), (\mathrm{F}\_)\ldots(\mathrm{FF})\}$$

In this case, the input variable `math` is used since only the initial state (F_) yields the result value F for `passing` (in the final state (FF)). ■

The input data usage property $\mathcal{N}$ can now be formally defined as follows:

$$\mathcal{N} \stackrel{\text{def}}{=} \left\{[\![P]\!] \in \mathcal{P}\left(\Sigma^{+\infty}\right) \mid \forall i \in \mathrm{I}_P : \textsc{unused}_i([\![P]\!])\right\} \tag{4}$$

which states that the outcome of a program does not depend on *any* input data. In practice one is interested in weaker input data usage properties for a subset

J of the input variables, i.e., $\mathcal{N}_J \stackrel{\text{def}}{=} \{[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty}) \mid \forall i \in J \subseteq \mathrm{I}_P : \textsc{unused}_i([\![P]\!])\}$.

In the following, we use abstract interpretation to reason about input data usage. In the next section, we discuss the challenges to the application of the standard abstract interpretation framework that emerge from the fact that input data usage cannot be expressed as a trace property.

4 Sound Input Data Usage Validation

In the standard framework of abstract interpretation, one defines a semantics that precisely captures a property $\mathcal{S}$ of interest by abstraction of the trace semantics Λ [12]. Then, further abstractions $\Lambda^\sharp$ provide sound over-approximations $\gamma(\Lambda^\sharp)$ of Λ (by means of a concretization function γ): $\Lambda \subseteq \gamma(\Lambda^\sharp)$. For a *trace property*, an over-approximation $\gamma([\![P]\!]^\sharp)$ of the semantics $[\![P]\!]$ of a program P allows a sound validation of the property: since $[\![P]\!] \subseteq \gamma([\![P]\!]^\sharp)$, we have that $\gamma([\![P]\!]^\sharp) \subseteq \mathcal{S} \Rightarrow [\![P]\!] \subseteq \mathcal{S}$ and so, if $\gamma([\![P]\!]^\sharp) \subseteq \mathcal{S}$, we can conclude that $P \models \mathcal{S}$ (cf. Sect. 3). This conclusion is also valid for all other *subset-closed* properties [11]: since by definition $\gamma([\![P]\!]^\sharp) \in \mathcal{S} \Rightarrow \forall T \subseteq \gamma([\![P]\!]^\sharp) : T \in \mathcal{S}$, we have that $\gamma([\![P]\!]^\sharp) \in \mathcal{S} \Rightarrow [\![P]\!] \in \mathcal{S}$ (and so we can conclude that $P \models \mathcal{S}$). However, for program properties that are not subset-closed, we have that $\gamma([\![P]\!]^\sharp) \in \mathcal{S} \not\Rightarrow [\![P]\!] \in \mathcal{S}$ [28] and so we cannot conclude that $P \models \mathcal{S}$, even if $\gamma([\![P]\!]^\sharp) \in \mathcal{S}$ (cf. Eq. 2).

We have seen in the previous section that input data usage is not a trace property. The example below shows that it is *not* a subset-closed property either.

Example 2. Let us consider again the program P and its semantics $[\![P]\!]_{\texttt{science}}$ and $[\![P]\!]_{\texttt{math}}$ shown in Example 1. We have seen in Example 1 that the semantics $[\![P]\!]_{\texttt{science}}$ belongs to the data usage property $\mathcal{N}$: $[\![P]\!]_{\texttt{science}} \in \mathcal{N}$. Let us consider now the following subset T of $[\![P]\!]_{\texttt{science}}$:

$$T = \{(\mathtt{T}\_) \ldots (\mathtt{TT}), (\mathtt{F}\_) \ldots (\mathtt{FT}), (\mathtt{F}\_) \ldots (\mathtt{FF})\}$$

In this case, the input variable `science` is used. Indeed, we can observe that T coincides with $[\![P]\!]_{\texttt{math}}$ (except for the considered input variable). Thus $T \notin \mathcal{N}$ even though $T \subseteq [\![P]\!]_{\texttt{science}}$. ■

Since input data usage is not subset-closed, we are in the unfortunate situation that we cannot use the standard abstract interpretation framework to soundly prove that a program does not use (some of) its input data using an over-approximation of the semantics of the program: $\gamma([\![P]\!]^\sharp) \in \mathcal{N}_J \not\Rightarrow [\![P]\!] \in \mathcal{N}_J$.

We solve this problem in the next section, by lifting the trace semantics $[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty})$ of a program P (i.e., a set of traces) to a set of sets of traces $(\!|P|\!) \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ [28]. In this setting, a program P satisfies a property $\mathcal{H}$ if and only if its semantics $(\!|P|\!)$ is a subset of $\mathcal{H}$:

$$P \models \mathcal{H} \Leftrightarrow (\!|P|\!) \subseteq \mathcal{H} \tag{5}$$

As we will explain in the next section, now an over-approximation $\gamma(\llparenthesis P \rrparenthesis^\natural)$ of $\llparenthesis P \rrparenthesis$ allows again a sound validation of the property: since $\llparenthesis P \rrparenthesis \subseteq \gamma(\llparenthesis P \rrparenthesis^\natural)$, we have that $\gamma(\llparenthesis P \rrparenthesis^\natural) \subseteq \mathcal{H} \Rightarrow \llparenthesis P \rrparenthesis \subseteq \mathcal{H}$ (and so we can conclude that $P \models \mathcal{H}$).

More specifically, in the next section, we define a program semantics $\llparenthesis P \rrparenthesis$ that precisely captures which subset J of the input variables is unused by a program P. In later sections, we present further abstractions $\llparenthesis P \rrparenthesis^\natural$ that over-approximate the subset of the input variables that *may be used* by P, and thus allows a sound validation of an *under-approximation* $J^\natural$ of J: $\gamma(\llparenthesis P \rrparenthesis^\natural) \subseteq \mathcal{N}_{J^\natural} \Rightarrow \llparenthesis P \rrparenthesis \subseteq \mathcal{N}_{J^\natural}$. In other words, this means that every input variable reported as unused by an abstraction is indeed not used by the program.

5 Outcome Semantics

We lift the trace semantics Λ to a set of sets of traces by *partitioning*. The *partitioning abstraction* $\alpha_Q \colon \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})) \to \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ of a set of traces T is:

$$\alpha_Q(T) \stackrel{\text{def}}{=} \{T \cap C \mid C \in Q\} \tag{6}$$

where $Q \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ is a *partition* of sequences of program states.

More specifically, to reason about input data usage of a program P, we lift the trace semantics $[\![P]\!]$ to $\llparenthesis P \rrparenthesis$ by partitioning it into sets of traces that yield the same program outcome. The key insight behind this idea is that, given an input variable i, the initial states of all traces in a partition give all initial values for i that yield a program outcome; the variable i is unused if and only if these initial values are all the possible values for i (or the set of values is empty because the outcome is unfeasible, cf. Eq. 3). Thus, if the trace semantics $[\![P]\!]$ of a program P belongs to the input data usage property $\mathcal{N}_J$, then each partition in $\llparenthesis P \rrparenthesis$ must also belong to $\mathcal{N}_J$, and vice versa: we have that $[\![P]\!] \in \mathcal{N}_J \Leftrightarrow \llparenthesis P \rrparenthesis \subseteq \mathcal{N}_J$, which is precisely what we want (cf. Eq. 5).

Let $T^+_{o=v}$ denote the subset of the finite sequences of program states in $T \in \mathcal{P}(\Sigma^{+\infty})$ with value v for the output variable o in their outcome (i.e., their final state): $T^+_{o=v} \stackrel{\text{def}}{=} \{\sigma \in T^+ \mid \sigma[\omega](o) = v\}$. We define the *outcome partition* $O \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ of sequences of program states:

$$O \stackrel{\text{def}}{=} \left\{ \Sigma^+_{o_1=v_1,\ldots,o_k=v_k} \mid v_1, \ldots, v_k \in \mathrm{V} \right\} \cup \{\Sigma^\omega\}$$

where V is the set of possible values of the output variables $o_1, \ldots, o_k$ (cf. Sect. 3). The partition contains all sets of finite sequences that agree on the values of the output variables in their outcome, and all infinite sequences of program states (i.e., all sequences with outcome $\perp$). We instantiate α_Q above with the outcome partition to obtain the *outcome abstraction* $\alpha_\bullet \colon \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})) \to \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$:

$$\alpha_\bullet(T) \stackrel{\text{def}}{=} \left\{ T^+_{o_1=v_1,\ldots,o_k=v_k} \mid v_1, \ldots, v_k \in \mathrm{V} \right\} \cup \{T^\omega\} \tag{7}$$

Example 3. The program P of Example 1 has only one output variable `passing` with boolean value T or F. Let us consider again the trace semantics $[\![P]\!]_{\texttt{math}}$ shown in Example 1. Its outcome abstraction $\alpha_\bullet([\![P]\!]_{\texttt{math}})$ is:

$$\alpha_\bullet([\![P]\!]_{\texttt{math}}) = \{\emptyset, \{(\texttt{F}\_)\ldots(\texttt{FF})\}, \{(\texttt{T}\_)\ldots(\texttt{TT}), (\texttt{F}\_)\ldots(\texttt{FT})\}\}$$

Note that all traces with different result values for the output variable `passing` belong to different sets of traces (i.e., partitions) in $\alpha_\bullet([\![P]\!]_{\texttt{math}})$. The empty set corresponds to the (unfeasible) non-terminating outcome of the program. ■

We can now use the outcome abstraction $\alpha_\bullet$ to define the *outcome semantics* $\Lambda_\bullet \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ as an abstraction of the trace semantics Λ:

Definition 1. *The* outcome semantics $\Lambda_\bullet \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ *is defined as:*

$$\Lambda_\bullet \stackrel{\text{def}}{=} \alpha_\bullet(\Lambda) \tag{8}$$

where $\alpha_\bullet$ is the outcome abstraction (cf. Eq. 7) and $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ is the trace semantics (cf. Eq. 1).

The outcome semantics contains the set of all infinite traces and all sets of finite traces that agree on the value of the output variables in their outcome.

In the following, we express the outcome semantics $\Lambda_\bullet$ in a constructive fixpoint form. This allows us to later derive further abstractions of $\Lambda_\bullet$ by *fixpoint transfer* and *fixpoint approximation* [12]. Given a set of sets of traces S, we write $S^+_{o=v} \stackrel{\text{def}}{=} \{T \in S \mid T = T^+_{o=v}\}$ for the selection of the sets of traces in S that agree on the value v of the output variable o in their outcome, and $S^\omega \stackrel{\text{def}}{=} \{T \in S \mid T = T^\omega\}$ for the selection of the sets of infinite traces in S. When $S^+_{o=v}$ (resp. S^ω) contains a single set of traces T, we abuse notation and write $S^+_{o=v}$ (resp. S^ω) to also denote T. The following result gives a fixpoint definition of $\Lambda_\bullet$ in the complete lattice $\langle \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})), \dot{\sqsubseteq}, \dot{\cup}, \dot{\cap}, \{\Sigma^\omega, \emptyset\}, \{\emptyset, \Sigma^+\}\rangle$, where the computational order $\dot{\sqsubseteq}$ is defined (similarly to $\sqsubseteq$, cf. Sect. 2) as:

$$S_1 \mathrel{\dot{\sqsubseteq}} S_2 \stackrel{\text{def}}{=} \bigwedge_{v_1,\ldots,v_k \in \mathrm{V}} {S_1}^+_{o_1=v_1,\ldots,o_k=v_k} \subseteq {S_2}^+_{o_1=v_1,\ldots,o_k=v_k} \wedge S_1^\omega \supseteq S_2^\omega$$

Theorem 1. *The outcome semantics $\Lambda_\bullet \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ can be expressed as a least fixpoint in $\langle \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})), \dot{\sqsubseteq}, \dot{\cup}, \dot{\cap}, \{\Sigma^\omega, \emptyset\}, \{\emptyset, \Sigma^+\}\rangle$ as:*

$$\begin{aligned} \Lambda_\bullet &= \mathrm{lfp}^{\dot{\sqsubseteq}}\, \Theta_\bullet \\ \Theta_\bullet(S) &\stackrel{\text{def}}{=} \{\Omega_{o_1=v_1,\ldots,o_k=v_k} \mid v_1,\ldots,v_k \in \mathrm{V}\} \mathbin{\dot{\cup}} \{\tau\,;T \mid T \in S\} \end{aligned} \tag{9}$$

where $S_1 \mathbin{\dot{\cup}} S_2 \stackrel{\text{def}}{=} \left\{ {S_1}^+_{o_1=v_1,\ldots,o_k=v_k} \cup {S_2}^+_{o_1=v_1,\ldots,o_k=v_k} \mid v_1,\ldots,v_k \in \mathrm{V} \right\} \cup S_1^\omega \cup S_2^\omega$.

Figure 4 illustrates the first fixpoint iterates of the outcome semantics for a single output variable o. The fixpoint iteration starts from the partition containing the set of all infinite sequences of program states and the empty set (which

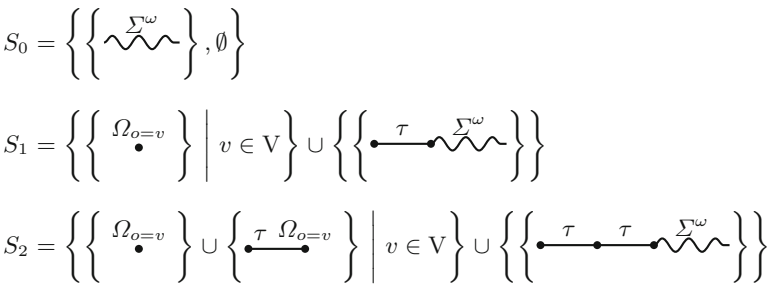


Fig. 4. First iterates of the outcome semantics $\Lambda_\bullet$ for a single output variable o.

represents an empty set of finite traces). At the first iteration, the empty set is replaced with a partition of the final states Ω based on the value v of the output variable o, while the infinite sequences are extended by prepending transitions to them (similarly to the trace semantics, cf. Eq. 1). At the next iterations, all sequences contained in each partition are further extended, and the final states that agree on the value v of o are again added to the matching set of traces that agree on v in their outcome. At the limit, we obtain a partition containing the set of all infinite traces and all sets of finite traces that agree on the value v of the output variable o in their outcome.

To prove Theorem 1 we first need to show that the outcome abstraction $\alpha_\bullet$ preserves least upper bounds of non-empty sets of sets of traces.

Lemma 1. *The outcome abstraction $\alpha_\bullet$ is Scott-continuous.*

Proof. We need to show that for any non-empty ascending chain C of sets of traces with least upper bound $\sqcup C$, we have that $\alpha_\bullet(\sqcup C) = \dot{\sqcup}\{\alpha_\bullet(T) \mid T \in C\}$, that is, $\alpha_\bullet(\sqcup C)$ is the least upper bound of $\alpha_\bullet(C)$, the image of C via $\alpha_\bullet$.

First, we know that $\alpha_\bullet$ is monotonic, i.e., for any two sets of traces T_1 and T_2 we have $T_1 \sqsubseteq T_2 \Rightarrow \alpha_\bullet(T_1) \dot{\sqsubseteq} \alpha_\bullet(T_2)$. Since $\sqcup C$ is the least upper bound of C, for any set T in C we have that $T \sqsubseteq \sqcup C$ and, since $\alpha_\bullet$ is monotonic, we have that $\alpha_\bullet(T) \dot{\sqsubseteq} \alpha_\bullet(\sqcup C)$. Thus $\alpha(\sqcup C)$ is an upper bound of $\{\alpha_\bullet(T) \mid T \in C\}$.

To show that $\alpha(\sqcup C)$ is the least upper bound of $\alpha_\bullet(C)$, we need to show that for any other upper bound U of $\alpha_\bullet(C)$ we have $\alpha_\bullet(\sqcup C) \dot{\sqsubseteq} U$. Let us assume by absurd that $\alpha_\bullet(\sqcup C) \not\dot{\sqsubseteq} U$. Then, there exists $T_1 \in \alpha_\bullet(\sqcup C)$ and $T_2 \in U$ such that $T_1 \not\sqsubseteq T_2$: $T_1^+ \supset T_2^+$ or $T_1^\omega \subset T_2^\omega$. Let us assume that $T_1^+ \supset T_2^+$. By definition of $\alpha_\bullet$, we observe that T_1 is a partition of $\dot{\sqcup} C$ and, since $\dot{\sqcup} C$ is the least upper bound of C, U cannot be an upper bound of $\alpha_\bullet(C)$ (since T_2 does not contain enough finite traces). Similarly, if $T_1^\omega \subset T_2^\omega$, then U cannot be an upper bound of $\alpha_\bullet(C)$ (since T_2 contains too many infinite traces). Thus, we must have $\alpha_\bullet(\sqcup C) \dot{\sqsubseteq} U$ and we can conclude that $\alpha(\sqcup C)$ is the least upper bound of $\alpha_\bullet(C)$. □

We can now prove Theorem 1 by Kleenian fixpoint transfer [12].

Proof (Sketch). The proof follows by Kleenian fixpoint transfer. We have that $\langle \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})), \sqsubseteq, \sqcup, \sqcap, \{\Sigma^\omega, \emptyset\}, \{\emptyset, \Sigma^+\}\rangle$ is a complete lattice and that $\phi^{+\infty}$ (cf. Eq. 1) and $\Theta_\bullet$ (cf. Eq. 8) are monotonic function. Additionally, we have that the outcome abstraction $\alpha_\bullet$ (cf. Eq. 7) is Scott-continuous (cf. Lemma 1) and such that $\alpha_\bullet(\Sigma^\omega) = \{\Sigma^\omega, \emptyset\}$ and $\alpha_\bullet \circ \phi^{+\infty} = \Theta_\bullet \circ \alpha_\bullet$. Then, by Kleenian fixpoint transfer, we have that $\alpha_\bullet(\Lambda) = \alpha_\bullet(\mathrm{lfp}^{\sqsubseteq}\ \phi^{+\infty}) = \mathrm{lfp}^{\dot\sqsubseteq}\ \Theta_\bullet$. Thus, we can conclude that $\Lambda_\bullet = \mathrm{lfp}^{\dot\sqsubseteq}\ \Theta_\bullet$. □

Finally, we show that the outcome semantics $\Lambda_\bullet$ is sound and complete for proving that a program does not use (a subset of) its input variables.

Theorem 2. *A program does not use a subset J of its input variables if and only if its outcome semantics $\Lambda_\bullet$ is a subset of $\mathcal{N}_J$:*

$$P \models \mathcal{N}_J \Leftrightarrow \Lambda_\bullet \subseteq \mathcal{N}_J$$

Proof (Sketch). The proof follows immediately from the definition of $\mathcal{N}_J$ (cf. Eq. 3 and Sect. 4) and the definition of $\Lambda_\bullet$ (cf. Eq. 8). □

Example 4. Let us consider again the program P and its semantics $[\![P]\!]_{\texttt{science}}$ shown in Example 1. The corresponding outcome semantics $\alpha_\bullet([\![P]\!]_{\texttt{science}})$ is:

$$\alpha_\bullet([\![P]\!]_{\texttt{science}}) = \{\emptyset, \{(\mathrm{T}\_)\ldots(\mathrm{TF}), (\mathrm{F}\_)\ldots(\mathrm{FF})\}, \{(\mathrm{T}\_)\ldots(\mathrm{TT}), (\mathrm{F}\_)\ldots(\mathrm{FT})\}\}$$

Note that all sets of traces in $\alpha_\bullet([\![P]\!]_{\texttt{science}})$ belong to $\mathcal{N}_{\{\texttt{science}\}}$: the initial states of all traces in a non-empty partition contain all possible initial values (T or F) for the input variable `science`. Thus, P satisfies $\mathcal{N}_{\{\texttt{science}\}}$ and, indeed, the input variable `science` is unused by P. ■

As discussed in Sect. 4, we now can again use the standard framework of abstract interpretation to soundly over-approximate $\Lambda_\bullet$ and prove that a program does not use (some of) its input data. In the next section, we propose an abstraction that remains sound and complete for input data usage. Further sound but not complete abstractions are presented in later sections.

6 Dependency Semantics

We observe that, to reason about input data usage, it is not necessary to consider all intermediate state computations between the initial state of a trace and its outcome. Thus, we can further abstract the outcome semantics $\Lambda_\bullet$ into a set $\Lambda_{\rightsquigarrow}$ of (dependency) relations between initial states and outcomes of a set of traces.

We lift the abstraction defined for this purpose on sets of traces [12] to $\alpha_{\rightsquigarrow} : \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})) \to \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_\perp))$ on sets of sets of traces:

$$\alpha_{\rightsquigarrow}(S) \stackrel{\text{def}}{=} \{\{\langle \sigma[0], \sigma[\omega]\rangle \in \Sigma \times \Sigma_\perp \mid \sigma \in T\} \mid T \in S\} \tag{10}$$

where $\Sigma_\perp \stackrel{\text{def}}{=} \Sigma \cup \{\perp\}$. The *dependency abstraction* $\alpha_{\rightsquigarrow}$ ignores all intermediate states between the initial state $\sigma[0]$ and the outcome $\sigma[\omega]$ of all traces σ in

all partitions T of S. Observe that a trace σ that consists of a single state s is abstracted as a pair $\langle s, s\rangle$. The corresponding dependency concretization function $\gamma_{\rightsquigarrow} \colon \mathcal{P}\left(\mathcal{P}\left(\Sigma \times \Sigma_\bot\right)\right) \to \mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right)$ over-approximates the original sets of traces by inserting arbitrary intermediate states:

$$\gamma_{\rightsquigarrow}(S) \stackrel{\text{def}}{=} \left\{T \in \mathcal{P}\left(\Sigma^{+\infty}\right) \mid \left\{\langle \sigma[0], \sigma[\omega]\rangle \in \Sigma \times \Sigma_\bot \mid \sigma \in T\right\} \in S\right\} \tag{11}$$

Example 5. Let us consider again the program of Example 1 and its outcome semantics $\alpha_\bullet([\![P]\!]_{\texttt{math}})$ shown in Example 3. Its dependency abstraction is:

$$\alpha_{\rightsquigarrow}(\alpha_\bullet([\![P]\!]_{\texttt{math}})) = \left\{\emptyset, \left\{\langle \texttt{F}\_, \texttt{FF}\rangle\right\}, \left\{\langle \texttt{T}\_, \texttt{TT}\rangle, \langle \texttt{F}\_, \texttt{FT}\rangle\right\}\right\}$$

which explicitly ignores intermediate program states. ■

Using $\alpha_{\rightsquigarrow}$, we now define the *dependency semantics* $\Lambda_{\rightsquigarrow} \in \mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right)$ as an abstraction of the outcome semantics $\Lambda_\bullet$.

Definition 2. *The* dependency semantics $\Lambda_{\rightsquigarrow} \in \mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right)$ *is defined as:*

$$\Lambda_{\rightsquigarrow} \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(\Lambda_\bullet) \tag{12}$$

where $\Lambda_\bullet \in \mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right)$ *is the outcome semantics (cf. Eq. 8) and* $\alpha_{\rightsquigarrow}$ *is the dependency abstraction (cf. Eq. 10).*

Neither the Kleenian fixpoint transfer nor the Tarskian fixpoint transfer can be used to obtain a fixpoint definition for the dependency semantics, but we have to proceed by union of disjoint fixpoints [12]. To this end, we observe that the outcome semantics $\Lambda_\bullet$ can be equivalently expressed as follows:

$$\begin{aligned} \Lambda_\bullet &= \Lambda_\bullet^+ \cup \Lambda_\bullet^\omega = \mathrm{lfp}^{\sqsubseteq}_{\emptyset}\ \Theta_\bullet^+ \cup \mathrm{lfp}^{\sqsubseteq}_{\{\Sigma^\omega\}}\ \Theta_\bullet^\omega \\ \Theta_\bullet^+(S) &\stackrel{\text{def}}{=} \{\Omega_{o_1=v_1,\ldots,o_k=v_k} \mid v_1, \ldots, v_k \in \mathrm{V}\} \uplus \{\tau\ ;\ T \mid T \in S\} \\ \Theta_\bullet^\omega(S) &\stackrel{\text{def}}{=} \{\tau\ ;\ T \mid T \in S\} \end{aligned} \tag{13}$$

where $\Lambda_\bullet^+$ and $\Lambda_\bullet^\omega$ separately compute the set of all sets of finite traces that agree on their outcome, and the set of all infinite traces, respectively.

In the following, given a set of traces $T \in \mathcal{P}\left(\Sigma^{+\infty}\right)$ and its dependency abstraction $\alpha_{\rightsquigarrow}(T)$, we abuse notation and write T^+ (resp. T^ω) to also denote $\alpha_{\rightsquigarrow}(T)^+ \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(T) \cap (\Sigma \times \Sigma)$ (resp. $\alpha_{\rightsquigarrow}(T)^\omega \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(T) \cap (\Sigma \times \{\bot\})$). Similarly, we reuse the symbols for the computational order $\sqsubseteq$, least upper bound $\sqcup$, and greatest lower bound $\sqcap$, instead of their abstractions. We can now use the Kleenian and Tarskian fixpoint transfer to separately derive fixpoint definitions of $\alpha_{\rightsquigarrow}(\Lambda_\bullet^+)$ and $\alpha_{\rightsquigarrow}(\Lambda_\bullet^\omega)$ in $\langle \mathcal{P}\left(\mathcal{P}\left(\Sigma \times \Sigma_\bot\right)\right), \sqsubseteq, \sqcup, \sqcap, \{\Sigma \times \{\bot\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$.

Lemma 2. *The abstraction* $\Lambda_{\rightsquigarrow}^+ \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(\Lambda_\bullet^+) \in \mathcal{P}\left(\mathcal{P}\left(\Sigma \times \Sigma\right)\right)$ *can be expressed as a least fixpoint in* $\langle \mathcal{P}\left(\mathcal{P}\left(\Sigma \times \Sigma_\bot\right)\right), \sqsubseteq, \sqcup, \sqcap, \{\Sigma \times \{\bot\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$ *as:*

$$\begin{aligned} \Lambda_{\rightsquigarrow}^+ &= \mathrm{lfp}^{\sqsubseteq}_{\{\emptyset\}}\ \Theta_{\rightsquigarrow}^+ \\ \Theta_{\rightsquigarrow}^+(S) &\stackrel{\text{def}}{=} \{\Omega_{o_1=v_1,\ldots,o_k=v_k} \times \Omega_{o_1=v_1,\ldots,o_k=v_k} \mid v_1, \ldots, v_k \in \mathrm{V}\} \uplus \{\tau \circ R \mid R \in S\} \end{aligned} \tag{14}$$

Proof (Sketch). By Kleenian fixpoint transfer (cf. Theorem 17 in [12]). □

Lemma 3. *The abstraction* $\Lambda^{\omega}_{\rightsquigarrow} \stackrel{\text{def}}{=} \alpha_{\rightsquigarrow}(\Lambda^{\omega}_{\bullet}) \in \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$ *can be expressed as a least fixpoint in* $\langle \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_{\perp})), \sqsubseteq, \sqcup, \sqcap, \{\Sigma \times \{\perp\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$ *as:*

$$\begin{aligned} \Lambda^{\omega}_{\rightsquigarrow} &= \mathrm{lfp}^{\dot{\sqsubseteq}}_{\{\Sigma\times\{\perp\}\}} \Theta^{\omega}_{\rightsquigarrow} \\ \Theta^{\omega}_{\rightsquigarrow}(S) &\stackrel{\text{def}}{=} \{\tau \circ R \mid R \in S\} \end{aligned} \tag{15}$$

Proof (Sketch). By Tarskian fixpoint transfer (cf. Theorem 18 in [12]). □

The fixpoint iteration for $\Lambda^{+}_{\rightsquigarrow}$ starts from the set containing only the empty relation. At the first iteration, the empty relation is replaced by all relations between pairs of final states that agree on the values of the output variables. At each next iteration, all relations are combined with the transition relation to obtain relations between initial and final states of increasingly longer traces. At the limit, we obtain the set of all relations between the initial and the final states of a program that agree on the final value of the output variables. The fixpoint iteration for $\Lambda^{\omega}_{\rightsquigarrow}$ starts from the set containing (the set of) all pairs of states and the $\perp$ outcome, and each iteration discards more and more pairs with initial states that do not belong to infinite traces of the program.

Now we can use Lemmas 2 and 3 to express the dependency semantics $\Lambda_{\rightsquigarrow}$ in a constructive fixpoint form (as the union of $\Lambda^{+}_{\rightsquigarrow}$ and $\Lambda^{\omega}_{\rightsquigarrow}$).

Theorem 3. *The dependency semantics* $\Lambda_{\rightsquigarrow} \in \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_{\perp}))$ *can be expressed as a least fixpoint in* $\langle \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_{\perp})), \sqsubseteq, \sqcup, \sqcap, \{\Sigma \times \{\perp\}, \emptyset\}, \{\emptyset, \Sigma \times \Sigma\}\rangle$ *as:*

$$\begin{aligned} \Lambda_{\rightsquigarrow} &= \Lambda^{+}_{\rightsquigarrow} \cup \Lambda^{\omega}_{\rightsquigarrow} = \mathrm{lfp}^{\dot{\sqsubseteq}}_{\{\Sigma\times\{\perp\},\emptyset\}} \Theta_{\rightsquigarrow} \\ \Theta_{\rightsquigarrow}(S) &\stackrel{\text{def}}{=} \{\Omega_{o_1=v_1,\ldots,o_k=v_k} \times \Omega_{o_1=v_1,\ldots,o_k=v_k} \mid v_1, \ldots, v_k \in \mathrm{V}\} \uplus \{\tau \circ R \mid R \in S\} \end{aligned} \tag{16}$$

Proof (Sketch). The proof follows immediately from Lemmas 2 and 3. □

Finally, we show that the dependency semantics $\Lambda_{\rightsquigarrow}$ is sound and complete for proving that a program does not use (a subset of) its input variables.

Theorem 4. *A program does not use a subset* J *of its input variables if and only if the image via* $\gamma_{\rightsquigarrow}$ *of its dependency semantics* $\Lambda_{\rightsquigarrow}$ *is a subset of* $\mathcal{N}_J$*:*

$$P \models \mathcal{N}_J \Leftrightarrow \gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}_J$$

Proof (Sketch). The proof follows from the definition of $\Lambda_{\rightsquigarrow}$ (cf. Eq. 12) and $\gamma_{\rightsquigarrow}$ (cf. Eq. 11), and from Theorem 2. □

Example 6. Let us consider again the program P and its outcome semantics $\alpha_{\bullet}([\![P]\!]_{\texttt{science}})$ from Example 4. The corresponding dependency semantics is:

$$\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})) = \{\emptyset, \{\langle \mathrm{T}\_, \mathrm{TF}\rangle, \langle \mathrm{F}\_, \mathrm{FF}\rangle\}, \{\langle \mathrm{T}\_, \mathrm{TT}\rangle, \langle \mathrm{F}\_, \mathrm{FT}\rangle\}\}$$

and, by definition of $\gamma_{\rightsquigarrow}$, we have that its concretization $\gamma_{\rightsquigarrow}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})))$ is an over-approximation of $\alpha_{\bullet}([\![P]\!]_{\texttt{science}})$. In particular, since intermediate state computations are irrelevant for deciding the input data usage property, all sets of traces in $\gamma_{\rightsquigarrow}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})))$ are over-approximations of exactly one set in $\alpha_{\bullet}([\![P]\!]_{\texttt{science}})$ with the same set of initial states and outcome. Thus, in this case, we can observe that all sets of traces in $\gamma_{\rightsquigarrow}(\alpha_{\rightsquigarrow}(\alpha_{\bullet}([\![P]\!]_{\texttt{science}})))$ belong to $\mathcal{N}_{\{\texttt{science}\}}$ and correctly conclude that P does not use the variable `science`. ■

At this point we have a sound and complete program semantics that captures only the minimal information needed to decide which input variables are unused by a program. In the rest of the paper, we present various static analyses for input data usage by means of sound abstractions of this semantics, which *under-approximate* (resp. over-approximate) the subset of the input variables that are *unused* (resp. used) by a program.

7 Input Data Usage Abstractions

We introduce a simple sequential programming language with boolean variables, which we use for illustration throughout the rest of the paper:

$$e ::= v \mid x \mid \texttt{not}\ e \mid e\ \texttt{and}\ e \mid e\ \texttt{or}\ e \qquad \text{(expressions)}$$
$$s ::= \texttt{skip} \mid x = e \mid \texttt{if}\ e\texttt{:}\ s\ \texttt{else:}\ s \mid \texttt{while}\ e\texttt{:}\ s \mid s\ s \qquad \text{(statements)}$$

where v ranges over boolean values, and x ranges over program variables. The `skip` statement, which does nothing, is a placeholder useful, for instance, for writing a conditional `if` statement without an `else` branch: `if` e`:` s `else: skip`. In the following, we often simply write `if` e`:` s instead of `if` e`:` s `else: skip`. Note that our work is not limited by the choice of a particular programming language, as the formal treatment in previous sections is language independent.

In Sects. 8 and 9, we show that existing static analyses based on dependencies [6,20] are abstractions of the dependency semantics $\Lambda_{\rightsquigarrow}$. We define each abstraction $\Lambda^{\sharp}$ over a partially ordered set $\langle \mathcal{A}, \sqsubseteq_A \rangle$ called *abstract domain*. More specifically, for each program statement s, we define a *transfer function* $\Theta^{\sharp}[\![s]\!] \colon \mathcal{A} \to \mathcal{A}$, and the abstraction $\Lambda^{\sharp}$ is the composition of the transfer functions of all statements in a program. We derive a more precise static analysis similar to dependency analyses used for program slicing [37] in Sect. 10. Finally, Sect. 11 demonstrates the value of expressing such analyses as abstract domains by combining them with an existing abstraction of compound data structures such as arrays and lists [16] to detect unused chunks of input data.

8 Secure Information Flow Abstractions

Secure information flow analysis [18] aims at proving that a program will not leak sensitive information. Most analyses focus on proving *non-interference* [35] by classifying program variables into different security levels [17], and ensuring the

absence of information flow from variables with higher security level to variables with lower security level. The most basic classification comprises a low security level L, and a high security level H: program variables classified as L are public information, while variables classified as H are private information.

In our context, if we classify input variables as H and all other variables as L, possiblistic non-interference [21] coincides with the input data usage property $\mathcal{N}$ (cf. Eq. 4) *restricted to consider only terminating programs*. However, in general, (possibilistic) non-interference is too strong for our purposes as it requires that *none* of the input variables is used by a program. We illustrate this using as an example a non-interference analysis recently proposed by Assaf et al. [6] that is conveniently formalized in the framework of abstract interpretation. We briefly present here a version of the originally proposed analysis, simplified to consider only the security levels L and H, and we point out the significance of the definitions for input data usage.

Let $\mathcal{L} \stackrel{\text{def}}{=} \{L, H\}$ be the set of security levels, and let the set X of all program variables be partitioned into a set X_L of variables classified as L and a set X_H of variables classified as H (i.e., the input variables). A dependency constraint $L \rightsquigarrow x$ expresses that the current value of the variable x depends only on the initial values of variables having at most security level L (i.e., it does not depend on the initial value of any of the input variables). The non-interference analysis Λ_{F} proposed by Assaf et al. is a *forward analysis* in the lattice $\langle \mathcal{P}(\mathrm{F}), \sqsubseteq_{\mathrm{F}}, \sqcup_{\mathrm{F}} \rangle$ where $\mathrm{F} \stackrel{\text{def}}{=} \{L \rightsquigarrow x \mid x \in \mathrm{X}\}$ is the set of all dependency constraints, $S_1 \sqsubseteq_{\mathrm{F}} S_2 \stackrel{\text{def}}{=} S_1 \supseteq S_2$, and $S_1 \sqcup_{\mathrm{F}} S_2 \stackrel{\text{def}}{=} S_1 \cap S_2$. The transfer function $\Theta_{\mathrm{F}}[\![s]\!] : \mathcal{P}(\mathrm{F}) \to \mathcal{P}(\mathrm{F})$ for each statement s in our simple programming language is defined as follows:

$$\Theta_{\mathrm{F}}[\![\texttt{skip}]\!](S) \stackrel{\text{def}}{=} S$$

$$\Theta_{\mathrm{F}}[\![x = e]\!](S) \stackrel{\text{def}}{=} \{L \rightsquigarrow y \in S \mid y \neq x\} \cup \{L \rightsquigarrow x \mid \mathcal{V}_{\mathrm{F}}[\![e]\!]S\}$$

$$\Theta_{\mathrm{F}}[\![\texttt{if } e\texttt{: } s_1 \texttt{ else: } s_2]\!](S) \stackrel{\text{def}}{=} \begin{cases} \Theta_{\mathrm{F}}[\![s_1]\!](S) \sqcup_{\mathrm{F}} \Theta_{\mathrm{F}}[\![s_2]\!](S) & \text{if } \mathcal{V}_{\mathrm{F}}[\![e]\!]S \\ \{L \rightsquigarrow x \in S \mid x \notin \mathrm{W}(s_1) \cup \mathrm{W}(s_2)\} & \text{otherwise} \end{cases}$$

$$\Theta_{\mathrm{F}}[\![\texttt{while } e\texttt{: } s]\!](S) \stackrel{\text{def}}{=} \mathrm{lfp}_S^{\sqsubseteq_{\mathrm{F}}}\ \Theta_{\mathrm{F}}[\![\texttt{if } e\texttt{: } s \texttt{ else: skip}]\!]$$

$$\Theta_{\mathrm{F}}[\![s_1\ s_2]\!](S) \stackrel{\text{def}}{=} \Theta_{\mathrm{F}}[\![s_2]\!] \circ \Theta_{\mathrm{F}}[\![s_1]\!](S)$$

where $\mathrm{W}(s)$ denotes the set of variables modified by the statement s, and $\mathcal{V}_{\mathrm{F}}[\![e]\!]S$ determines whether a set of dependencies S guarantees that the expression e has a unique value independently of the initial value of the input variables. For a variable x, $\mathcal{V}_{\mathrm{F}}[\![x]\!]S$ is true if and only if $L \rightsquigarrow x \in S$. Otherwise, $\mathcal{V}_{\mathrm{F}}[\![e]\!]S$ is defined recursively on the structure of e, and it is always true for a boolean value v [6]. An assignment $x = e$ discards all dependency constraints related to the assigned variable x, and adds constraints $L \rightsquigarrow x$ if e has a unique value independently of the initial values of the input variables. This captures an *explicit flow* of information between e and x. A conditional statement $\texttt{if } e\texttt{: } s_1 \texttt{ else: } s_2$ joins the dependency constraints obtained from s_1 and s_2, if e does not depend

on the initial values of the input variables (i.e., $\mathcal{V}_{\mathrm{F}}[\![e]\!]S$ is true). Otherwise, it discards all dependency constraints related to the variables modified in either of its branches. This captures an *implicit flow* of information from e. The initial set of dependencies contains a constraint $L \rightsquigarrow x$ for each variable x that is not an input variable. We exemplify the analysis below.

Example 7. Let us consider again the program P from Example 1 (stripped of the `input` and `print` statements, which are not present in our simple language):

```
1 passing = True
2 if not english: english = False              # english should be passing
3 if not math: passing = bonus
4 if not math: passing = bonus                 # math should be science
```

The analysis begins from the set of dependency constraints $\{L \rightsquigarrow \texttt{passing}\}$, which classifies input variables as H and all other variables as L. The assignment at line 1 leaves the set unchanged as the value of the expression `True` on the right-hand side of the assignment does not depend on the initial value of the input variables. The set remains unchanged by the conditional statement at line 2, even though the boolean condition depends on the input variable `english`, because the variable `passing` is not modified. Finally, at line 3 and 4, the analysis captures an explicit flow of information from the input variable `bonus` and an implicit flow of information from the input variable `math`. Thus, the set of dependency constraints becomes empty at line 3, and remains empty at line 4.

Observe that, in this case, non-interference does not hold since the result of the program depends on some of the input variables. Therefore, the analysis is only able to conclude that at least one of the input variables may be used by the program, but it cannot determine which input variables are unused. ■

The example shows that non-interference is too strong a property in general. Of course, one could determine which input variables are unused by running multiple instances of the non-interference analysis Λ_{F}, each one of them classifying a single different input variable as H and all other variables as L. However, this becomes cumbersome in a data science application where a program reads and manipulates a large amount of input data.

Moreover, we emphasize that our input data usage property is more general than (possibilistic) non-interference since it also considers non-termination. We are not aware of any work on termination-sensitive possibilistic non-interference.

Example 8. Let us modify the program P shown in Example 7 as follows:

```
1 passing = True
2 while not english: english = False
```

In this case, since the loop at line 2 does not modify the output variable `passing`, the non-interference analysis Λ_{F} will leave the initial set of dependency constraints $\{L \rightsquigarrow \texttt{passing}\}$ unchanged, meaning that the result of the program does not depend on any of its input variables. However, the input variable `english` is used since its value influences the outcome of the program: the program terminates if `english` is true, and does not terminate otherwise. ■

The example demonstrates that the analysis is *unsound* for a non-terminating program.[2] We show that the non-interference analysis Λ_{F} is sound for proving that a program does not use any of its input variables, *only if the program is terminating*. We define the concretization function $\gamma_{\mathrm{F}} \colon \mathcal{P}(\mathrm{F}) \to \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$:

$$\gamma_{\mathrm{F}}(S) \stackrel{\text{def}}{=} \{R \in \mathcal{P}(\Sigma \times \Sigma) \mid \alpha_{\mathrm{F}}(R) \sqsubseteq_{\mathrm{F}} S\} \tag{17}$$

The abstraction function $\alpha_{\mathrm{F}} \colon \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma)) \to \mathcal{P}(\mathrm{F})$ maps each relation R between states of a program to the corresponding set of dependency constraints: $\alpha_{\mathrm{F}}(R) \stackrel{\text{def}}{=} \{L \rightsquigarrow x \mid x \in \mathrm{X}_L \wedge \forall i \in \mathrm{X}_H \colon \textsc{unused}_{i,x}(R)\}$, where $\textsc{unused}_{i,x}$ is the relational abstraction of $\textsc{unused}_i$ (cf. Eq. 3) in which we compare only the result stored in the variable x (i.e., we compare $\sigma[\omega](o)$ and $\sigma'[\omega](o)$, instead of $\sigma[\omega]$ and $\sigma'[\omega]$ as in Eq. 3).

Theorem 5. *A terminating program does not use any of its input variables if the image via $\gamma_{\rightsquigarrow} \circ \gamma_{\mathrm{F}}$ of its non-interference abstraction Λ_{F} is a subset of $\mathcal{N}$:*

$$\gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})) \subseteq \mathcal{N} \Rightarrow P \models \mathcal{N}$$

Proof. Let us assume that $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})) \subseteq \mathcal{N}$. By definition of γ_{F} (cf. Eq. 17), since the program is terminating, we have that $\Lambda_{\rightsquigarrow} \subseteq \gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})$ and, by monotonicity of the concretization function $\gamma_{\rightsquigarrow}$ (cf. Eq. 11), we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}}))$. Thus, since $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})) \subseteq \mathcal{N}$, we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}$. The conclusion follows from Theorem 4. □

Note that the termination of the program is necessary for the proof of Theorem 5. Indeed, for a non-terminating program, we have that $\Lambda_{\rightsquigarrow} \not\subseteq \gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})$ (since $\Lambda_{\rightsquigarrow}$ includes relational abstractions of infinite traces that are missing from $\gamma_{\mathrm{F}}(\Lambda_{\mathrm{F}})$) and thus we cannot conclude the proof.

This result shows that the non-interference analysis Λ_{F} is an abstraction of the dependency semantics $\Lambda_{\rightsquigarrow}$ presented earlier. However, we remark that the same result applies to all other instances in this important class of analysis [5,25, etc.], which are therefore subsumed by our framework.

9 Strongly Live Variable Abstraction

Strongly live variable analysis [20] is a variant of the classic live variable analysis [32] performed by compilers to determine, for each program point, which variables may be potentially used before they are assigned to. A variable is *strongly live* if it is used in an assignment to another strongly live variable, or if is used in a statement other than an assignment. Otherwise, a variable is considered *faint*.

[2] The case of a program using an input variable and then always diverging is not problematic because the analysis would be imprecise but still sound.

Strongly live variable analysis Λ_{X} is a *backward analysis* in the complete lattice $\langle \mathcal{P}(\mathrm{X}), \subseteq, \cup, \cap, \emptyset, \mathrm{X} \rangle$, where X is the set of all program variables. The transfer function $\Theta_{\mathrm{X}}[\![s]\!] \colon \mathcal{P}(\mathrm{X}) \to \mathcal{P}(\mathrm{X})$ for each statement s is defined as:

$$\Theta_{\mathrm{X}}[\![\texttt{skip}]\!](S) \stackrel{\text{def}}{=} S$$

$$\Theta_{\mathrm{X}}[\![x = e]\!](S) \stackrel{\text{def}}{=} \begin{cases} (S \setminus \{x\}) \cup \textsc{vars}(e) & x \in S \\ S & \text{otherwise} \end{cases}$$

$$\Theta_{\mathrm{X}}[\![\texttt{if } b\texttt{: } s_1 \texttt{ else: } s_2]\!](S) \stackrel{\text{def}}{=} \textsc{vars}(b) \cup \Theta_{\mathrm{X}}[\![s_1]\!](S) \cup \Theta_{\mathrm{X}}[\![s_2]\!](S)$$

$$\Theta_{\mathrm{X}}[\![\texttt{while } b\texttt{: } s]\!](S) \stackrel{\text{def}}{=} \textsc{vars}(b) \cup \Theta_{\mathrm{X}}[\![s]\!](S)$$

$$\Theta_{\mathrm{X}}[\![s_1\ s_2]\!](S) \stackrel{\text{def}}{=} \Theta_{\mathrm{X}}[\![s_1]\!] \circ \Theta_{\mathrm{X}}[\![s_2]\!](S)$$

where $\textsc{vars}(e)$ is the set of variables in the expression e. For input data usage, the initial set of strongly live variables contains the output variables of the program.

Example 9. Let us consider again the program P shown in Example 7. The strongly live variable analysis begins from the set {`passing`} containing the output variable `passing`. At line 3, the set of strongly live variables is {`math`, `bonus`} since `bonus` is used in an assignment to the strongly live variable `passing`, and `math` is used in the condition of the `if` statement. Finally, at line 1, the set of strongly live variables is {`english`, `math`, `bonus`} because `english` is used in the condition of the `if` statement at line 2. Thus, strongly live variable analysis is able to conclude that the input variable `science` is unused. However, it is not precise enough to determine that the variable `english` is also unused. ■

The imprecision of the analysis derives from the fact that it does not capture implicit flows of information precisely (cf. Sect. 8) but only over-approximates their presence. Thus, the analysis is unable to detect when a conditional statement, for instance, modifies only variables that have no impact on the outcome of a program; a situation likely to arise due to a programming error, as shown in the previous example. However, in virtue of this imprecise treatment of implicit flows, we can show that strongly live variable analysis is sound for input data usage, even for non-terminating programs.

We define the concretization function $\gamma_{\mathrm{X}} \colon \mathcal{P}(X) \to \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma_\perp))$ as:

$$\gamma_{\mathrm{X}}(S) \stackrel{\text{def}}{=} \{R \in \Sigma \times \Sigma_\perp \mid \forall i \in \mathrm{X} \setminus S \colon \textsc{unused}_i(R)\} \tag{18}$$

where we abuse notation and use $\textsc{unused}_i$ (cf. Eq. 3) to also denote its dependency abstraction (cf. Eq. 10). We now show that strongly live variable analysis is sound for proving that a program does not use the faint variables.

Theorem 6. *A program does not use a subset J of its input variables if the image via $\gamma_{\rightsquigarrow} \circ \gamma_{\mathrm{X}}$ of its strongly live variable abstraction Λ_{X} is a subset of $\mathcal{N}_J$:*

$$\gamma_{\rightsquigarrow}(\gamma_{\mathrm{X}}(\Lambda_{\mathrm{X}})) \subseteq \mathcal{N}_J \Rightarrow P \models \mathcal{N}_J$$

Proof. Let us assume that $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{X}}(\Lambda_{\mathrm{X}})) \subseteq \mathcal{N}_J$. By definition of γ_{X} (cf. Eq. 18), we have that $\Lambda_{\rightsquigarrow} \subseteq \gamma_{\mathrm{X}}(\Lambda_{\mathrm{X}})$ and, by monotonicity of $\gamma_{\rightsquigarrow}$ (cf. Eq. 11), we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \gamma_{\rightsquigarrow}(\gamma_{\mathrm{X}}(\Lambda_{\mathrm{X}}))$. Thus, since $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{X}}(\Lambda_{\mathrm{X}})) \subseteq \mathcal{N}_J$, we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}_J$. The conclusion follows from Theorem 4. □

This result shows that also strongly live variable analysis is subsumed by our framework as it is an abstraction of the dependency semantics $\Lambda_{\rightsquigarrow}$.

10 Syntactic Dependency Abstractions

In the following, we derive a more precise data usage analysis based on *syntactic* dependencies between program variables. For simplicity, the analysis does not take program termination into account, but we discuss possible solutions at the end of the section. Due to space limitations, we only provide a terse description of the abstraction and refer to [36] for further details.

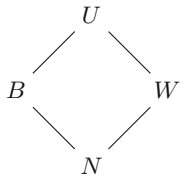


Fig. 5. Hasse diagram for the complete lattice $\langle \textsc{usage}, \sqsubseteq_{\textsc{usage}}, \sqcup_{\textsc{usage}}, \sqcap_{\textsc{usage}}, N, U \rangle$.

In order to capture implicit dependencies from variables appearing in boolean conditions of conditional and while statements, we track when the value of a variable is used or modified in a statement based on the level of nesting of the statement in other statements. More formally, each program variable maps to a value in the complete lattice shown in Fig. 5: the values U (*used*) and N (*not-used*) respectively denote that a variable may be used and is not used at the current nesting level; the values B (*below*) and W (*overwritten*) denote that a variable may be used at a lower nesting level, and the value W additionally indicates that the variable is modified at the current nesting level.

A variable is used (i.e., maps to U) if it is used in an assignment to another variable that is used in the current or a lower nesting level (i.e., a variable that maps to U or B). We define the operator $\textsc{assign}[\![x = e]\!]$ to compute the effect of an assignment on a map $m \colon \mathrm{X} \to \textsc{usage}$, where X is the set of all variables:

$$\textsc{assign}[\![x = e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} W & y = x \wedge y \notin \textsc{vars}(e) \wedge m(x) \in \{U, B\} \\ U & y \in \textsc{vars}(e) \wedge m(x) \in \{U, B\} \\ m(y) & \text{otherwise} \end{cases} \tag{19}$$

The assigned variable is overwritten (i.e., maps to W), unless it is used in e.

Another reason for a variable to be used is if it appears in the boolean condition e of a statement that uses another variable or modifies another used variable (i.e., there exists a variable x that maps to U or W):

$$\textsc{filter}[\![e]\!](m) \stackrel{\text{def}}{=} \lambda y. \begin{cases} U & y \in \textsc{vars}(e) \wedge \exists x \in \mathrm{X}\colon m(x) \in \{U, W\} \\ m(y) & \text{otherwise} \end{cases} \tag{20}$$

We maintain a *stack* of these maps that grows or shrinks based on the level of nesting of the currently analyzed statement. More formally, a stack is a tuple $\langle m_0, m_1, \ldots, m_k \rangle$ of mutable length k, where each element $m_0, m_1, \ldots, m_k$ is a map from X to USAGE. In the following, we use Q to denote the set of all stacks, and we abuse notation by writing $\textsc{assign}[\![x = e]\!]$ and $\textsc{filter}[\![e]\!]$ to also denote the corresponding operators on stacks:

$$\textsc{assign}[\![x = e]\!](\langle m_0, m_1, \ldots, m_k \rangle) \stackrel{\text{def}}{=} \langle \textsc{assign}[\![x = e]\!](m_0), m_1, \ldots, m_k \rangle$$
$$\textsc{filter}[\![e]\!](\langle m_0, m_1, \ldots, m_k \rangle) \stackrel{\text{def}}{=} \langle \textsc{filter}[\![e]\!](m_0), m_1, \ldots, m_k \rangle$$

The operator PUSH duplicates the map at the top of the stack and modifies the copy using the operator INC, to account for an increased nesting level:

$$\begin{aligned} \textsc{push}(\langle m_0, m_1, \ldots, m_k \rangle) &\stackrel{\text{def}}{=} \langle \textsc{inc}(m_0), m_0, m_1, \ldots, m_k \rangle \\ \textsc{inc}(m) &\stackrel{\text{def}}{=} \lambda y. \begin{cases} B & m(y) \in \{U\} \\ N & m(y) \in \{W\} \\ m(y) & \text{otherwise} \end{cases} \end{aligned} \tag{21}$$

A used variable (i.e., mapping to U) becomes used below (i.e., now maps to B), and a modified variable (i.e., mapping to W) becomes unused (i.e., now maps to N). The dual operator POP combines the two maps at the top of the stack:

$$\begin{aligned} \textsc{pop}(\langle m_0, m_1, \ldots, m_k \rangle) &\stackrel{\text{def}}{=} \langle \textsc{dec}(m_0, m_1), \ldots, m_k \rangle \\ \textsc{dec}(m, k) &\stackrel{\text{def}}{=} \lambda y. \begin{cases} k(y) & m(y) \in \{B, N\} \\ m(y) & \text{otherwise} \end{cases} \end{aligned} \tag{22}$$

where the DEC operator restores the value a variable y mapped to before increasing the nesting level (i.e., $k(y)$) if it has not changed since (i.e., if the variable still maps to B or N), and otherwise retains the new value y maps to.

We can now define the data usage analysis Λ_{Q}, which is a *backward analysis* on the lattice $\langle \mathrm{Q}, \sqsubseteq_{\mathrm{Q}}, \sqcup_{\mathrm{Q}} \rangle$. The partial order $\sqsubseteq_{\mathrm{Q}}$ and the least upper bound $\sqcup_{\mathrm{Q}}$ are the pointwise lifting, for each element of the stack, of the partial order and least upper bound between maps from X to USAGE (which in turn are the pointwise lifting of the partial order $\sqsubseteq_{\textsc{usage}}$ and least upper bound $\sqcup_{\textsc{usage}}$ of the USAGE lattice, cf. Fig. 5). We define the transfer function $\Theta_{\mathrm{Q}}[\![s]\!]\colon \mathrm{Q} \to \mathrm{Q}$ for each statement s in our simple programming language as follows:

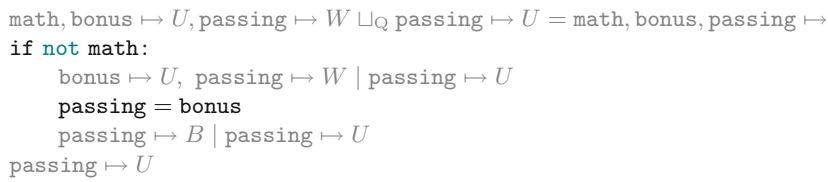


Fig. 6. Data usage analysis of the last statement of the program shown in Example 7. Stack elements are separated by | and, for brevity, variables mapping to N are omitted.

$$
\begin{aligned}
\Theta_{\mathrm{Q}}[\![\texttt{skip}]\!](q) &\stackrel{\text{def}}{=} q \\
\Theta_{\mathrm{Q}}[\![x = e]\!](q) &\stackrel{\text{def}}{=} \textsc{assign}[\![x = e]\!](q) \\
\Theta_{\mathrm{Q}}[\![\texttt{if } b\texttt{: } s_1 \texttt{ else: } s_2]\!](q) &\stackrel{\text{def}}{=} \textsc{pop} \circ \textsc{filter}[\![b]\!] \circ \Theta_{\mathrm{Q}}[\![s_1]\!] \circ \textsc{push}(q) \\
&\quad \sqcup_{\mathrm{Q}} \textsc{pop} \circ \textsc{filter}[\![b]\!] \circ \Theta_{\mathrm{Q}}[\![s_2]\!] \circ \textsc{push}(q) \\
\Theta_{\mathrm{Q}}[\![\texttt{while } b\texttt{: } s]\!](q) &\stackrel{\text{def}}{=} \mathrm{lfp}_t^{\sqsubseteq_{\mathrm{Q}}}\, \Theta_{\mathrm{Q}}[\![\texttt{if } b\texttt{: } s \texttt{ else: skip}]\!] \\
\Theta_{\mathrm{Q}}[\![s_1\ s_2]\!](q) &\stackrel{\text{def}}{=} \Theta_{\mathrm{Q}}[\![s_1]\!] \circ \Theta_{\mathrm{Q}}[\![s_2]\!](q)
\end{aligned}
$$

The initial stack contains a single map, in which the output variables map to the value U, and all other variables map to N. We exemplify the analysis below.

Example 10. Let us consider again the program P shown in Example 7. The initial stack begins with a single map m, in which the output variable `passing` maps to U and all other variables map to N.

At line 4, before analyzing the body of the conditional statement, a modified copy of m is pushed onto the stack: this copy maps `passing` to B, meaning that `passing` is only used in a lower nesting level, and all other variables still map to N (cf. Eq. 21). As a result of the assignment (cf. Eq. 19), `passing` is overwritten (i.e., maps to W), and bonus is used (i.e., maps to U). Since the body of the conditional statement modifies a used variable and uses another variable, the analysis of its boolean condition makes `math` used as well (cf. Eq. 20). Finally, the maps at the top of the stack are merged and the result maps `math`, `bonus`, and `passing` to U, and all other variables to N (cf. Eq. 22). The analysis is visualized in Fig. 6.

The stack remains unchanged at line 3 and line 2, since the statement at line 3 is identical to line 4 and the body of the conditional statement at line 2 does not modify any used variable and does not use any other variable. Finally, at line 1 the variable `passing` is modified (i.e., it now maps to W), while `math` and `bonus` remain used (i.e., they map to U). Thus, the analysis is precise enough to conclude that the input variables `english` and `science` are unused. ■

Note that, similarly to the non-interference analysis presented in Sect. 8, the data usage analysis Λ_{Q} does not consider non-termination. Indeed, for the program shown in Example 8, the analysis does not capture that the input variable

`english` is used, even though the termination of the program depends on its value. We define the concretization function $\gamma_{\mathrm{Q}} \colon \mathrm{Q} \to \mathcal{P}(\mathcal{P}(\Sigma \times \Sigma))$ as:

$$\gamma_{\mathrm{Q}}(\langle m_0, \ldots, m_k \rangle) \stackrel{\text{def}}{=} \{R \in \Sigma \times \Sigma \mid \forall i \in \mathrm{X} \colon m_0(i) \in \{N\} \Rightarrow \textsc{unused}_i(R)\} \tag{23}$$

where again we write $\textsc{unused}_i$ (cf. Eq. 3) to also denote its dependency abstraction. We now show that Λ_{Q} is sound for proving that a program does not use a subset of its input variables, *if the program is terminating.*

Theorem 7. *A terminating program does not use a subset J of its input variables if the image via $\gamma_{\rightsquigarrow} \circ \gamma_{\mathrm{Q}}$ of its abstraction Λ_{Q} is a subset of $\mathcal{N}_J$:*

$$\gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})) \subseteq \mathcal{N}_J \Rightarrow P \models \mathcal{N}_J$$

Proof. Let us assume that $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})) \subseteq \mathcal{N}_J$. Since the program is terminating, we have that $\Lambda_{\rightsquigarrow} \subseteq \gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})$, by definition of the concretization function γ_{Q} (cf. Eq. 23). Then, by monotonicity of $\gamma_{\rightsquigarrow}$ (cf. Eq. 11), we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}}))$. Thus, since $\gamma_{\rightsquigarrow}(\gamma_{\mathrm{Q}}(\Lambda_{\mathrm{Q}})) \subseteq \mathcal{N}_J$, we have that $\gamma_{\rightsquigarrow}(\Lambda_{\rightsquigarrow}) \subseteq \mathcal{N}_J$. The conclusion follows from Theorem 4. □

In order to take termination into account, one could map each variable appearing in the guard of a loop to the value U. Alternatively, one could run a termination analysis [3,33,34], along with the data usage analysis, and only map to U variables appearing in the loop guard of a possibly non-terminating loop.

11 Piecewise Abstractions

The static analyses presented so far can be used only to detect unused data stored in program variables. However, realistic data science applications read and manipulate data organized in data structures such as arrays, lists, and dictionaries. In the following, we demonstrate that having expressed the analyses as abstract domains allows us to easily lift the analyses to such a scenario. In particular, to detect unused chunks of the input data, we combine the more precise data usage analysis presented in the previous section with the array content abstraction proposed by Cousot et al. [16]. Due to space limitations, we provide only an informal description of the resulting abstract domain and refer to [36] for further details and examples. The analyses presented in earlier sections can be similarly combined with the array abstraction for the same purpose.

We extend our small programming language introduced in Sect. 7 with integer variables, arithmetic and boolean comparison expressions, and arrays:

$$\begin{aligned} e &::= \cdots \mid a[e] \mid \texttt{len}(a) \mid e \ \oplus \ e \mid e \ \bowtie \ e && \text{(expressions)} \\ s &::= \cdots \mid a[e] = e && \text{(statements)} \end{aligned}$$

where $\oplus$ and $\bowtie$ respectively range over arithmetic and boolean comparison operators, a ranges over array variables, and $\texttt{len}(a)$ denotes the length of a.

Piecewise Array Abstraction. The array abstraction [16] divides an array into consecutive segments, each segment being a uniform abstraction of the array content in that segment. The bounds of the segments are specified by sets of side-effect free expressions restricted to a canonical normal form, all having the same (concrete) value. The abstraction is parametric in the choice of the abstract domains used to manipulate sets of expressions and to represent the array content within each segment. For our analysis, we use the octagon abstract domain [31] for the expressions, and the USAGE lattice presented in the previous section (cf. Fig. 5) for the segments. Thus, an array a is abstracted, for instance, as $\{0, i\}\, N\, \{j+1\}?\ U\, \{\texttt{len}(a)\}$, where the symbol ? indicates that the segment $\{0, i\}\, N\, \{j+1\}$ might be empty. The abstraction indicates that all array elements (if any) from index i (which is equal to zero) to index j (the bound $j+1$ is exclusive) are unused, and all elements from $j+1$ to $\texttt{len}(a)-1$ may be used. Let A be the set of all such array abstractions. The initial segmentation of an array $a \in \mathrm{A}$ is a single segment with unused content (i.e., $\{0\}\, N\, \{\texttt{len}(a)\}?$).

For our analysis, we augment the array abstraction with new backward assignment and filter operators. The operators $\textsc{assign}_{\mathrm{A}}[\![a[i] = e]\!]$ and $\textsc{filter}_{\mathrm{A}}[\![e]\!]$ split and fill segments to take into account assignments and accesses to array elements that influence the program outcome. For instance, an assignment to $a[i]$ with an expression containing a used variable modifies the segmentation $\{0\}\, N\, \{\texttt{len}(a)\}?$ into $\{0\}\, N\, \{i\}?\ U\, \{i+1\}\, N\, \{\texttt{len}(a)\}?$, which indicates that the array element at index i is used by the program. An access $a[i]$ in a boolean condition guarding a statement that uses or modifies another used variables is handled analogously. Instead, the operator $\textsc{assign}_{\mathrm{A}}[\![x = e]\!]$ modifies the segmentation of an array by replacing each occurrence of the assigned variable x with the canonical normal form of the expression e. For instance, an assignment $i = i + 1$ modifies the segmentation $\{0\}\, N\, \{i\}?\ U\, \{i+1\}\, N\, \{\texttt{len}(a)\}?$ into $\{0\}\, N\, \{i+1\}?\ U\, \{i+2\}\, N\, \{\texttt{len}(a)\}?$. If e cannot be precisely put into a canonical normal form, the operator replaces the assigned variable with an approximation of e as an integer interval [13] computed using the underlying numerical domain, and possibly merges segments together as a result of the approximation. For instance, a non-linear assignment $i = i * j$ approximated as $i = [0, 1]$ modifies the segmentation $\{0\}\, N\, \{i\}?\ U\, \{i+1\}\, N\, \{\texttt{len}(a)\}?$ into $\{0\}\, U\, \{2\}\, N\, \{\texttt{len}(a)\}?$, which loses the information that the initial segment of the array is unused.

When merging control flows, segmentations are compared or joined by means of a *unification algorithm* [16], which finds the coarsest common refinement of both segmentations. Then, the comparison $\sqsubseteq_{\mathrm{A}}$ or the join $\sqcup_{\mathrm{A}}$ is performed pointwise for each segment using the corresponding operators of the underlying abstract domain chosen to abstract the array content. For our analysis, we adapt and refine the originally proposed unification algorithm to take into account the knowledge of the numerical domain chosen to abstract the segment bounds. We refer to [36] for further details. A widening $\triangledown_{\mathrm{A}}$ limits the number of segments to enforce termination of the analysis.

Piecewise Data Usage Analysis. We can now map each scalar variable to an element of the USAGE lattice and each array variable to an array segmentation

```
1 failed = 0
2 i = 1                                                    # 1 should be 0
3 while i < len(grades):
4     if grades[i] < 4: failed = failed + 1
5     i = i + 1
6 passing = 2 * failed < len(grades)
```

Fig. 7. Another program to check if a student has passed a number of exams based on their grades stored in the array `grades`. The programmer has made a mistake at line 2 that causes the program to ignore the grade stored at index 0 in `grades`.

```
grades ↦ {0} N {i}? U {i + 1}? U {len(grades)}?
while i < len(grades):
    grades ↦ {0} N {i}? U {i + 1}? B {i + 2}? B {len(grades)}? | ...
    if grades[i] < 4:
        grades ↦ {0} N {i + 1}? B {i + 2}? B {len(grades)}? | ··· | ...
        failed = failed + 1
        grades ↦ {0} N {i + 1}? B {i + 2}? B {len(grades)}? | ··· | ...
    grades ↦ {0} N {i + 1}? B {i + 2}? B {len(grades)}? | ...
    i = i + 1
    grades ↦ {0} N {i}? B {i + 1}? B {len(grades)}? | ...
grades ↦ {0} N {len(grades)}?
```

Fig. 8. Data usage analysis of the loop statement of the program shown in Example 11. Stack elements are separated by | and, for brevity, only array variables are shown.

in A, and use the data usage analysis Λ_{Q} presented in the previous section to identify unused input data stored in variables and portions of arrays.

Example 11. Let us consider the program shown in Fig. 7 where the array variable `grades` and the variable `passing` are the input and output variables, respectively. The initial stack contains a single map in which `passing` maps to U, all other scalar variables map to N, and `grades` maps to $\{0\}\, N\, \{\texttt{len(grades)}\}?$, indicating that all elements of the array (if any) are unused.

At line 6, the assignment modifies the variable `passing` (i.e., `passing` now maps to W) and uses the variable `failed` (i.e., `failed` now maps to U), while every other variable remains unchanged.

The result of the analysis of the loop statement at line 3 is shown in Fig. 8. The analysis of the loop begins by pushing (cf. Eq. 21) a map onto the stack in which `passing` becomes unused (i.e., maps to N) and `failed` is used only in a lower nesting level (i.e., maps to B), and every other variable still remains unchanged. At the first iteration of the analysis of the loop body, the assignment at line 4 uses `failed` and thus the access `grades[i]` at line 3 creates a used segment in the segmentation for `grades`, which becomes $\{0\}\, N\, \{\texttt{i}\}?\ U\, \{\texttt{i}+1\}\ \ N\, \{\texttt{len(grades)}\}?$. At the second iteration, the PUSH operator turns the used segment $\{\texttt{i}\}\, U\, \{\texttt{i}+1\}$ into $\{\texttt{i}\}\, B\, \{\texttt{i}+1\}$, and the assignment to `i` modifies the segment into $\{\texttt{i}+1\}\, B\, \{\texttt{i}+2\}$ (while the segmentation in the second stack element becomes

$\{0\}\, N\, \{\mathtt{i}+1\}?\; U\, \{\mathtt{i}+2\}\;\; N\, \{\mathtt{len(grades)}\}?$). Then, the access to the array at line 3 creates again a used segment $\{\mathtt{i}\}\, U\, \{\mathtt{i}+1\}$ (in the first segmentation) and the analysis continues with the result of the POP operator (cf. Eq. 22): $\{0\}\, N\, \{\mathtt{i}\}?\; U\, \{\mathtt{i}+1\}?\; U\, \{\mathtt{i}+2\}?\; N\, \{\mathtt{len(grades)}\}?$. After widening, the last two segments are merged into a single segment, and the analysis of the loop terminates with $\{0\}\, N\, \{\mathtt{i}\}?\; U\, \{\mathtt{i}+1\}?\; U\, \{\mathtt{len(grades)}\}?$.

Finally, the analysis of the assignment at line 2 produces the segmentation $\{0\}\, N\, \{1\}?\; U\, \{2\}?\; U\, \{\mathtt{len(grades)}\}?$, which correctly indicates that the first element of the array `grades` (if any) is unused by the program. ■

Implementation. The analyses presented in this and in the previous section are implemented in the prototype static analyzer LYRA and are available online[3].

The implementation is in PYTHON and, at the time of writing, accepts programs written in a limited subset of PYTHON without user-defined classes. A type inference is run before the analysis of a program. The analysis is performed backwards on the control flow graph of the program with a standard worklist algorithm [32], using widening at loop heads to enforce termination.

12 Related Work

The most directly relevant work has been discussed throughout the paper. The non-interference analysis proposed by Assaf et al. [6] (cf. Sect. 8) is similar to the logic of Amtoft and Banerjee [5] and the type system of Hunt and Sands [25]. The data usage analysis proposed in Sect. 10 is similar to dependency analyses used for program slicing [37] (e.g., [24]). Both analyses as well as strongly live variable analysis (cf. Sect. 9) are based on the *syntactic* presence of a variable in the definition of another variable. To overcome this limitation, one should look further for *semantic* dependencies between *values* of program variables. In this direction, Giacobazzi, Mastroeni, and others [19,22,29] have proposed the notion of *abstract dependency*. However, note that an analysis based on abstract dependencies would over-approximate the subset of the input variables that are unused by a program. Indeed, the absence of an abstract dependency between variables (e.g., a dependency between the parity of the variables [19,29]) does not imply the absence of a (concrete) dependency between the variables (i.e., a dependency between the values of the variables). Thus, such an analysis could not be used to prove that a program *does not use* a subset of its input variables, but would be used to prove that a program *uses* a subset of its input variables.

Semantics formulations using *sets of sets of traces* have already been proposed in the literature [6,28]. Mastroeni and Pasqua [28] lift the hierarchy of semantics developed by Cousot [12] to sets of sets of traces to obtain a hierarchy of semantics suitable for verifying general program properties (i.e., properties that are not subset-closed, cf. Sect. 7). However, *none* of the semantics that they proposed is suitable for input data usage: all semantics in the hierarchy are abstractions of a semantics that contains sets with both finite and infinite traces

[3] http://www.pm.inf.ethz.ch/research/lyra.html.

and thus, unlike our outcome semantics (cf. Sect. 5), cannot be used to reason about terminating and non-terminating outcomes of a program. Similarly, as observed in [28], the semantics proposed by Assaf et al. [6] can be used to verify only subset-closed properties. Thus, it cannot be used for input data usage.

Finally, to the best of our knowledge, our work is the first to aim at detecting programming errors in data science code using static analysis. Closely related are [7,10] which, however, focus on spreadsheet applications and target errors in the data rather than the code that analyzes it. Recent work [2] proposes an approach to repair *bias* in data science code. We believe that our work can be applied in this context to prove absence of bias, e.g., by showing that a program does not use gender information to decide whether to hire a person.

13 Conclusion and Future Work

In this paper, we have proposed an abstract interpretation framework to automatically detect input data that remains unused by a program. Additionally, we have shown that existing static analyses based on dependencies are subsumed by our unifying framework and can be used, with varying degrees of precision, for proving that a program does not use some of its input data. Finally, we have proposed a data usage analysis for more realistic data science applications that store input data in compound data structures such as arrays or lists.

As part of our future work, we plan to use our framework to guide the design of new, more precise static analyses for data usage. We also want to explore the complementary direction of proving that a program *uses* its input data by developing an analysis based on abstract dependencies [19,22,29] between program variables, as discussed above. Additionally, we plan to investigate other applications of our work such as provenance or lineage analysis [9] as well as proving absence of algorithmic bias [2]. Finally, we want to study other programming errors related to data usage such as accidental data duplication.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: POPL, pp. 147–160 (1999)
2. Albarghouthi, A., D'Antoni, L., Drews, S.: Repairing decision-making programs under uncertainty. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 181–200. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_9
3. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_8
4. Alpern, B., Schneider, F.B.: Defining Liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
5. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 100–115. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27864-1_10

6. Assaf, M., Naumann, D.A., Signoles, J., Totel, E., Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. In: POPL, pp. 874–887 (2017)
7. Barowy, D.W., Gochev, D., Berger, E.D.: CheckCell: data debugging for spreadsheets. In: OOPSLA, pp. 507–523 (2014)
8. Binkley, D., Gallagher, K.B.: Program slicing. Adv. Comput. **43**, 1–50 (1996)
9. Cheney, J., Ahmed, A., Acar, U.A.: Provenance as dependency analysis. Math. Struct. Comput. Sci. **21**(6), 1301–1337 (2011)
10. Cheng, T., Rival, X.: Static analysis of spreadsheet applications for type-unsafe operations detection. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 26–52. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_2
11. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
12. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoret. Comput. Sci. **277**(1–2), 47–103 (2002)
13. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Symposium on Programming, pp. 106–130 (1976)
14. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
15. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
16. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL, pp. 105–118 (2011)
17. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976)
18. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
19. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: parameterizing non-interference by abstract interpretation. In POPL, pp. 186–197 (2004)
20. Giegerich, R., Möncke, U., Wilhelm, R.: Invariance of approximate semantics with respect to program transformations. In: Brauer, W. (ed.) GI - 11. Jahrestagung. Informatik-Fachberichte, vol. 50. Springer, Heidelberg (1981). https://doi.org/10.1007/978-3-662-01089-1_1
21. Goguen, J.A., Meseguer, J.: Security policies and security models. In: S & P, pp. 11–20 (1982)
22. Halder, R., Cortesi, A.: Abstract program slicing on dependence condition graphs. Sci. Comput. Program. **78**(9), 1240–1263 (2013)
23. Herndon, T., Ash, M., Pollin, R.: Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff. Camb. J. Econ. **38**(2), 257–279 (2014)
24. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990)
25. Hunt, S., Sands, D.: On flow-sensitive security types. In: POPL, pp. 79–90 (2006)
26. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **3**(2), 125–143 (1977)
27. Leveson, N.G., Turner, C.S.: Investigation of the Therac-25 accidents. IEEE Comput. **26**(7), 18–41 (1993)
28. Mastroeni, I., Pasqua, M.: Hyperhierarchy of semantics - a formal framework for hyperproperties verification. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 232–252. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_12

29. Mastroeni, I., Zanardini, D.: Abstract program slicing: an abstract interpretation-based approach to program slicing. ACM Trans. Comput. Log. **18**(1), 7:1–7:58 (2017)
30. Mencinger, J., Aristovnik, A., Verbic, M.: The impact of growing public debt on economic growth in the European Union. Amfiteatru Econ. **16**(35), 403–414 (2014)
31. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006)
32. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
33. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_20
34. Urban, C.: The abstract domain of segmented ranking functions. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 43–62. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_5
35. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996)
36. Wehrli, S.: Static program analysis of data usage properties. Master's thesis, ETH Zurich, Zurich, Switzerland (2017)
37. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984)

Higher-Order Program Verification via HFL Model Checking

Naoki Kobayashi(✉), Takeshi Tsukada, and Keiichi Watanabe

The University of Tokyo, Tokyo, Japan
koba@is.s.u-tokyo.ac.jp

Abstract. There are two kinds of higher-order extensions of model checking: HORS model checking and HFL model checking. Whilst the former has been applied to automated verification of higher-order functional programs, applications of the latter have not been well studied. In the present paper, we show that various verification problems for functional programs, including may/must-reachability, trace properties, and linear-time temporal properties (and their negations), can be naturally reduced to (extended) HFL model checking. The reductions yield a sound and complete logical characterization of those program properties. Compared with the previous approaches based on HORS model checking, our approach provides a more uniform, streamlined method for higher-order program verification.

1 Introduction

There are two kinds of higher-order extensions of model checking in the literature: HORS model checking [16,32] and HFL model checking [42]. The former is concerned about whether the tree generated by a given higher-order tree grammar called a higher-order recursion scheme (HORS) satisfies the property expressed by a given modal μ-calculus formula (or a tree automaton), and the latter is concerned about whether a given finite state system satisfies the property expressed by a given formula of higher-order modal fixpoint logic (HFL), a higher-order extension of the modal μ-calculus. Whilst HORS model checking has been applied to automated verification of higher-order functional programs [17,18,22,26,33,41,43], there have been few studies on applications of HFL model checking to program/system verification. Despite that HFL has been introduced more than 10 years ago, we are only aware of applications to assume-guarantee reasoning [42] and process equivalence checking [28].

In the present paper, we show that various verification problems for higher-order functional programs can actually be reduced to (extended) HFL model checking in a rather natural manner. We briefly explain the idea of our reduction below.[1] We translate a program to an HFL formula that says "the program has a valid behavior" (where the *validity* of a behavior depends on each verification

[1] In this section, we use only a fragment of HFL that can be expressed in the modal μ-calculus. Some familiarity with the modal μ-calculus [25] would help.

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 711–738, 2018.
https://doi.org/10.1007/978-3-319-89884-1_25

problem). Thus, a program is actually mapped to a *property*, and a program property is mapped to a system to be verified; this has been partially inspired by the recent work of Kobayashi et al. [19], where HORS model checking problems have been translated to HFL model checking problems by switching the roles of models and properties.

For example, consider a simple program fragment $\texttt{read}(x);\texttt{close}(x)$ that reads and then closes a file (pointer) x. The transition system in Fig. 1 shows a valid access protocol to read-only files. Then, the property that a read operation is allowed in the current state can be expressed by a formula of the form $\langle\texttt{read}\rangle\varphi$, which says that the current state has a `read`-transition, after which φ is satisfied. Thus, the program $\texttt{read}(x);\texttt{close}(x)$ being valid is expressed as $\langle\texttt{read}\rangle\langle\texttt{close}\rangle\mathbf{true}$,[2] which is indeed satisfied by the initial state q_0 of the transition system in Fig. 1. Here, we have just replaced the operations `read` and `close` of the program with the corresponding modal operators $\langle\texttt{read}\rangle$ and $\langle\texttt{close}\rangle$. We can also naturally deal with branches and recursions. For example, consider the program $\texttt{close}(x)\Box(\texttt{read}(x);\texttt{close}(x))$, where $e_1\Box e_2$ represents a non-deterministic choice between e_1 and e_2. Then the property that the program always accesses x in a valid manner can be expressed by $(\langle\texttt{close}\rangle\mathbf{true}) \wedge (\langle\texttt{read}\rangle\langle\texttt{close}\rangle\mathbf{true})$. Note that we have just replaced the non-deterministic branch with the logical conjunction, as we wish here to require that the program's behavior is valid in *both* branches. We can also deal with conditional branches if HFL is extended with predicates; **if** b **then** $\texttt{close}(x)$ **else** $(\texttt{read}(x);\texttt{close}(x))$ can be translated to $(b \Rightarrow \langle\texttt{close}\rangle\mathbf{true}) \wedge (\neg b \Rightarrow \langle\texttt{read}\rangle\langle\texttt{close}\rangle\mathbf{true})$. Let us also consider the recursive function f defined by:

$$f\,x = \texttt{close}(x)\Box(\texttt{read}(x);\texttt{read}(x);f\,x),$$

Then, the program $f\,x$ being valid can be represented by using a (greatest) fixpoint formula:

$$\nu F.(\langle\texttt{close}\rangle\mathbf{true}) \wedge (\langle\texttt{read}\rangle\langle\texttt{read}\rangle F).$$

If the state q_0 satisfies this formula (which is indeed the case), then we know that all the file accesses made by $f\,x$ are valid. So far, we have used only the modal μ-calculus formulas. If we wish to express the validity of higher-order programs, we need HFL formulas; such examples are given later.

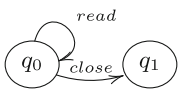


Fig. 1. File access protocol

[2] Here, for the sake of simplicity, we assume that we are interested in the usage of the single file pointer x, so that the name x can be ignored in HFL formulas; usage of multiple files can be tracked by using the technique of [17].

We generalize the above idea and formalize reductions from various classes of verification problems for simply-typed higher-order functional programs with recursion, integers and non-determinism – including verification of may/must-reachability, trace properties, and linear-time temporal properties (and their negations) – to (extended) HFL model checking where HFL is extended with integer predicates, and prove soundness and completeness of the reductions. Extended HFL model checking problems obtained by the reductions are (necessarily) undecidable in general, but for finite-data programs (i.e., programs that consist of only functions and data from finite data domains such as Booleans), the reductions yield *pure* HFL model checking problems, which are decidable [42].

Our reductions provide sound and complete logical characterizations of a wide range of program properties mentioned above. Nice properties of the logical characterizations include: (i) (like verification conditions for Hoare triples,) once the logical characterization is obtained as an HFL formula, purely logical reasoning can be used to prove or disprove it (without further referring to the program semantics); for that purpose, one may use theorem provers with various degrees of automation, ranging from interactive ones like Coq, semi-automated ones requiring some annotations, to fully automated ones (though the latter two are yet to be implemented), (ii) (unlike the standard verification condition generation for Hoare triples using invariant annotations) the logical characterization can *automatically* be computed, without any annotations,[3] (iii) standard logical reasoning can be applied based on the semantics of formulas; for example, co-induction and induction can be used for proving ν- and μ-formulas respectively, and (iv) thanks to the completeness, the set of program properties characterizable by HFL formula is closed under negations; for example, from a formula characterizing may-reachability, one can obtain a formula characterizing non-reachability by just taking the De Morgan dual.

Compared with previous approaches based on HORS model checking [18, 22, 26, 33, 37], our approach based on (extended) HFL model checking provides more uniform, streamlined methods for higher-order program verification. HORS model checking provides sound and complete verification methods for *finite-data* programs [17, 18], but for infinite-data programs, other techniques such as predicate abstraction [22] and program transformation [27, 31] had to be combined to obtain sound (but incomplete) reductions to HORS model checking. Furthermore, the techniques were different for each of program properties, such as reachability [22], termination [27], non-termination [26], fair termination [31], and fair non-termination [43]. In contrast, our reductions are sound and complete even for infinite-data programs. Although the obtained HFL model checking problems are undecidable in general, the reductions allow us to treat various program properties uniformly; all the verifications are boiled down to the issue of how to prove μ- and ν-formulas (and as remarked above, we can use induction and co-induction to deal with them). Technically, our reduction to HFL model

[3] This does not mean that invariant discovery is unnecessary; invariant discovery is just postponed to the later phase of discharging verification conditions, so that it can be uniformly performed among various verification problems.

checking may actually be considered an extension of HORS model checking in the following sense. HORS model checking algorithms [21,32] usually consist of two phases, one for computing a kind of higher-order "procedure summaries" in the form of variable profiles [32] or intersection types [21], and the other for nested least/greatest fixpoint computations. Our reduction from program verification to extended HFL model checking (the reduction given in Sect. 7, in particular) can be regarded as an extension of the first phase to deal with infinite data domains, where the problem for the second phase is expressed in the form of extended HFL model checking: see [23] for more details.

The rest of this paper is structured as follows. Section 2 introduces HFL extended with integer predicates and defines the HFL model checking problem. Section 3 informally demonstrates some examples of reductions from program verification problems to HFL model checking. Section 4 introduces a functional language used to formally discuss the reductions in later sections. Sections 5, 6, and 7 consider may/must-reachability, trace properties, and temporal properties respectively, and present (sound and complete) reductions from verification of those properties to HFL model checking. Section 8 discusses related work, and Sect. 9 concludes the paper. Proofs are found in an extended version [23].

2 (Extended) HFL

In this section, we introduce an extension of higher-order modal fixpoint logic (HFL) [42] with integer predicates (which we call $\mathrm{HFL}_{\mathbf{Z}}$; we often drop the subscript and write HFL, as in Sect. 1), and define the $\mathrm{HFL}_{\mathbf{Z}}$ model checking problem. The set of integers can actually be replaced by another infinite set X of data (like the set of natural numbers or the set of finite trees) to yield HFL_X.

2.1 Syntax

For a map f, we write $dom(f)$ and $codom(f)$ for the domain and codomain of f respectively. We write $\mathbf{Z}$ for the set of integers, ranged over by the meta-variable n below. We assume a set **Pred** of primitive predicates on integers, ranged over by p. We write $\mathtt{arity}(p)$ for the arity of p. We assume that **Pred** contains standard integer predicates such as $=$ and $<$, and also assume that, for each predicate $p \in \mathbf{Pred}$, there also exists a predicate $\neg p \in \mathbf{Pred}$ such that, for any integers $n_1, \ldots, n_k$, $p(n_1, \ldots, n_k)$ holds if and only if $\neg p(n_1, \ldots, n_k)$ does not hold; thus, $\neg p(n_1, \ldots, n_k)$ should be parsed as $(\neg p)(n_1, \ldots, n_k)$, but can semantically be interpreted as $\neg(p(n_1, \ldots, n_k))$.

The syntax of $HFL_{\mathbf{Z}}$ *formulas* is given by:

$$\begin{array}{l}\varphi \text{ (formulas)} ::= n \mid \varphi_1 \ \mathtt{op} \ \varphi_2 \mid \mathbf{true} \mid \mathbf{false} \mid p(\varphi_1, \ldots, \varphi_k) \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \\ \qquad\qquad \mid X \mid \langle a \rangle \varphi \mid [a]\varphi \mid \mu X^{\tau}.\varphi \mid \nu X^{\tau}.\varphi \mid \lambda X : \sigma.\varphi \mid \varphi_1 \ \varphi_2 \\ \tau \text{ (types)} ::= \bullet \mid \sigma \to \tau \qquad \sigma \text{ (extended types)} ::= \tau \mid \mathtt{int}\end{array}$$

Here, $\mathtt{op}$ ranges over a set of binary operations on integers, such as $+$, and X ranges over a denumerable set of variables. We have extended the original HFL [42] with integer expressions (n and $\varphi_1 \ \mathtt{op} \ \varphi_2$), and atomic formulas

$p(\varphi_1, \ldots, \varphi_k)$ on integers (here, the arguments of integer operations or predicates will be restricted to integer expressions by the type system introduced below). Following [19], we have omitted negations, as any formula can be transformed to an equivalent negation-free formula [30].

We explain the meaning of each formula informally; the formal semantics is given in Sect. 2.2. Like modal μ-calculus [10,25], each formula expresses a property of a labeled transition system. The first line of the syntax of formulas consists of the standard constructs of predicate logics. On the second line, as in the standard modal μ-calculus, $\langle a \rangle \varphi$ means that there exists an a-labeled transition to a state that satisfies φ. The formula $[a]\varphi$ means that after any a-labeled transition, φ is satisfied. The formulas $\mu X^\tau.\varphi$ and $\nu X^\tau.\varphi$ represent the least and greatest fixpoints respectively (the least and greatest X that $X = \varphi$) respectively; unlike the modal μ-calculus, X may range over not only propositional variables but also higher-order predicate variables (of type τ). The λ-abstractions $\lambda X{:}\sigma.\varphi$ and applications $\varphi_1\ \varphi_2$ are used to manipulate higher-order predicates. We often omit type annotations in $\mu X^\tau.\varphi$, $\nu X^\tau.\varphi$ and $\lambda X : \sigma.\varphi$, and just write $\mu X.\varphi$, $\nu X.\varphi$ and $\lambda X.\varphi$.

Example 1. Consider $\varphi_{\mathtt{ab}}\ \varphi$ where $\varphi_{\mathtt{ab}} = \mu X^{\bullet\to\bullet}.\lambda Y : \bullet. Y \vee \langle \mathtt{a} \rangle (X(\langle \mathtt{b} \rangle Y))$. We can expand the formula as follows:

$$\begin{aligned}\varphi_{\mathtt{ab}}\ \varphi &= (\lambda Y.\ \bullet\ .Y \vee \langle \mathtt{a} \rangle (\varphi_{\mathtt{ab}}(\langle \mathtt{b} \rangle Y)))\varphi = \varphi \vee \langle \mathtt{a} \rangle (\varphi_{\mathtt{ab}}(\langle \mathtt{b} \rangle \varphi)) \\ &= \varphi \vee \langle \mathtt{a} \rangle (\langle \mathtt{b} \rangle \varphi \vee \langle \mathtt{a} \rangle (\varphi_{\mathtt{ab}}(\langle \mathtt{b} \rangle \langle \mathtt{b} \rangle \varphi))) = \cdots,\end{aligned}$$

and obtain $\varphi \vee (\langle \mathtt{a} \rangle \langle \mathtt{b} \rangle \varphi) \vee (\langle \mathtt{a} \rangle \langle \mathtt{a} \rangle \langle \mathtt{b} \rangle \langle \mathtt{b} \rangle \varphi) \vee \cdots$. Thus, the formula means that there is a transition sequence of the form $\mathtt{a}^n\mathtt{b}^n$ for some $n \geq 0$ that leads to a state satisfying φ.

Following [19], we exclude out unmeaningful formulas such as $(\langle a \rangle \mathbf{true}){+}1$ by using a simple type system. The types $\bullet$, $\mathtt{int}$, and $\sigma \to \tau$ describe propositions, integers, and (monotonic) functions from σ to τ, respectively. Note that the integer type $\mathtt{int}$ may occur only in an argument position; this restriction is required to ensure that least and greatest fixpoints are well-defined. The typing rules for formulas are given in Fig. 2. In the figure, Δ denotes a type environment, which is a finite map from variables to (extended) types. Below we consider only well-typed formulas.

2.2 Semantics and $\mathrm{HFL}_{\mathbf{Z}}$ Model Checking

We now define the formal semantics of $\mathrm{HFL}_{\mathbf{Z}}$ formulas. A *labeled transition system* (LTS) is a quadruple $\mathtt{L} = (U, A, \longrightarrow, \mathtt{s}_{\mathtt{init}})$, where U is a finite set of states, A is a finite set of actions, $\longrightarrow \subseteq U \times A \times U$ is a labeled transition relation, and $\mathtt{s}_{\mathtt{init}} \in U$ is the initial state. We write $\mathtt{s}_1 \xrightarrow{a} \mathtt{s}_2$ when $(\mathtt{s}_1, a, \mathtt{s}_2) \in \longrightarrow$.

For an LTS $\mathtt{L} = (U, A, \longrightarrow, \mathtt{s}_{\mathtt{init}})$ and an extended type σ, we define the partially ordered set $(\mathcal{D}_{\mathtt{L},\sigma}, \sqsubseteq_{\mathtt{L},\sigma})$ inductively by:

$$\begin{array}{l}\mathcal{D}_{\mathtt{L},\bullet} = 2^U \qquad \sqsubseteq_{\mathtt{L},\bullet} = \subseteq \qquad \mathcal{D}_{\mathtt{L},\mathtt{int}} = \mathbf{Z} \qquad \sqsubseteq_{\mathtt{L},\mathtt{int}} = \{(n,n) \mid n \in \mathbf{Z}\} \\ \mathcal{D}_{\mathtt{L},\sigma\to\tau} = \{f \in \mathcal{D}_{\mathtt{L},\sigma} \to \mathcal{D}_{\mathtt{L},\tau} \mid \forall x, y.(x \sqsubseteq_{\mathtt{L},\sigma} y \Rightarrow f\ x \sqsubseteq_{\mathtt{L},\tau} f\ y)\} \\ \sqsubseteq_{\mathtt{L},\sigma\to\tau} = \{(f,g) \mid \forall x \in \mathcal{D}_{\mathtt{L},\sigma}.f(x) \sqsubseteq_{\mathtt{L},\tau} g(x)\}\end{array}$$

$$\frac{}{\Delta \vdash_{\mathtt{H}} n : \mathtt{int}}\ (\text{HT-Int}) \qquad \frac{\Delta \vdash_{\mathtt{H}} \varphi_i : \mathtt{int} \text{ for each } i \in \{1,2\}}{\Delta \vdash_{\mathtt{H}} \varphi_1\ \mathtt{op}\ \varphi_2 : \mathtt{int}}\ (\text{HT-Op})$$

$$\frac{}{\Delta \vdash_{\mathtt{H}} \mathbf{true} : \bullet}\ (\text{HT-True}) \qquad \frac{}{\Delta \vdash_{\mathtt{H}} \mathbf{false} : \bullet}\ (\text{HT-False})$$

$$\frac{\mathtt{arity}(p) = k \quad \Delta \vdash_{\mathtt{H}} \varphi_i : \mathtt{int} \text{ for each } i \in \{1,\ldots,k\}}{\Delta \vdash_{\mathtt{H}} p(\varphi_1,\ldots,\varphi_k) : \bullet}\ (\text{HT-Pred})$$

$$\frac{}{\Delta, X : \sigma \vdash_{\mathtt{H}} X : \sigma}\ (\text{HT-Var}) \qquad \frac{\Delta \vdash_{\mathtt{H}} \varphi_i : \bullet \text{ for each } i \in \{1,2\}}{\Delta \vdash_{\mathtt{H}} \varphi_1 \vee \varphi_2 : \bullet}\ (\text{HT-Or})$$

$$\frac{\Delta \vdash_{\mathtt{H}} \varphi_i : \bullet \text{ for each } i \in \{1,2\}}{\Delta \vdash_{\mathtt{H}} \varphi_1 \wedge \varphi_2 : \bullet}\ (\text{HT-And}) \qquad \frac{\Delta \vdash_{\mathtt{H}} \varphi : \bullet}{\Delta \vdash_{\mathtt{H}} \langle a \rangle \varphi : \bullet}\ (\text{HT-Some})$$

$$\frac{\Delta \vdash_{\mathtt{H}} \varphi : \bullet}{\Delta \vdash_{\mathtt{H}} [a] \varphi : \bullet}\ (\text{HT-All}) \qquad \frac{\Delta, X : \tau \vdash_{\mathtt{H}} \varphi : \tau}{\Delta \vdash_{\mathtt{H}} \mu X^{\tau}.\ \varphi : \tau}\ (\text{HT-Mu})$$

$$\frac{\Delta, X : \tau \vdash_{\mathtt{H}} \varphi : \tau}{\Delta \vdash_{\mathtt{H}} \nu X^{\tau}.\ \varphi : \tau}\ (\text{HT-Nu}) \qquad \frac{\Delta, X : \sigma \vdash_{\mathtt{H}} \varphi : \tau}{\Delta \vdash_{\mathtt{H}} \lambda X : \sigma.\ \varphi : \sigma \to \tau}\ (\text{HT-Abs})$$

$$\frac{\Delta \vdash_{\mathtt{H}} \varphi_1 : \sigma \to \tau \quad \Delta \vdash_{\mathtt{H}} \varphi_2 : \sigma}{\Delta \vdash_{\mathtt{H}} \varphi_1\ \varphi_2 : \tau}\ (\text{HT-App})$$

Fig. 2. Typing rules for HFL$_{\mathbf{Z}}$ formulas

Note that $(\mathcal{D}_{\mathtt{L},\tau}, \sqsubseteq_{\mathtt{L},\tau})$ forms a complete lattice (but $(\mathcal{D}_{\mathtt{L},\mathtt{int}}, \sqsubseteq_{\mathtt{L},\mathtt{int}})$ does not). We write $\bot_{\mathtt{L},\tau}$ and $\top_{\mathtt{L},\tau}$ for the least and greatest elements of $\mathcal{D}_{\mathtt{L},\tau}$ (which are $\lambda\widetilde{x}.\emptyset$ and $\lambda\widetilde{x}.U$) respectively. We sometimes omit the subscript $\mathtt{L}$ below. Let $[\![\Delta]\!]_{\mathtt{L}}$ be the set of functions (called *valuations*) that maps X to an element of $\mathcal{D}_{\mathtt{L},\sigma}$ for each $X : \sigma \in \Delta$. For an HFL formula φ such that $\Delta \vdash_{\mathtt{H}} \varphi : \sigma$, we define $[\![\Delta \vdash_{\mathtt{H}} \varphi : \sigma]\!]_{\mathtt{L}}$ as a map from $[\![\Delta]\!]_{\mathtt{L}}$ to $\mathcal{D}_{\sigma}$, by induction on the derivation[4] of $\Delta \vdash_{\mathtt{H}} \varphi : \sigma$, as follows.

$$[\![\Delta \vdash_{\mathtt{H}} n : \mathtt{int}]\!]_{\mathtt{L}}(\rho) = n \qquad [\![\Delta \vdash_{\mathtt{H}} \mathbf{true} : \bullet]\!]_{\mathtt{L}}(\rho) = U \qquad [\![\Delta \vdash_{\mathtt{H}} \mathbf{false} : \bullet]\!]_{\mathtt{L}}(\rho) = \emptyset$$

$$[\![\Delta \vdash_{\mathtt{H}} \varphi_1\ \mathtt{op}\ \varphi_2 : \mathtt{int}]\!]_{\mathtt{L}}(\rho) = ([\![\Delta \vdash_{\mathtt{H}} \varphi_1 : \mathtt{int}]\!]_{\mathtt{L}}(\rho))[\![\mathtt{op}]\!]([\![\Delta \vdash_{\mathtt{H}} \varphi_2 : \mathtt{int}]\!]_{\mathtt{L}}(\rho))$$

$$[\![\Delta \vdash_{\mathtt{H}} p(\varphi_1,\ldots,\varphi_k) : \bullet]\!]_{\mathtt{L}}(\rho) =$$

$$\begin{cases} U \text{ if } ([\![\Delta \vdash_{\mathtt{H}} \varphi_1 : \mathtt{int}]\!]_{\mathtt{L}}(\rho),\ldots,[\![\Delta \vdash_{\mathtt{H}} \varphi_k : \mathtt{int}]\!]_{\mathtt{L}}(\rho)) \in [\![p]\!] \\ \emptyset \text{ otherwise} \end{cases}$$

$$[\![\Delta, X : \sigma \vdash_{\mathtt{H}} X : \sigma]\!]_{\mathtt{L}}(\rho) = \rho(X)$$

$$[\![\Delta \vdash_{\mathtt{H}} \varphi_1 \vee \varphi_2 : \bullet]\!]_{\mathtt{L}}(\rho) = [\![\Delta \vdash_{\mathtt{H}} \varphi_1 : \bullet]\!]_{\mathtt{L}}(\rho) \cup [\![\Delta \vdash_{\mathtt{H}} \varphi_2 : \bullet]\!]_{\mathtt{L}}(\rho)$$

$$[\![\Delta \vdash_{\mathtt{H}} \varphi_1 \wedge \varphi_2 : \bullet]\!]_{\mathtt{L}}(\rho) = [\![\Delta \vdash_{\mathtt{H}} \varphi_1 : \bullet]\!]_{\mathtt{L}}(\rho) \cap [\![\Delta \vdash_{\mathtt{H}} \varphi_2 : \bullet]\!]_{\mathtt{L}}(\rho)$$

$$[\![\Delta \vdash_{\mathtt{H}} \langle a \rangle \varphi : \bullet]\!]_{\mathtt{L}}(\rho) = \{\mathtt{s} \mid \exists \mathtt{s}' \in [\![\Delta \vdash_{\mathtt{H}} \varphi : \bullet]\!]_{\mathtt{L}}(\rho).\ \mathtt{s} \xrightarrow{a} \mathtt{s}'\}$$

$$[\![\Delta \vdash_{\mathtt{H}} [a] \varphi : \bullet]\!]_{\mathtt{L}}(\rho) = \{\mathtt{s} \mid \forall \mathtt{s}' \in U.\ (\mathtt{s} \xrightarrow{a} \mathtt{s}' \text{ implies } \mathtt{s}' \in [\![\Delta \vdash_{\mathtt{H}} \varphi : \bullet]\!]_{\mathtt{L}}(\rho))\}$$

$$[\![\Delta \vdash_{\mathtt{H}} \mu X^{\tau}.\varphi : \tau]\!]_{\mathtt{L}}(\rho) = \mathbf{lfp}_{\mathtt{L},\tau}([\![\Delta \vdash_{\mathtt{H}} \lambda X : \tau.\ \varphi : \tau \to \tau]\!]_{\mathtt{L}}(\rho))$$

$$[\![\Delta \vdash_{\mathtt{H}} \nu X^{\tau}.\varphi : \tau]\!]_{\mathtt{L}}(\rho) = \mathbf{gfp}_{\mathtt{L},\tau}([\![\Delta \vdash_{\mathtt{H}} \lambda X : \tau.\ \varphi : \tau \to \tau]\!]_{\mathtt{L}}(\rho))$$

$$[\![\Delta \vdash_{\mathtt{H}} \lambda X : \sigma.\ \varphi : \sigma \to \tau]\!]_{\mathtt{L}}(\rho) = \{(v, [\![\Delta, X : \sigma \vdash_{\mathtt{H}} \varphi : \tau]\!]_{\mathtt{L}}(\rho[X \mapsto v])) \mid v \in \mathcal{D}_{\mathtt{L},\sigma}\}$$

$$[\![\Delta \vdash_{\mathtt{H}} \varphi_1\ \varphi_2 : \tau]\!]_{\mathtt{L}}(\rho) = [\![\Delta \vdash_{\mathtt{H}} \varphi_1 : \sigma \to \tau]\!]_{\mathtt{L}}(\rho)([\![\Delta \vdash_{\mathtt{H}} \varphi_2 : \sigma]\!]_{\mathtt{L}}(\rho))$$

[4] Note that the derivation of each judgment $\Delta \vdash_{\mathtt{H}} \varphi : \sigma$ is unique if there is any.

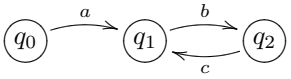
Here, $[\![\mathtt{op}]\!]$ denotes the binary function on integers represented by $\mathtt{op}$ and $[\![p]\!]$ denotes the k-ary relation on integers represented by p. The least/greatest fixpoint operators $\mathbf{lfp}_{\mathtt{L},\tau}$ and $\mathbf{gfp}_{\mathtt{L},\tau}$ are defined by $\mathbf{lfp}_{\mathtt{L},\tau}(f) = \bigsqcap_{\mathtt{L},\tau}\{x \in \mathcal{D}_{\mathtt{L},\tau} \mid f(x) \sqsubseteq_{\mathtt{L},\tau} x\}$ and $\mathbf{gfp}_{\mathtt{L},\tau}(f) = \bigsqcup_{\mathtt{L},\tau}\{x \in \mathcal{D}_{\mathtt{L},\tau} \mid x \sqsubseteq_{\mathtt{L},\tau} f(x)\}$. Here, $\bigsqcup_{\mathtt{L},\tau}$ and $\bigsqcap_{\mathtt{L},\tau}$ respectively denote the least upper bound and the greatest lower bound with respect to $\sqsubseteq_{\mathtt{L},\tau}$. We often omit the subscript $\mathtt{L}$ and write $[\![\Delta \vdash_{\mathtt{H}} \varphi : \sigma]\!]$ for $[\![\Delta \vdash_{\mathtt{H}} \varphi : \sigma]\!]_{\mathtt{L}}$. For a closed formula, i.e., a formula well-typed under the empty type environment $\emptyset$, we often write $[\![\varphi]\!]_{\mathtt{L}}$ or just $[\![\varphi]\!]$ for $[\![\emptyset \vdash_{\mathtt{H}} \varphi : \sigma]\!]_{\mathtt{L}}(\emptyset)$.

Example 2. For the LTS $\mathtt{L}_{file}$ in Fig. 1, we have:

$$[\![\nu X^{\bullet}.(\langle \mathtt{close}\rangle \mathbf{true} \wedge \langle \mathtt{read}\rangle X)]\!] = \mathbf{gfp}_{\mathtt{L},\bullet}(\lambda x \in \mathcal{D}_{\mathtt{L},\bullet}.[\![X : \bullet \vdash \langle \mathtt{close}\rangle \mathbf{true} \wedge \langle \mathtt{read}\rangle X : \bullet]\!](\{X \mapsto x\})) = \{q_0\}.$$

In fact, $x = \{q_0\} \in \mathcal{D}_{\mathtt{L},\bullet}$ satisfies the equation: $[\![X : \bullet \vdash \langle \mathtt{close}\rangle \mathbf{true} \wedge \langle \mathtt{read}\rangle X : \bullet]\!]_{\mathtt{L}}(\{X \mapsto x\}) = x$, and $x = \{q_0\} \in \mathcal{D}_{\mathtt{L},\bullet}$ is the greatest such element.

Consider the following LTS $\mathtt{L}_1$:



and $\varphi_{\mathtt{ab}}(\langle c\rangle \mathbf{true})$ where $\varphi_{\mathtt{ab}}$ is the one introduced in Example 1. Then, $[\![\varphi_{\mathtt{ab}}(\langle c\rangle \mathbf{true})]\!]_{\mathtt{L}_1} = \{q_0, q_2\}$.

Definition 1 ($\mathrm{HFL}_{\mathbf{Z}}$ model checking). *For a closed formula φ of type $\bullet$, we write $\mathtt{L}, \mathtt{s} \models \varphi$ if $\mathtt{s} \in [\![\varphi]\!]_{\mathtt{L}}$, and write $\mathtt{L} \models \varphi$ if $\mathtt{s}_{\mathtt{init}} \in [\![\varphi]\!]_{\mathtt{L}}$.* $\mathrm{HFL}_{\mathbf{Z}}$ model checking *is the problem of, given $\mathtt{L}$ and φ, deciding whether $\mathtt{L} \models \varphi$ holds.*

The $\mathrm{HFL}_{\mathbf{Z}}$ model checking problem is *un*decidable, due to the presence of integers; in fact, the semantic domain $\mathcal{D}_{\mathtt{L},\sigma}$ is not finite for σ that contains $\mathtt{int}$. The undecidability is obtained as a corollary of the soundness and completeness of the reduction from the may-reachability problem to HFL model checking discussed in Sect. 5. For the fragment of pure HFL (i.e., $\mathrm{HFL}_{\mathbf{Z}}$ without integers, which we write $\mathrm{HFL}_{\emptyset}$ below), the model checking problem is decidable [42].

The *order* of an $\mathrm{HFL}_{\mathbf{Z}}$ model checking problem $\mathtt{L} \stackrel{?}{\models} \varphi$ is the highest order of types of subformulas of φ, where the order of a type is defined by: $\mathtt{order}(\bullet) = \mathtt{order}(\mathtt{int}) = 0$ and $\mathtt{order}(\sigma \to \tau) = \max(\mathtt{order}(\sigma) + 1, \mathtt{order}(\tau))$. The complexity of order-k $\mathrm{HFL}_{\emptyset}$ model checking is k-EXPTIME complete [1], but polynomial time in the size of HFL formulas under the assumption that the other parameters (the size of LTS and the largest size of types used in formulas) are fixed [19].

Remark 1. Though we do not have quantifiers on integers as primitives, we can encode them using fixpoint operators. Given a formula $\varphi : \mathtt{int} \to \bullet$, we can express $\exists x : \mathtt{int}.\varphi(x)$ and $\forall x : \mathtt{int}.\varphi(x)$ by $(\mu X^{\mathtt{int}\to\bullet}.\lambda x : \mathtt{int}.\varphi(x) \vee X(x-1) \vee X(x+1))0$ and $(\nu X^{\mathtt{int}\to\bullet}.\lambda x : \mathtt{int}.\varphi(x) \wedge X(x-1) \wedge X(x+1))0$ respectively.

2.3 HES

As in [19], we often write an $\mathrm{HFL}_{\mathbf{Z}}$ formula as a sequence of fixpoint equations, called a *hierarchical equation system* (HES).

Definition 2. *An (extended)* hierarchical equation system *(HES) is a pair* $(\mathcal{E}, \varphi)$ *where* $\mathcal{E}$ *is a sequence of fixpoint equations, of the form:* $X_1^{\tau_1} =_{\alpha_1} \varphi_1; \cdots; X_n^{\tau_n} =_{\alpha_n} \varphi_n$*, where* $\alpha_i \in \{\mu, \nu\}$*. We assume that* $X_1 : \tau_1, \ldots, X_n : \tau_n \vdash_{\mathtt{H}} \varphi_i : \tau_i$ *holds for each* $i \in \{1, \ldots, n\}$*, and that* $\varphi_1, \ldots, \varphi_n, \varphi$ *do not contain any fixpoint operators.*

The HES $\Phi = (\mathcal{E}, \varphi)$ represents the $\mathrm{HFL}_{\mathbf{Z}}$ formula $\mathit{toHFL}(\mathcal{E}, \varphi)$ defined inductively by: $\mathit{toHFL}(\epsilon, \varphi) = \varphi$ and $\mathit{toHFL}(\mathcal{E}; X^{\tau} =_{\alpha} \varphi', \varphi) = \mathit{toHFL}([\alpha X^{\tau}.\varphi'/X]\mathcal{E}, [\alpha X^{\tau}.\varphi'/X]\varphi)$. Conversely, every $\mathrm{HFL}_{\mathbf{Z}}$ formula can be easily converted to an equivalent HES. In the rest of the paper, we often represent an $\mathrm{HFL}_{\mathbf{Z}}$ formula in the form of HES, and just call it an $\mathrm{HFL}_{\mathbf{Z}}$ formula. We write $[\![\Phi]\!]$ for $[\![\mathit{toHFL}(\Phi)]\!]$. An HES $(X_1^{\tau_1} =_{\alpha_1} \varphi_1; \cdots; X_n^{\tau_n} =_{\alpha_n} \varphi_n, \varphi)$ can be normalized to $(X_0^{\tau_0} =_{\nu} \varphi; X_1^{\tau_1} =_{\alpha_1} \varphi_1; \cdots; X_n^{\tau_n} =_{\alpha_n} \varphi_n, X_0)$ where τ_0 is the type of φ. Thus, we sometimes call just a sequence of equations $X_0^{\tau_0} =_{\nu} \varphi; X_1^{\tau_1} =_{\alpha_1} \varphi_1; \cdots; X_n^{\tau_n} =_{\alpha_n} \varphi_n$ an HES, with the understanding that "the main formula" is the first variable X_0. Also, we often write $X^{\tau}\ x_1 \cdots x_k =_{\alpha} \varphi$ for the equation $X^{\tau} =_{\alpha} \lambda x_1. \cdots \lambda x_k.\varphi$. We often omit type annotations and just write $X =_{\alpha} \varphi$ for $X^{\tau} =_{\alpha} \varphi$.

Example 3. The formula $\nu X.\mu Y.\langle \mathtt{b} \rangle X \vee \langle \mathtt{a} \rangle Y$ (which means that the current state has a transition sequence of the form $(\mathtt{a}^*\mathtt{b})^{\omega}$) is expressed as the following HES:

$$((X =_{\nu} Y; Y =_{\mu} \langle \mathtt{b} \rangle X \vee \langle \mathtt{a} \rangle Y), \quad X)\,.$$

3 Warming Up

To help readers get more familiar with $\mathrm{HFL}_{\mathbf{Z}}$ and the idea of reductions, we give here some variations of the examples of verification of file-accessing programs in Sect. 1, which are instances of the "resource usage verification problem" [15]. General reductions will be discussed in Sects. 5, 6 and 7, after the target language is set up in Sect. 4.

Consider the following OCaml-like program, which uses exceptions.

```
let readex x = read x; (if * then () else raise Eof) in
let rec f x = readex x; f x in
let d = open_in "foo" in try f d with Eof -> close d
```

Here, `*` represents a non-deterministic boolean value. The function `readex` reads the file pointer x, and then non-deterministically raises an end-of-file (`Eof`) exception. The main expression (on the third line) first opens file "foo", calls `f` to read the file repeatedly, and closes the file upon an end-of-file exception. Suppose, as in the example of Sect. 1, we wish to verify that the file "foo" is accessed following the protocol in Fig. 1.

First, we can remove exceptions by representing an exception handler as a special continuation [6]:

```
let readex x h k = read x; (if * then k() else h()) in
let rec f x h k = readex x h (fun _ -> f x h k) in
let d = open_in "foo" in f d (fun _ -> close d) (fun _ -> ())
```

Here, we have added to each function two parameters `h` and `k`, which represent an exception handler and a (normal) continuation respectively.

Let Φ be $(\mathcal{E}, F\ \mathbf{true}\ (\lambda r.\langle \mathtt{close}\rangle \mathbf{true})\ (\lambda r.\mathbf{true}))$ where $\mathcal{E}$ is:

$$\begin{array}{l} \mathit{Readex}\ x\ h\ k =_\nu \langle \mathtt{read}\rangle (k\ \mathbf{true} \wedge h\ \mathbf{true}); \\ F\ x\ h\ k =_\nu \mathit{Readex}\ x\ h\ (\lambda r.F\ x\ h\ k). \end{array}$$

Here, we have just replaced read/close operations with the modal operators $\langle \mathtt{read}\rangle$ and $\langle \mathtt{close}\rangle$, non-deterministic choice with a logical conjunction, and the unit value ( ) with **true**. Then, $\mathtt{L}_{\mathit{file}} \models \Phi$ if and only if the program performs only valid accesses to the file (e.g., it does not access the file after a close operation), where $\mathtt{L}_{\mathit{file}}$ is the LTS shown in Fig. 1. The correctness of the reduction can be informally understood by observing that there is a close correspondence between reductions of the program and those of the HFL formula above, and when the program reaches a read command `read` x, the corresponding formula is of the form $\langle \mathtt{read}\rangle \cdots$, meaning that the read operation is valid in the current state; a similar condition holds also for close operations. We will present a general translation and prove its correctness in Sect. 6.

Let us consider another example, which uses integers:

```
let rec f y x k = if y=0 then (close x; k())
                  else (read x; f (y-1) x k) in
let d = open_in "foo" in f n d (fun _ -> ())
```

Here, `n` is an integer constant. The function `f` reads `x` `y` times, and then calls the continuation `k`. Let $\mathtt{L}'_{\mathit{file}}$ be the LTS obtained by adding to $\mathtt{L}_{\mathit{file}}$ a new state q_2 and the transition $q_1 \xrightarrow{\mathtt{end}} q_2$ (which intuitively means that a program is allowed to terminate in the state q_1), and let Φ' be $(\mathcal{E}', F\ n\ \mathbf{true}\ (\lambda r.\langle \mathtt{end}\rangle \mathbf{true}))$ where $\mathcal{E}'$ is:

$$F\ y\ x\ k =_\mu (y = 0 \Rightarrow \langle \mathtt{close}\rangle (k\ \mathbf{true})) \wedge (y \neq 0 \Rightarrow \langle \mathtt{read}\rangle (F\ (y-1)\ x\ k)).$$

Here, $p(\varphi_1, \ldots, \varphi_k) \Rightarrow \varphi$ is an abbreviation of $\neg p(\varphi_1, \ldots, \varphi_k) \vee \varphi$. Then, $\mathtt{L}'_{\mathit{file}} \models \Phi'$ if and only if (i) the program performs only valid accesses to the file, (ii) it eventually terminates, and (iii) the file is closed when the program terminates. Notice the use of μ instead of ν above; by using μ, we can express liveness properties. The property $\mathtt{L}'_{\mathit{file}} \models \Phi'$ indeed holds for $n \geq 0$, but not for $n < 0$. In fact, $F\ n\ x\ k$ is equivalent to **false** for $n < 0$, and $\langle \mathtt{read}\rangle^n \langle \mathtt{close}\rangle (k\ \mathbf{true})$ for $n \geq 0$.

4 Target Language

This section sets up, as the target of program verification, a call-by-name[5] higher-order functional language extended with events. The language is essentially the same as the one used by Watanabe et al. [43] for discussing fair non-termination.

4.1 Syntax and Typing

We assume a finite set **Ev** of names called *events*, ranged over by a, and a denumerable set of variables, ranged over by $x, y, \ldots$. Events are used to express temporal properties of programs. We write $\widetilde{x}$ ($\widetilde{t}$, resp.) for a sequence of variables (terms, resp.), and write $|\widetilde{x}|$ for the length of the sequence.

A *program* is a pair (D, t) consisting of a set D of function definitions $\{f_1\ \widetilde{x}_1 = t_1, \ldots, f_n\ \widetilde{x}_n = t_n\}$ and a term t. The set of *terms*, ranged over by t, is defined by:

$$t ::= ( ) \mid x \mid n \mid t_1\ \mathtt{op}\ t_2 \mid \mathbf{event}\ a; t \mid \mathbf{if}\ p(t'_1, \ldots, t'_k)\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid t_1 t_2 \mid t_1 \Box t_2.$$

Here, n and p range over the sets of integers and integer predicates as in HFL formulas. The expression **event** $a; t$ raises an event a, and then evaluates t. Events are used to encode program properties of interest. For example, an assertion **assert**(b) can be expressed as **if** b **then** ( ) **else** (**event** `fail`; Ω), where `fail` is an event that expresses an assertion failure and Ω is a non-terminating term. If program termination is of interest, one can insert "**event** `end`" to every termination point and check whether an `end` event occurs. The expression $t_1 \Box t_2$ evaluates t_1 or t_2 in a non-deterministic manner; it can be used to model, e.g., unknown inputs from an environment. We use the meta-variable P for programs. When $P = (D, t)$ with $D = \{f_1\ \widetilde{x}_1 = t_1, \ldots, f_n\ \widetilde{x}_n = t_n\}$, we write **funs**$(P)$ for $\{f_1, \ldots, f_n\}$ (i.e., the set of function names defined in P). Using λ-abstractions, we sometimes write $f = \lambda\widetilde{x}.t$ for the function definition $f\ \widetilde{x} = t$. We also regard D as a map from function names to terms, and write $dom(D)$ for $\{f_1, \ldots, f_n\}$ and $D(f_i)$ for $\lambda\widetilde{x}_i.t_i$.

Any program (D, t) can be normalized to $(D \cup \{\mathbf{main} = t\}, \mathbf{main})$ where **main** is a name for the "main" function. We sometimes write just D for a program $(D, \mathbf{main})$, with the understanding that D contains a definition of **main**.

We restrict the syntax of expressions using a type system. The set of *simple types*, ranged over by κ, is defined by:

$$\kappa ::= \star \mid \eta \to \kappa \qquad\qquad \eta ::= \kappa \mid \mathtt{int}.$$

The types $\star$, `int`, and $\eta \to \kappa$ describe the unit value, integers, and functions from η to κ respectively. Note that `int` is allowed to occur only in argument

[5] Call-by-value programs can be handled by applying the CPS transformation before applying the reductions to HFL model checking.

positions. We defer typing rules to [23], as they are standard, except that we require that the righthand side of each function definition must have type $\star$; this restriction, as well as the restriction that `int` occurs only in argument positions, does not lose generality, as those conditions can be ensured by applying CPS transformation. We consider below only well-typed programs.

4.2 Operational Semantics

We define the labeled transition relation $t \xrightarrow{\ell}_D t'$, where ℓ is either ϵ or an event name, as the least relation closed under the rules in Fig. 3. We implicitly assume that the program (D, t) is well-typed, and this assumption is maintained throughout reductions by the standard type preservation property. In the rules for if-expressions, $[\![t'_i]\!]$ represents the integer value denoted by t'_i; note that the well-typedness of (D, t) guarantees that t'_i must be arithmetic expressions consisting of integers and integer operations; thus, $[\![t'_i]\!]$ is well defined. We often omit the subscript D when it is clear from the context. We write $t \xrightarrow{\ell_1 \cdots \ell_k}_D t'$ if $t \xrightarrow{\ell_1}_D \cdots \xrightarrow{\ell_k}_D t'$. Here, ϵ is treated as an empty sequence; thus, for example, we write $t \xrightarrow{ab}_D t'$ if $t \xrightarrow{a}_D \xrightarrow{\epsilon}_D \xrightarrow{b}_D \xrightarrow{\epsilon}_D t'$.

$$\frac{}{\mathbf{event}\ a; t \xrightarrow{a}_D t} \qquad \frac{f\widetilde{x} = u \in D \qquad |\widetilde{x}| = |\widetilde{t}|}{f\ \widetilde{t} \xrightarrow{\epsilon}_D [\widetilde{t}/\widetilde{x}]u} \qquad \frac{([\![t'_1]\!], \ldots, [\![t'_k]\!]) \in [\![p]\!]}{\mathbf{if}\ p(t'_1, \ldots, t'_k)\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \xrightarrow{\epsilon}_D t_1}$$

$$\frac{i \in \{1, 2\}}{t_1 \Box t_2 \xrightarrow{\epsilon}_D t_i} \qquad \frac{([\![t'_1]\!], \ldots, [\![t'_k]\!]) \notin [\![p]\!]}{\mathbf{if}\ p(t'_1, \ldots, t'_k)\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \xrightarrow{\epsilon}_D t_2}$$

Fig. 3. Labeled transition semantics

For a program $P = (D, t_0)$, we define the set $\mathbf{Traces}(P)(\subseteq \mathbf{Ev}^* \cup \mathbf{Ev}^\omega)$ of *traces* by:

$$\begin{aligned}\mathbf{Traces}(D, t_0) = \{\ell_0 \cdots \ell_{n-1} \in (\{\epsilon\} \cup \mathbf{Ev})^* \mid \forall i \in \{0, \ldots, n-1\}. t_i \xrightarrow{\ell_i}_D t_{i+1}\} \\ \cup \{\ell_0 \ell_1 \cdots \in (\{\epsilon\} \cup \mathbf{Ev})^\omega \mid \forall i \in \omega. t_i \xrightarrow{\ell_i}_D t_{i+1}\}.\end{aligned}$$

Note that since the label ϵ is regarded as an empty sequence, $\ell_0\ell_1\ell_2 = aa$ if $\ell_0 = \ell_2 = a$ and $\ell_1 = \epsilon$, and an element of $(\{\epsilon\} \cup \mathbf{Ev})^\omega$ is regarded as that of $\mathbf{Ev}^* \cup \mathbf{Ev}^\omega$. We write $\mathbf{FinTraces}(P)$ and $\mathbf{InfTraces}(P)$ for $\mathbf{Traces}(P) \cap \mathbf{Ev}^*$ and $\mathbf{Traces}(P) \cap \mathbf{Ev}^\omega$ respectively. The set of *full traces* $\mathbf{FullTraces}(D, t_0)(\subseteq \mathbf{Ev}^* \cup \mathbf{Ev}^\omega)$ is defined as:

$$\begin{aligned}\{\ell_0 \cdots \ell_{n-1} \in (\{\epsilon\} \cup \mathbf{Ev})^* \mid t_n = () \wedge \forall i \in \{0, \ldots, n-1\}. t_i \xrightarrow{\ell_i}_D t_{i+1}\} \\ \cup \{\ell_0 \ell_1 \cdots \in (\{\epsilon\} \cup \mathbf{Ev})^\omega \mid \forall i \in \omega. t_i \xrightarrow{\ell_i}_D t_{i+1}\}.\end{aligned}$$

Example 4. The last example in Sect. 1 is modeled as $P_{file} = (D, f\,())$, where $D = \{f\,x = (\textbf{event}\ \texttt{close}; ())\Box(\textbf{event}\ \texttt{read}; \textbf{event}\ \texttt{read}; f\,x)\}$. We have:

$\textbf{Traces}(P) = \{\texttt{read}^n \mid n \geq 0\} \cup \{\texttt{read}^{2n}\texttt{close} \mid n \geq 0\} \cup \{\texttt{read}^\omega\}$
$\textbf{FinTraces}(P) = \{\texttt{read}^n \mid n \geq 0\} \cup \{\texttt{read}^{2n}\texttt{close} \mid n \geq 0\}$
$\textbf{InfTraces}(P) = \{\texttt{read}^\omega\}$ $\textbf{FullTraces}(P) = \{\texttt{read}^{2n}\texttt{close} \mid n \geq 0\} \cup \{\texttt{read}^\omega\}$.

5 May/Must-Reachability Verification

Here we consider the following problems:

- May-reachability: "Given a program P and an event a, may P raise a?"
- Must-reachability: "Given a program P and an event a, must P raise a?"

Since we are interested in a particular event a, we restrict here the event set **Ev** to a singleton set of the form $\{a\}$. Then, the may-reachability is formalized as $a \stackrel{?}{\in} \textbf{Traces}(P)$, whereas the must-reachability is formalized as "does every trace in $\textbf{FullTraces}(P)$ contain a?" We encode both problems into the validity of $\text{HFL}_\textbf{Z}$ formulas (without any modal operators $\langle a\rangle$ or $[a]$), or the $\text{HFL}_\textbf{Z}$ model checking of those formulas against a trivial model (which consists of a single state without any transitions). Since our reductions are sound and complete, the characterizations of their negations –non-reachability and may-non-reachability– can also be obtained immediately. Although these are the simplest classes of properties among those discussed in Sects. 5, 6 and 7, they are already large enough to accommodate many program properties discussed in the literature, including lack of assertion failures/uncaught exceptions [22] (which can be characterized as non-reachability; recall the encoding of assertions in Sect. 4), termination [27,29] (characterized as must-reachability), and non-termination [26] (characterized as may-non-reachability).

5.1 May-Reachability

As in the examples in Sect. 3, we translate a program to a formula that says "the program may raise an event a" in a compositional manner. For example, $\textbf{event}\ a; t$ can be translated to **true** (since the event will surely be raised immediately), and $t_1\Box t_2$ can be translated to $t_1^\dagger \vee t_2^\dagger$ where $t_i^\dagger$ is the result of the translation of t_i (since only one of t_1 and t_2 needs to raise an event).

Definition 3. *Let $P = (D, t)$ be a program. $\Phi_{P,may}$ is the HES $(D^{\dagger_{may}}, t^{\dagger_{may}})$, where $D^{\dagger_{may}}$ and $t^{\dagger_{may}}$ are defined by:*

$$\{f_1\,\widetilde{x}_1 = t_1, \ldots, f_n\,\widetilde{x}_n = t_n\}^{\dagger_{may}} = \left(f_1\,\widetilde{x}_1 =_\mu {t_1}^{\dagger_{may}}; \cdots; f_n\,\widetilde{x}_n =_\mu {t_n}^{\dagger_{may}}\right)$$
$$()^{\dagger_{may}} = \textbf{false} \qquad x^{\dagger_{may}} = x \qquad n^{\dagger_{may}} = n \qquad (t_1\ \texttt{op}\ t_2)^{\dagger_{may}} = {t_1}^{\dagger_{may}}\ \texttt{op}\ {t_2}^{\dagger_{may}}$$
$$(\textbf{if}\ p(t_1', \ldots, t_k')\ \textbf{then}\ t_1\ \textbf{else}\ t_2)^{\dagger_{may}} =$$
$$\left(p({t_1'}^{\dagger_{may}}, \ldots, {t_k'}^{\dagger_{may}}) \wedge {t_1}^{\dagger_{may}}\right) \vee \left(\neg p({t_1'}^{\dagger_{may}}, \ldots, {t_k'}^{\dagger_{may}}) \wedge {t_2}^{\dagger_{may}}\right)$$
$$(\textbf{event}\ a; t)^{\dagger_{may}} = \textbf{true} \quad (t_1 t_2)^{\dagger_{may}} = {t_1}^{\dagger_{may}} {t_2}^{\dagger_{may}} \quad (t_1\Box t_2)^{\dagger_{may}} = {t_1}^{\dagger_{may}} \vee {t_2}^{\dagger_{may}}.$$

Note that, in the definition of $D^{\dagger_{may}}$, the order of function definitions in D does not matter (i.e., the resulting HES is unique up to the semantic equality), since all the fixpoint variables are bound by μ.

Example 5. Consider the program:

$$P_{loop} = (\{loop\ x = loop\ x\}, loop(\mathbf{event}\ a; (\,))).$$

It is translated to the HES $\Phi_{loop} = (loop\ x =_\mu loop\ x, loop(\mathbf{true}))$. Since $loop \equiv \mu loop.\lambda x.loop\ x$ is equivalent to $\lambda x.\mathbf{false}$, Φ_{loop} is equivalent to $\mathbf{false}$. In fact, P_{loop} never raises an event a (recall that our language is call-by-name).

Example 6. Consider the program $P_{sum} = (D_{sum}, \mathbf{main})$ where D_{sum} is:

$$\begin{array}{l}\mathbf{main} = sum\ n\ (\lambda r.\mathbf{assert}(r \geq n))\\ sum\ x\ k = \mathbf{if}\ x = 0\ \mathbf{then}\ k\,0\ \mathbf{else}\ sum\ (x-1)\ (\lambda r.k(x+r))\end{array}$$

Here, n is some integer constant, and $\mathbf{assert}(b)$ is the macro introduced in Sect. 4. We have used λ-abstractions for the sake of readability. The function *sum* is a CPS version of a function that computes the summation of integers from 1 to x. The main function computes the sum $r = 1 + \cdots + n$, and asserts $r \geq n$. It is translated to the HES $\Phi_{P_2,may} = (\mathcal{E}_{sum}, \mathbf{main})$ where $\mathcal{E}_{sum}$ is:

$$\begin{array}{l}\mathbf{main} =_\mu sum\ n\ (\lambda r.(r \geq n \wedge \mathbf{false}) \vee (r < n \wedge \mathbf{true}));\\ sum\ x\ k =_\mu (x = 0 \wedge k\,0) \vee (x \neq 0 \wedge sum\ (x-1)\ (\lambda r.k(x+r))).\end{array}$$

Here, n is treated as a constant. Since the shape of the formula does not depend on the value of n, the property "an assertion failure may occur for some n" can be expressed by $\exists n.\Phi_{P_2,may}$. □

The following theorem states that $\Phi_{P,may}$ is a complete characterization of the may-reachability of P.

Theorem 1. *Let P be a program. Then, $a \in \mathbf{Traces}(P)$ if and only if $\mathtt{L}_0 \models \Phi_{P,may}$ for $\mathtt{L}_0 = (\{\mathtt{s}_\star\}, \emptyset, \emptyset, \mathtt{s}_\star)$.*

A proof of the theorem above is found in [23]. We only provide an outline. We first show the theorem for recursion-free programs and then lift it to arbitrary programs by using the continuity of functions represented in the fixpoint-free fragment of $\mathrm{HFL}_{\mathbf{Z}}$ formulas. To show the theorem for recursion-free programs, we define the reduction relation $t \longrightarrow_D t'$ by:

$$\frac{f\widetilde{x} = u \in D \qquad |\widetilde{x}| = |\widetilde{t}|}{E[f\ \widetilde{t}] \longrightarrow_D E[[\widetilde{t}/\widetilde{x}]u]} \qquad \frac{([\![t'_1]\!], \ldots, [\![t'_k]\!]) \in [\![p]\!]}{E[\mathbf{if}\ p(t'_1, \ldots, t'_k)\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2] \longrightarrow_D E[t_1]}$$

$$\frac{([\![t'_1]\!], \ldots, [\![t'_k]\!]) \notin [\![p]\!]}{E[\mathbf{if}\ p(t'_1, \ldots, t'_k)\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2] \longrightarrow_D E[t_2]}$$

Here, E ranges over the set of evaluation contexts given by $E ::= [\,] \mid E\Box t \mid t\Box E \mid \mathbf{event}\ a; E$. The reduction relation differs from the labeled transition

relation given in Sect. 4, in that $\Box$ and **event** $a; \cdots$ are not eliminated. By the definition of the translation, the theorem holds for programs in normal form (with respect to the reduction relation), and the semantics of translated HFL formulas is preserved by the reduction relation; thus the theorem holds for recursion-free programs, as they are strongly normalizing.

5.2 Must-Reachability

The characterization of must-reachability can be obtained by an easy modification of the characterization of may-reachability: we just need to replace branches with logical conjunction.

Definition 4. *Let $P = (D, t)$ be a program. $\Phi_{P,must}$ is the HES $(D^{\dagger_{must}}, t^{\dagger_{must}})$, where $D^{\dagger_{must}}$ and $t^{\dagger_{must}}$ are defined by:*

$$\{f_1\,\widetilde{x}_1 = t_1, \ldots, f_n\,\widetilde{x}_n = t_n\}^{\dagger_{must}} = \left(f_1\,\widetilde{x}_1 =_\mu {t_1}^{\dagger_{must}}; \cdots; f_n\,\widetilde{x}_n =_\mu {t_n}^{\dagger_{must}}\right)$$
$$()^{\dagger_{must}} = \mathbf{false} \qquad x^{\dagger_{must}} = x \qquad n^{\dagger_{must}} = n \qquad (t_1\ \mathtt{op}\ t_2)^{\dagger_{must}} = {t_1}^{\dagger_{must}}\ \mathtt{op}\ {t_2}^{\dagger_{must}}$$
$$(\mathbf{if}\ p(t'_1, \ldots, t'_k)\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)^{\dagger_{must}} =$$
$$\left(p({t'_1}^{\dagger_{must}}, \ldots, {t'_k}^{\dagger_{must}}) \Rightarrow {t_1}^{\dagger_{must}}\right) \wedge \left(\neg p({t'_1}^{\dagger_{must}}, \ldots, {t'_k}^{\dagger_{must}}) \Rightarrow {t_2}^{\dagger_{must}}\right)$$
$$(\mathbf{event}\ a; t)^{\dagger_{must}} = \mathbf{true} \quad (t_1 t_2)^{\dagger_{must}} = {t_1}^{\dagger_{must}} {t_2}^{\dagger_{must}}\ (t_1 \Box t_2)^{\dagger_{must}} = {t_1}^{\dagger_{must}} \wedge {t_2}^{\dagger_{must}}.$$

Here, $p(\varphi_1, \ldots, \varphi_k) \Rightarrow \varphi$ is a shorthand for $\neg p(\varphi_1, \ldots, \varphi_k) \vee \varphi$.

Example 7. Consider $P_{\mathtt{loop}} = (D, \mathtt{loop}\, m\, n)$ where D is:

$$\begin{aligned}\mathtt{loop}\ x\ y = {} & \mathbf{if}\ x \leq 0 \vee y \leq 0\ \mathbf{then}\ (\mathbf{event}\ \mathtt{end}; ())\\ & \mathbf{else}\ (\mathtt{loop}\ (x-1)\ (y * y)) \Box (\mathtt{loop}\ x\ (y-1))\end{aligned}$$

Here, the event `end` is used to signal the termination of the program. The function `loop` non-deterministically updates the values of x and y until either x or y becomes non-positive. The must-termination of the program is characterized by $\Phi_{P_{\mathtt{loop}},must} = (\mathcal{E}, \mathtt{loop}\, m\, n)$ where $\mathcal{E}$ is:

$$\begin{aligned}\mathtt{loop}\ x\ y =_\mu {} & (x \leq 0 \vee y \leq 0 \Rightarrow \mathbf{true})\\ & \wedge (\neg(x \leq 0 \vee y \leq 0) \Rightarrow (\mathtt{loop}\ (x-1)\ (y * y)) \wedge (\mathtt{loop}\ x\ (y-1))).\end{aligned}$$

We write $\mathbf{Must}_a(P)$ if every $\pi \in \mathbf{FullTraces}(P)$ contains a. The following theorem, which can be proved in a manner similar to Theorem 1, guarantees that $\Phi_{P,must}$ is indeed a sound and complete characterization of the must-reachability.

Theorem 2. *Let P be a program. Then, $\boldsymbol{Must}_a(P)$ if and only if $\mathtt{L}_0 \models \Phi_{P,must}$ for $\mathtt{L}_0 = (\{\mathtt{s}_\star\}, \emptyset, \emptyset, \mathtt{s}_\star)$.*

6 Trace Properties

Here we consider the verification problem: "Given a (non-ω) regular language L and a program P, does *every* finite event sequence of P belong to L? (i.e. $\mathbf{FinTraces}(P) \stackrel{?}{\subseteq} L$)" and reduce it to an $\mathrm{HFL}_{\mathbf{Z}}$ model checking problem. The verification of file-accessing programs considered in Sect. 3 may be considered an instance of the problem.

Here we assume that the language L is closed under the prefix operation; this does not lose generality because $\mathbf{FinTraces}(P)$ is also closed under the prefix operation. We write $A_L = (Q, \Sigma, \delta, q_0, F)$ for the minimal, deterministic automaton with no dead states (hence the transition function δ may be partial). Since L is prefix-closed and the automaton is minimal, $w \in L$ if and only if $\hat{\delta}(q_0, w)$ is defined (where $\hat{\delta}$ is defined by: $\hat{\delta}(q, \epsilon) = q$ and $\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w)$). We use the corresponding LTS $\mathtt{L}_L = (Q, \Sigma, \{(q, a, q') \mid \delta(q, a) = q'\}, q_0)$ as the model of the reduced $\mathrm{HFL}_{\mathbf{Z}}$ model checking problem.

Given the LTS $\mathtt{L}_L$ above, whether an event sequence $a_1 \cdots a_k$ belongs to L can be expressed as $\mathtt{L}_L \stackrel{?}{\models} \langle a_1 \rangle \cdots \langle a_k \rangle \mathbf{true}$. Whether all the event sequences in $\{a_{j,1} \cdots a_{j,k_j} \mid j \in \{1, \ldots, n\}\}$ belong to L can be expressed as $\mathtt{L}_L \stackrel{?}{\models} \bigwedge_{j \in \{1,\ldots,n\}} \langle a_{j,1} \rangle \cdots \langle a_{j,k_j} \rangle \mathbf{true}$. We can lift these translations for event sequences to the translation from a program (which can be considered a description of a set of event sequences) to an $\mathrm{HFL}_{\mathbf{Z}}$ formula, as follows.

Definition 5. *Let $P = (D, t)$ be a program. $\Phi_{P,path}$ is the HES $(D^{\dagger_{path}}, t^{\dagger_{path}})$, where $D^{\dagger_{path}}$ and $t^{\dagger_{path}}$ are defined by:*

$$\{f_1\, \widetilde{x}_1 = t_1, \ldots, f_n\, \widetilde{x}_n = t_n\}^{\dagger_{path}} = \left(f_1\, \widetilde{x}_1 =_\nu {t_1}^{\dagger_{path}}; \cdots; f_n\, \widetilde{x}_n =_\nu {t_n}^{\dagger_{path}}\right)$$
$$(\,)^{\dagger_{path}} = \mathbf{true} \qquad x^{\dagger_{path}} = x \qquad n^{\dagger_{path}} = n \qquad (t_1\ \mathtt{op}\ t_2)^{\dagger_{path}} = {t_1}^{\dagger_{path}}\ \mathtt{op}\ {t_2}^{\dagger_{path}}$$
$$(\mathbf{if}\ p(t'_1, \ldots, t'_k)\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)^{\dagger_{path}} =$$
$$\left(p({t'_1}^{\dagger_{path}}, \ldots, {t'_k}^{\dagger_{path}}) \Rightarrow {t_1}^{\dagger_{path}}\right) \wedge \left(\neg p({t'_1}^{\dagger_{path}}, \ldots, {t'_k}^{\dagger_{path}}) \Rightarrow {t_2}^{\dagger_{path}}\right)$$
$$(\mathbf{event}\ a; t)^{\dagger_{path}} = \langle a \rangle t^{\dagger_{path}} \quad (t_1 t_2)^{\dagger_{path}} = {t_1}^{\dagger_{path}} {t_2}^{\dagger_{path}} \quad (t_1 \Box t_2)^{\dagger_{path}} = {t_1}^{\dagger_{path}} \wedge {t_2}^{\dagger_{path}}.$$

Example 8. The last program discussed in Sect. 3 is modeled as $P_2 = (D_2, f\ m\ g)$, where m is an integer constant and D_2 consists of:

$$f\ y\ k = \mathbf{if}\ y = 0\ \mathbf{then}\ (\mathbf{event}\ \mathtt{close}; k\,(\,))\ \mathbf{else}\ (\mathbf{event}\ \mathtt{read}; f\ (y - 1)\ k)$$
$$g\ r = \mathbf{event}\ \mathtt{end}; (\,)$$

Here, we have modeled accesses to the file, and termination as events. Then, $\Phi_{P_2,path} = (\mathcal{E}_{P_2,path}, f\ m\ g)$ where $\mathcal{E}_{P_2,path}$ is:[6]

$$f\ n\ k =_\nu (n = 0 \Rightarrow \langle \mathtt{close} \rangle (k\, \mathbf{true})) \wedge (n \neq 0 \Rightarrow \langle \mathtt{read} \rangle (f\ (n - 1)\ k))$$
$$g\ r =_\nu \langle \mathtt{end} \rangle \mathbf{true}.$$

Let L be the prefix-closure of $\mathtt{read}^* \cdot \mathtt{close} \cdot \mathtt{end}$. Then $\mathtt{L}_L$ is $\mathtt{L}'_{file}$ in Sect. 3, and $\mathbf{FinTraces}(P_2) \subseteq L$ can be verified by checking $\mathtt{L}_L \models \Phi_{P_2,path}$. □

[6] Unlike in Sect. 3, the variables are bound by ν since we are not concerned with the termination property here.

Theorem 3. *Let P be a program and L be a regular, prefix-closed language. Then,* $\mathbf{FinTraces}(P) \subseteq L$ *if and only if* $\mathtt{L}_L \models \Phi_{P,path}$.

As in Sect. 5, we first prove the theorem for programs in normal form, and then lift it to recursion-free programs by using the preservation of the semantics of $\mathrm{HFL}_{\mathbf{Z}}$ formulas by reductions, and further to arbitrary programs by using the (co-)continuity of the functions represented by fixpoint-free $\mathrm{HFL}_{\mathbf{Z}}$ formulas. See [23] for a concrete proof.

7 Linear-Time Temporal Properties

This section considers the following problem: "Given a program P and an ω-regular word language L, does $\mathbf{InfTraces}(P) \cap L = \emptyset$ hold?". From the viewpoint of program verification, L represents the set of "bad" behaviors. This can be considered an extension of the problems considered in the previous sections.

The reduction to HFL model checking is more involved than those in the previous sections. To see the difficulty, consider the program P_0:

$$(\{f = \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; f)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; f)\}, \quad f)\,,$$

where c is some boolean expression. Let L be the complement of $(\mathtt{a}^*\mathtt{b})^\omega$, i.e., the set of infinite sequences that contain only finitely many $\mathtt{b}$'s. Following Sect. 6 (and noting that $\mathbf{InfTraces}(P) \cap L = \emptyset$ is equivalent to $\mathbf{InfTraces}(P) \subseteq (\mathtt{a}^*\mathtt{b})^\omega$ in this case), one may be tempted to prepare an LTS like the one in Fig. 4 (which corresponds to the transition function of a (parity) word automaton accepting $(\mathtt{a}^*\mathtt{b})^\omega$), and translate the program to an HES Φ_{P_0} of the form:

$$(f =_\alpha (c \Rightarrow \langle \mathtt{a} \rangle f) \wedge (\neg c \Rightarrow \langle \mathtt{b} \rangle f), \quad f)\,,$$

where α is μ or ν. However, such a translation would not work. If $c = \mathbf{true}$, then $\mathbf{InfTraces}(P_0) = \mathtt{a}^\omega$, hence $\mathbf{InfTraces}(P_0) \cap L \neq \emptyset$; thus, α should be μ for Φ_{P_0} to be unsatisfied. If $c = \mathbf{false}$, however, $\mathbf{InfTraces}(P_0) = \mathtt{b}^\omega$, hence $\mathbf{InfTraces}(P_0) \cap L = \emptyset$; thus, α must be ν for Φ_{P_0} to be satisfied.

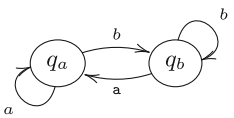


Fig. 4. LTS for $(a^*b)^\omega$

The example above suggests that we actually need to distinguish between the two occurrences of f in the body of f's definition. Note that in the then- and else-clauses respectively, f is called after different events $\mathtt{a}$ and $\mathtt{b}$. This difference

is important, since we are interested in whether $\mathtt{b}$ occurs infinitely often. We thus duplicate f, and replace the program with the following program P_{dup}:

$$(\{f_b = \textbf{if } c \textbf{ then } (\textbf{event } \mathtt{a}; f_a) \textbf{ else } (\textbf{event } \mathtt{b}; f_b),\\ f_a = \textbf{if } c \textbf{ then } (\textbf{event } \mathtt{a}; f_a) \textbf{ else } (\textbf{event } \mathtt{b}; f_b)\}, f_b).$$

For checking $\mathbf{InfTraces}(P_0) \cap L = \emptyset$, it is now sufficient to check that f_b is recursively called infinitely often. We can thus obtain the following HES:

$$((f_b =_\nu (c \Rightarrow \langle \mathtt{a} \rangle f_a) \wedge (\neg c \Rightarrow \langle \mathtt{b} \rangle f_b); \quad f_a =_\mu (c \Rightarrow \langle \mathtt{a} \rangle f_a) \wedge (\neg c \Rightarrow \langle \mathtt{b} \rangle f_b)), f_b).$$

Note that f_b and f_a are bound by ν and μ respectively, reflecting the fact that $\mathtt{b}$ should occur infinitely often, but $\mathtt{a}$ need not. If $c = \textbf{true}$, the formula is equivalent to $\nu f_b.\langle \mathtt{a} \rangle \mu f_a.\langle \mathtt{a} \rangle f_a$, which is false. If $c = \textbf{false}$, then the formula is equivalent to $\nu f_b.\langle \mathtt{b} \rangle f_b$, which is satisfied by by the LTS in Fig. 4.

The general translation is more involved due to the presence of higher-order functions, but, as in the example above, the overall translation consists of two steps. We first replicate functions according to what events may occur between two recursive calls, and reduce the problem $\mathbf{InfTraces}(P) \cap L \stackrel{?}{=} \emptyset$ to a problem of analyzing which functions are recursively called infinitely often, which we call a *call-sequence analysis*. We can then reduce the call-sequence analysis to HFL model checking in a rather straightforward manner (though the proof of the correctness is non-trivial). The resulting HFL formula actually does not contain modal operators.[7] So, as in Sect. 5, the resulting problem is the validity checking of HFL formulas without modal operators.

In the rest of this section, we first introduce the call-sequence analysis problem and its reduction to HFL model checking in Sect. 7.1. We then show how to reduce the temporal verification problem $\mathbf{InfTraces}(P) \cap L \stackrel{?}{=} \emptyset$ to an instance of the call-sequence analysis problem in Sect. 7.2.

7.1 Call-Sequence Analysis

We define the call-sequence analysis and reduce it to an HFL model-checking problem. As mentioned above, in the call-sequence analysis, we are interested in analyzing which functions are *recursively called* infinitely often. Here, we say that g is *recursively called from* f, if $f\,\widetilde{s} \xrightarrow{\epsilon}_D [\widetilde{s}/\widetilde{x}]t_f \xrightarrow{\widetilde{\ell}}{}^*_D g\,\widetilde{t}$, where $f\,\widetilde{x} = t_f \in D$ and g "originates from" t_f (a more formal definition will be given in Definition 6 below). For example, consider the following program P_{app}, which is a twisted version of P_{dup} above.

$$(\{\mathtt{app}\,h\,x = h\,x,\\ f_b\,x = \textbf{if } x > 0 \textbf{ then } (\textbf{event } \mathtt{a}; \mathtt{app}\,f_a\,(x-1)) \textbf{ else } (\textbf{event } \mathtt{b}; \mathtt{app}\,f_b\,5),\\ f_a\,x = \textbf{if } x > 0 \textbf{ then } (\textbf{event } \mathtt{a}; \mathtt{app}\,f_a\,(x-1)) \textbf{ else } (\textbf{event } \mathtt{b}; \mathtt{app}\,f_b\,5)\}, f_b\,5).$$

[7] In the example above, we can actually remove $\langle \mathtt{a} \rangle$ and $\langle \mathtt{b} \rangle$, as information about events has been taken into account when f was duplicated.

Then f_a is "recursively called" from f_b in $f_b\,5 \xrightarrow{\mathtt{a}}{}^*_D\ \mathtt{app}\ f_a\,4 \xrightarrow{\epsilon}{}^*_D\ f_a\,4$ (and so is $\mathtt{app}$). We are interested in infinite chains of recursive calls $f_0 f_1 f_2 \cdots$, and which functions may occur infinitely often in each chain. For instance, the program above has the unique infinite chain $(f_b f_a^5)^\omega$, in which both f_a and f_b occur infinitely often. (Besides the infinite chain, the program has finite chains like $f_b\,\mathtt{app}$; note that the chain cannot be extended further, as the body of $\mathtt{app}$ does not have any occurrence of recursive functions: $\mathtt{app}, f_a$ and f_b.)

We define the notion of "recursive calls" and call-sequences formally below.

Definition 6 (Recursive call relation, call sequences). *Let $P = (D, f_1\,\widetilde{s})$ be a program, with $D = \{f_i\,\tilde{x}_i = u_i\}_{1\le i\le n}$. We define $D^\sharp := D \cup \{f_i^\sharp\,\tilde{x} = u_i\}_{1\le i\le n}$ where $f_1^\sharp, \ldots, f_n^\sharp$ are fresh symbols. (Thus, $D^\sharp$ has two copies of each function symbol, one of which is marked by $\sharp$.) For the terms $\widetilde{t_i}$ and $\widetilde{t_j}$ that do not contain marked symbols, we write $f_i\,\widetilde{t_i} \rightsquigarrow_D f_j\,\widetilde{t_j}$ if (i) $[\widetilde{t_i}/\widetilde{x_i}][f_1^\sharp/f_1, \ldots, f_n^\sharp/f_n]u_i \xrightarrow{\widetilde{\ell}}{}^*_{D^\sharp} f_j^\sharp\,\widetilde{t'_j}$ and (ii) $\widetilde{t_j}$ is obtained by erasing all the marks in $\widetilde{t'_j}$. We write* $\mathbf{Callseq}(P)$ *for the set of (possibly infinite) sequences of function symbols:*

$$\{f_1\,g_1\,g_2 \cdots \mid f_1\,\widetilde{s} \rightsquigarrow_D g_1\,\widetilde{t_1} \rightsquigarrow_D g_2\,\widetilde{t_2} \rightsquigarrow_D \cdots\}.$$

We write $\mathbf{InfCallseq}(P)$ *for the subset of* $\mathbf{Callseq}(P)$ *consisting of infinite sequences, i.e.,* $\mathbf{Callseq}(P) \cap \{f_1, \ldots, f_n\}^\omega$.

For example, for P_{app} above, $\mathbf{Callseq}(P)$ is the prefix closure of $\{(f_b f_a^5)^\omega\} \cup \{s \cdot \mathtt{app} \mid s$ is a non-empty finite prefix of $(f_b f_a^5)^\omega\}$, and $\mathbf{InfCallseq}(P)$ is the singleton set $\{(f_b f_a^5)^\omega\}$.

Definition 7 (Call-sequence analysis). *A* priority assignment *for a program P is a function $\Omega : \mathbf{funs}(P) \to \mathbb{N}$ from the set of function symbols of P to the set $\mathbb{N}$ of natural numbers. We write $\models_{csa} (P, \Omega)$ if every infinite call-sequence $g_0 g_1 g_2 \cdots \in \mathbf{InfCallseq}(P)$ satisfies the parity condition w.r.t. Ω, i.e., the largest number occurring infinitely often in $\Omega(g_0)\Omega(g_1)\Omega(g_2)\ldots$ is even.* Call-sequence analysis *is the problem of, given a program P with a priority assignment Ω, deciding whether $\models_{csa} (P, \Omega)$ holds.*

For example, for P_{app} and the priority assignment $\Omega_{app} = \{\mathtt{app} \mapsto 3, f_a \mapsto 1, f_b \mapsto 2\}$, $\models_{csa} (P_{app}, \Omega_{app})$ holds.

The call-sequence analysis can naturally be reduced to HFL model checking against the trivial LTS $\mathtt{L}_0 = (\{\mathtt{s}_\star\}, \emptyset, \emptyset, \mathtt{s}_\star)$ (or validity checking).

Definition 8. *Let $P = (D, t)$ be a program and Ω be a priority assignment for P. The HES $\Phi_{(P,\Omega),csa}$ is $(D^{\dagger_{csa}}, t^{\dagger_{csa}})$, where $D^{\dagger_{csa}}$ and $t^{\dagger_{csa}}$ are defined by:*

$$\{f_1\,\widetilde{x}_1 = t_1, \ldots, f_n\,\widetilde{x}_n = t_n\}^{\dagger_{csa}} = \left(f_1\,\widetilde{x}_1 =_{\alpha_1} {t_1}^{\dagger_{csa}}; \cdots; f_n\,\widetilde{x}_n =_{\alpha_n} {t_n}^{\dagger_{csa}}\right)$$
$$()^{\dagger_{csa}} = \mathbf{true} \qquad x^{\dagger_{csa}} = x \qquad n^{\dagger_{csa}} = n \qquad (t_1\ \mathtt{op}\ t_2)^{\dagger_{csa}} = {t_1}^{\dagger_{csa}}\ \mathtt{op}\ {t_2}^{\dagger_{csa}}$$
$$(\mathbf{if}\ p(t'_1, \ldots, t'_k)\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)^{\dagger_{csa}} =$$
$$\quad (p({t'_1}^{\dagger_{csa}}, \ldots, {t'_k}^{\dagger_{csa}}) \Rightarrow {t_1}^{\dagger_{csa}}) \wedge (\neg p({t'_1}^{\dagger_{csa}}, \ldots, {t'_k}^{\dagger_{csa}}) \Rightarrow {t_2}^{\dagger_{csa}})$$
$$(\mathbf{event}\ a; t)^{\dagger_{csa}} = t^{\dagger_{csa}} \qquad (t_1\,t_2)^{\dagger_{csa}} = {t_1}^{\dagger_{csa}}\,{t_2}^{\dagger_{csa}} \qquad (t_1 \Box t_2)^{\dagger_{csa}} = {t_1}^{\dagger_{csa}} \wedge {t_2}^{\dagger_{csa}}.$$

Here, we assume that $\Omega(f_i) \geq \Omega(f_{i+1})$ *for each* $i \in \{1, \ldots, n-1\}$*, and* $\alpha_i = \nu$ *if* $\Omega(f_i)$ *is even and* μ *otherwise.*

The following theorem states the soundness and completeness of the reduction. See [23] for a proof.

Theorem 4. *Let* P *be a program and* Ω *be a priority assignment for* P*. Then* $\models_{csa} (P, \Omega)$ *if and only if* $\mathtt{L}_0 \models \Phi_{(P,\Omega),csa}$*.*

Example 9. For P_{app} and Ω_{app} above, $(P_{app}, \Omega_{app})^{\dagger_{csa}} = (\mathcal{E}, f_b\, 5)$, where: $\mathcal{E}$ is:

$$\mathtt{app}\, h\, x =_\mu h\, x; \quad f_b\, x =_\nu (x > 0 \Rightarrow \mathtt{app}\, f_a\, (x-1)) \wedge (x \leq 0 \Rightarrow \mathtt{app}\, f_b\, 5);$$
$$f_a\, x =_\mu (x > 0 \Rightarrow \mathtt{app}\, f_a\, (x-1)) \wedge (x \leq 0 \Rightarrow \mathtt{app}\, f_b\, 5).$$

Note that $\mathtt{L}_0 \models (P_{app}, \Omega_{app})^{\dagger_{csa}}$ holds.

7.2 From Temporal Verification to Call-Sequence Analysis

This subsection shows a reduction from the temporal verification problem $\mathbf{InfTraces}(P) \cap L \stackrel{?}{=} \emptyset$ to a call-sequence analysis problem $\stackrel{?}{\models}_{csa} (P', \Omega)$.

For the sake of simplicity, we assume without loss of generality that every program $P = (D, t)$ in this section is non-terminating and every infinite reduction sequence produces infinite events, so that $\mathbf{FullTraces}(P) = \mathbf{InfTraces}(P)$ holds. We also assume that the ω-regular language L for the temporal verification problem is specified by using a non-deterministic, parity word automaton [10]. We recall the definition of non-deterministic, parity word automata below.

Definition 9 (Parity automaton). *A* non-deterministic parity word automaton *is a quintuple* $\mathcal{A} = (Q, \Sigma, \delta, q_I, \Omega)$ *where (i)* Q *is a finite set of states; (ii)* Σ *is a finite alphabet; (iii)* δ*, called a transition function, is a* total *map from* $Q \times \Sigma$ *to* 2^Q*; (iv)* $q_I \in Q$ *is the initial state; and (v)* $\Omega \in Q \to \mathbb{N}$ *is the priority function. A* run *of* $\mathcal{A}$ *on an* ω*-word* $a_0 a_1 \cdots \in \Sigma^\omega$ *is an infinite sequence of states* $\rho = \rho(0)\rho(1)\cdots \in Q^\omega$ *such that (i)* $\rho(0) = q_I$*, and (ii)* $\rho(i+1) \in \delta(\rho(i), a_i)$ *for each* $i \in \omega$*. An* ω*-word* $w \in \Sigma^\omega$ *is* accepted *by* $\mathcal{A}$ *if, there exists a run* ρ *of* $\mathcal{A}$ *on* w *such that* $\mathbf{max}\{\Omega(q) \mid q \in \mathbf{Inf}(\rho)\}$ *is even, where* $\mathbf{Inf}(\rho)$ *is the set of states that occur infinitely often in* ρ*. We write* $\mathcal{L}(\mathcal{A})$ *for the set of* ω*-words accepted by* $\mathcal{A}$*.*

For technical convenience, we assume below that $\delta(q, a) \neq \emptyset$ for every $q \in Q$ and $a \in \Sigma$; this does not lose generality since if $\delta(q, a) = \emptyset$, we can introduce a new "dead" state q_{dead} (with priority 1) and change $\delta(q, a)$ to $\{q_{dead}\}$. Given a parity automaton $\mathcal{A}$, we refer to each component of $\mathcal{A}$ by $Q_\mathcal{A}$, $\Sigma_\mathcal{A}$, $\delta_\mathcal{A}$, $q_{I,\mathcal{A}}$ and $\Omega_\mathcal{A}$.

Example 10. Consider the automaton $\mathcal{A}_{ab} = (\{q_a, q_b\}, \{\mathtt{a}, \mathtt{b}\}, \delta, q_a, \Omega)$, where δ is as given in Fig. 4, $\Omega(q_a) = 0$, and $\Omega(q_b) = 1$. Then, $\mathcal{L}(\mathcal{A}_{ab}) = \overline{(\mathtt{a}^*\mathtt{b})^\omega} = (\mathtt{a}^*\mathtt{b})^*\mathtt{a}^\omega$.

The goal of this subsection is, given a program P and a parity word automaton $\mathcal{A}$, to construct another program P' and a priority assignment Ω for P', such that $\mathbf{InfTraces}(P) \cap \mathcal{L}(\mathcal{A}) = \emptyset$ if and only if $\models_{csa} (P', \Omega)$.

Note that a necessary and sufficient condition for $\mathbf{InfTraces}(P) \cap \mathcal{L}(\mathcal{A}) = \emptyset$ is that no trace in $\mathbf{InfTraces}(P)$ has a run whose priority sequence satisfies the parity condition; in other words, for every sequence in $\mathbf{InfTraces}(P)$, and for every run for the sequence, the largest priority that occurs in the associated priority sequence is odd. As explained at the beginning of this section, we reduce this condition to a call sequence analysis problem by appropriately duplicating functions in a given program. For example, recall the program P_0:

$$(\{f = \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; f)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; f)\}, f)\,.$$

It is translated to P_0':

$$\begin{array}{l}(\{f_b = \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; f_a)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; f_b),\\ \quad f_a = \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; f_a)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; f_b)\},\ f_b),\end{array}$$

where c is some (closed) boolean expression. Since the largest priorities encountered before calling f_a and f_b (since the last recursive call) respectively are 0 and 1, we assign those priorities plus 1 (to flip odd/even-ness) to f_a and f_b respectively. Then, the problem of $\mathbf{InfTraces}(P_0) \cap \mathcal{L}(\mathcal{A}) = \emptyset$ is reduced to $\models_{csa} (P_0', \{f_a \mapsto 1, f_b \mapsto 2\})$. Note here that the priorities of f_a and f_b represent *summaries* of the priorities (plus one) that occur in the run of the automaton until f_a and f_b are respectively called since the last recursive call; thus, the largest priority of states that occur infinitely often in the run for an infinite trace is equivalent to the largest priority that occurs infinitely often in the sequence of summaries $(\Omega(f_1)-1)(\Omega(f_2)-1)(\Omega(f_3)-1)\cdots$ computed from a corresponding call sequence $f_1 f_2 f_3 \cdots$.

Due to the presence of higher-order functions, the general reduction is more complicated than the example above. First, we need to replicate not only function symbols, but also arguments. For example, consider the following variation P_1 of P_0 above:

$$(\{g\,k = \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; k)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; k), \quad f = g\,f\}, \quad f)\,.$$

Here, we have just made the calls to f indirect, by preparing the function g. Obviously, the two calls to k in the body of g must be distinguished from each other, since different priorities are encountered before the calls. Thus, we duplicate the argument k, and obtain the following program P_1':

$$(\{g\,k_a\,k_b = \mathbf{if}\ c\ \mathbf{then}\ (\mathbf{event}\ \mathtt{a}; k_a)\ \mathbf{else}\ (\mathbf{event}\ \mathtt{b}; k_b), f_a = g\,f_a\,f_b, f_b = g\,f_a\,f_b\}, f_a).$$

Then, for the priority assignment $\Omega = \{f_a \mapsto 1, f_b \mapsto 2, g \mapsto 1\}$, $\mathbf{InfTraces}(P_1) \cap \mathcal{L}(\mathcal{A}_{ab}) = \emptyset$ if and only if $\models_{csa} (P_1', \Omega)$. Secondly, we need to take into account not only the priorities of states visited by $\mathcal{A}$, but also the states themselves. For example, if we have a function definition $f\,h = h(\mathbf{event}\ \mathtt{a}; f\,h)$, the largest

priority encountered before f is recursively called in the body of f depends on the priorities encountered inside h, *and also* the state of $\mathcal{A}$ when h uses the argument **event** $\mathtt{a}; f$ (because the state after the $\mathtt{a}$ event depends on the previous state in general). We, therefore, use *intersection types* (a la Kobayashi and Ong's intersection types for HORS model checking [21]) to represent summary information on how each function traverses states of the automaton, and replicate each function and its arguments for each type. We thus formalize the translation as an intersection-type-based program transformation; related transformation techniques are found in [8,11,12,20,38].

Definition 10. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_I, \Omega)$ be a non-deterministic parity word automaton. Let q and m range over Q and the set $\mathit{codom}(\Omega)$ of priorities respectively. The set* $\mathbf{Types}_{\mathcal{A}}$ *of* intersection types*, ranged over by θ, is defined by:*

$$\theta ::= q \mid \rho \to \theta \qquad\qquad \rho ::= \mathtt{int} \mid \textstyle\bigwedge_{1 \le i \le k} (\theta_i, m_i)$$

We assume a certain total order $<$ on $\mathbf{Types}_{\mathcal{A}} \times \mathbb{N}$, and require that in $\bigwedge_{1 \le i \le k} (\theta_i, m_i)$, $(\theta_i, m_i) < (\theta_j, m_j)$ holds for each $i < j$.

We often write $(\theta_1, m_1) \wedge \cdots \wedge (\theta_k, m_k)$ for $\bigwedge_{1 \le i \le k} (\theta_i, m_i)$, and $\top$ when $k = 0$. Intuitively, the type q describes expressions of simple type $\star$, which may be evaluated when the automaton $\mathcal{A}$ is in the state q (here, we have in mind an execution of the *product* of a program and the automaton, where the latter takes events produced by the program and changes its states). The type $(\bigwedge_{1 \le i \le k} (\theta_i, m_i)) \to \theta$ describes functions that take an argument, use it according to types $\theta_1, \ldots, \theta_k$, and return a value of type θ. Furthermore, the part m_i describes that the argument may be used as a value of type θ_i only when the largest priority visited since the function is called is m_i. For example, given the automaton in Example 10, the function $\lambda x.(\mathbf{event}\ \mathtt{a}; x)$ may have types $(q_a, 0) \to q_a$ and $(q_a, 0) \to q_b$, because the body may be executed from state q_a or q_b (thus, the return type may be any of them), but x is used only when the automaton is in state q_a and the largest priority visited is 1. In contrast, $\lambda x.(\mathbf{event}\ \mathtt{b}; x)$ have types $(q_b, 1) \to q_a$ and $(q_b, 1) \to q_b$.

Using the intersection types above, we shall define a type-based transformation relation of the form $\Gamma \vdash_{\mathcal{A}} t : \theta \Rightarrow t'$, where t and t' are the source and target terms of the transformation, and Γ, called an *intersection type environment*, is a finite set of type bindings of the form $x : \mathtt{int}$ or $x : (\theta, m, m')$. We allow multiple type bindings for a variable x except for $x : \mathtt{int}$ (i.e. if $x : \mathtt{int} \in \Gamma$, then this must be the unique type binding for x in Γ). The binding $x : (\theta, m, m')$ means that x should be used as a value of type θ when the largest priority visited is m; m' is auxiliary information used to record the largest priority encountered so far.

The transformation relation $\Gamma \vdash_{\mathcal{A}} t : \theta \Rightarrow t'$ is inductively defined by the rules in Fig. 5. (For technical convenience, we have extended terms with λ-abstractions; they may occur only at top-level function definitions.) In the figure, $[k]$ denotes the set $\{i \in \mathbb{N} \mid 1 \le i \le k\}$. The operation $\Gamma \uparrow m$ used in the figure is defined by:

$$\Gamma \uparrow m = \{x : \mathtt{int} \mid x : \mathtt{int} \in \Gamma\} \cup \{x : (\theta, m_1, \mathbf{max}(m_2, m)) \mid x : (\theta, m_1, m_2) \in \Gamma\}$$

The operation is applied when the priority m is encountered, in which case the largest priority encountered is updated accordingly. The key rules are IT-Var, IT-Event, IT-App, and IT-Abs. In IT-Var, the variable x is replicated for each type; in the target of the translation, $x_{\theta,m}$ and $x_{\theta',m'}$ are treated as different variables if $(\theta, m) \neq (\theta', m')$. The rule IT-Event reflects the state change caused by the event a to the type and the type environment. Since the state change may be non-deterministic, we transform t for each of the next states $q_1, \ldots, q_n$, and combine the resulting terms with non-deterministic choice. The rule IT-App and IT-Abs replicates function arguments for each type. In addition, in IT-App, the operation $\Gamma \uparrow m_i$ reflects the fact that t_2 is used as a value of type θ_i after the priority m_i is encountered. The other rules just transform terms in a compositional manner. If target terms are ignored, the entire rules are close to those of Kobayashi and Ong's type system for HORS model checking [21].

$$\frac{}{\Gamma \vdash_{\mathcal{A}} () : q \Rightarrow ()} \text{(IT-Unit)}$$

$$\frac{}{\Gamma, x : \mathtt{int} \vdash_{\mathcal{A}} x : \mathtt{int} \Rightarrow x_{\mathtt{int}}} \text{(IT-VarInt)}$$

$$\frac{}{\Gamma, x : (\theta, m, m) \vdash_{\mathcal{A}} x : \theta \Rightarrow x_{\theta,m}} \text{(IT-Var)}$$

$$\frac{}{\Gamma \vdash_{\mathcal{A}} n : \mathtt{int} \Rightarrow n} \text{(IT-Int)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} t_1 : \mathtt{int} \Rightarrow t_1' \qquad \Gamma \vdash_{\mathcal{A}} t_2 : \mathtt{int} \Rightarrow t_2'}{\Gamma \vdash_{\mathcal{A}} t_1 \ \mathtt{op}\ t_2 : \mathtt{int} \Rightarrow t_1' \ \mathtt{op}\ t_2'} \text{(IT-Op)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathcal{A}} t_i : \mathtt{int} \Rightarrow t_i' \quad (\text{for each } i \in [k]) \\ \Gamma \vdash_{\mathcal{A}} t_{k+1} : q \Rightarrow t_{k+1}' \\ \Gamma \vdash_{\mathcal{A}} t_{k+2} : q \Rightarrow t_{k+2}' \\ \widetilde{t} = t_1, \ldots, t_k \qquad \widetilde{t'} = t_1', \ldots, t_k'\end{array}}{\begin{array}{l}\Gamma \vdash_{\mathcal{A}} \mathbf{if}\ p(\widetilde{t})\ \mathbf{then}\ t_{k+1}\ \mathbf{else}\ t_{k+2} : q \\ \quad \Rightarrow \mathbf{if}\ p(\widetilde{t'})\ \mathbf{then}\ t_{k+1}'\ \mathbf{else}\ t_{k+2}'\end{array}} \text{(IT-If)}$$

$$\frac{\begin{array}{c}\delta_A(q, a) = \{q_1, \ldots, q_k\} \\ \Gamma \uparrow \Omega_A(q_i) \vdash_{\mathcal{A}} t : q_i \Rightarrow t_i' \quad (\text{for each } i \in [k])\end{array}}{\Gamma \vdash_{\mathcal{A}} (\mathbf{event}\ a; t) : q \Rightarrow (\mathbf{event}\ a; t_1' \Box \cdots \Box t_k')} \text{(IT-Event)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} t_1 : q \Rightarrow t_1' \qquad \Gamma \vdash_{\mathcal{A}} t_2 : q \Rightarrow t_2'}{\Gamma \vdash_{\mathcal{A}} t_1 \Box t_2 : q \Rightarrow t_1' \Box t_2'} \text{(IT-NonDet)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} t_1 : \mathtt{int} \to \theta \Rightarrow t_1' \qquad \Gamma \vdash_{\mathcal{A}} t_2 : \mathtt{int} \Rightarrow t_2'}{\Gamma \vdash_{\mathcal{A}} t_1\ t_2 : \theta \Rightarrow t_1'\ t_2'} \text{(IT-AppInt)}$$

$$\frac{\Gamma \vdash_{\mathcal{A}} t_1 : \bigwedge_{1 \leq i \leq k} (\theta_i, m_i) \to \theta \Rightarrow t_1' \qquad \Gamma \uparrow m_i \vdash_{\mathcal{A}} t_2 : \theta_i \Rightarrow t_{2,i}' \ (\text{for each } i \in [k])}{\Gamma \vdash_{\mathcal{A}} t_1\ t_2 : \theta \Rightarrow t_1'\ t_{2,1}'\ \ldots\ t_{2,k}'} \text{(IT-App)}$$

$$\frac{\Gamma, x : \mathtt{int} \vdash_{\mathcal{A}} t : \theta \Rightarrow t' \qquad x \notin dom(\Gamma)}{\Gamma \vdash_{\mathcal{A}} \lambda x.t : \mathtt{int} \to \theta \Rightarrow \lambda x_{\mathtt{int}}.t'} \text{(IT-AbsInt)}$$

$$\frac{\Gamma \cup \{x : (\theta_i, m_i, 0) \mid i \in [k]\} \vdash_{\mathcal{A}} t : \theta' \Rightarrow t' \qquad x \notin dom(\Gamma)}{\begin{array}{l}\Gamma \vdash_{\mathcal{A}} \lambda x.t : \bigwedge_{1 \leq i \leq k} (\theta_i, m_i) \to \theta' \\ \quad \Rightarrow \lambda x_{\theta_1, m_1} \ldots x_{\theta_k, m_k}.t'\end{array}} \text{(IT-Abs)}$$

Fig. 5. Type-based transformation rules for terms

We now define the transformation for programs. A *top-level type environment* Ξ is a finite set of type bindings of the form $x : (\theta, m)$. Like intersection type environments, Ξ may have more than one binding for each variable. We write $\Xi \vdash_{\mathcal{A}} t : \theta$ to mean $\{x : (\theta, m, 0) \mid x : (\theta, m) \in \Xi\} \vdash_{\mathcal{A}} t : \theta$. For a set D of function definitions, we write $\Xi \vdash_{\mathcal{A}} D \Rightarrow D'$ if $dom(D') = \{ f_{\theta,m} \mid f : (\theta, m) \in \Xi \}$ and $\Xi \vdash_{\mathcal{A}} D(f) : \theta \Rightarrow D'(f_{\theta,m})$ for every $f : (\theta, m) \in \Xi$. For a program $P = (D, t)$, we

write $\Xi \vdash_{\mathcal{A}} P \Rightarrow (P', \Omega')$ if $P' = (D', t')$, $\Xi \vdash_{\mathcal{A}} D \Rightarrow D'$ and $\Xi \vdash_{\mathcal{A}} t : q_I \Rightarrow t'$, with $\Omega'(f_{\theta,m}) = m+1$ for each $f_{\theta,m} \in dom(D')$. We just write $\vdash_{\mathcal{A}} P \Rightarrow (P', \Omega')$ if $\Xi \vdash_{\mathcal{A}} P \Rightarrow (P', \Omega')$ holds for some Ξ.

Example 11. Consider the automaton $\mathcal{A}_{ab}$ in Example 10, and the program $P_2 = (D_2, f\,5)$ where D_2 consists of the following function definitions:

$$g\ k = (\textbf{event}\ \mathtt{a}; k) \Box (\textbf{event}\ \mathtt{b}; k),$$
$$f\ x = \textbf{if}\ x > 0\ \textbf{then}\ g\,(f(x-1))\ \textbf{else}\ (\textbf{event}\ \mathtt{b}; f\,5).$$

Let Ξ be: $\{g : ((q_a,0) \wedge (q_b,1) \to q_a, 0), g : ((q_a,0) \wedge (q_b,1) \to q_b, 0), f : (\mathtt{int} \to q_a, 0), f : (\mathtt{int} \to q_b, 1)\}$. Then, $\Xi \vdash_{\mathcal{A}} P_1 \Rightarrow ((D_2', f_{\mathtt{int}\to q_a,0}\,5), \Omega)$ where:

$$\begin{aligned}
D_2' = \{&g_{(q_a,0)\wedge(q_b,1)\to q_a,0}\ k_{q_a,0}\ k_{q_b,1} = t_g, \quad g_{(q_a,0)\wedge(q_b,1)\to q_b,0}\ k_{q_a,0}\ k_{q_b,1} = t_g,\\
&f_{\mathtt{int}\to q_a,0}\ x_{\mathtt{int}} = t_{f,q_a}, \quad f_{\mathtt{int}\to q_b,1}\ x_{\mathtt{int}} = t_{f,q_b}\}\\
t_g = (&\textbf{event}\ \mathtt{a}; k_{q_a,0}) \Box (\textbf{event}\ \mathtt{b}; k_{q_b,1}),\\
t_{f,q} = &\textbf{if}\ x_{\mathtt{int}} > 0\ \textbf{then}\\
&\quad g_{(q_a,0)\wedge(q_b,1)\to q,0}\,(f_{\mathtt{int}\to q_a,0}(x_{\mathtt{int}}-1))\,(f_{\mathtt{int}\to q_b,1}(x_{\mathtt{int}}-1))\\
&\textbf{else}\ (\textbf{event}\ \mathtt{b}; f_{\mathtt{int}\to q_b,1}\,5), \qquad\qquad (\text{for each } q \in \{q_a, q_b\})\\
\Omega = \{&g_{(q_a,0)\wedge(q_b,1)\to q_a,0} \mapsto 1, g_{(q_a,0)\wedge(q_b,1)\to q_b,0} \mapsto 1, f_{\mathtt{int}\to q_a,0} \mapsto 1, f_{\mathtt{int}\to q_b,1} \mapsto 2\}.
\end{aligned}$$

Notice that f, g, and the arguments of g have been duplicated. Furthermore, whenever $f_{\theta,m}$ is called, the largest priority that has been encountered since the last recursive call is m. For example, in the then-clause of $f_{\mathtt{int}\to q_a,0}$, $f_{\mathtt{int}\to q_b,1}(x-1)$ may be called through $g_{(q_a,0)\wedge(q_b,1)\to q_a,0}$. Since $g_{(q_a,0)\wedge(q_b,1)\to q_a,0}$ uses the second argument only after an event $\mathtt{b}$, the largest priority encountered is 1. This property is important for the correctness of our reduction.

The following theorems below claim that our reduction is sound and complete, and that there is an effective algorithm for the reduction: see [23] for proofs.

Theorem 5. *Let P be a program and $\mathcal{A}$ be a parity automaton. Suppose that $\Xi \vdash_{\mathcal{A}} P \Rightarrow (P', \Omega)$. Then* $\mathbf{InfTraces}(P) \cap \mathcal{L}(\mathcal{A}) = \emptyset$ *if and only if* $\models_{csa} (P', \Omega)$.

Theorem 6. *For every P and $\mathcal{A}$, one can effectively construct Ξ, P' and Ω such that $\Xi \vdash_{\mathcal{A}} P \Rightarrow (P', \Omega)$.*

The proof of Theorem 6 above also implies that the reduction from temporal property verification to call-sequence analysis can be performed in polynomial time. Combined with the reduction from call-sequence analysis to HFL model checking, we have thus obtained a polynomial-time reduction from the temporal verification problem $\mathbf{InfTraces}(P) \stackrel{?}{\subseteq} \mathcal{L}(\mathcal{A})$ to HFL model checking.

8 Related Work

As mentioned in Sect. 1, our reduction from program verification problems to HFL model checking problems has been partially inspired by the translation of

Kobayashi et al. [19] from HORS model checking to HFL model checking. As in their translation (and unlike in previous applications of HFL model checking [28, 42]), our translation switches the roles of properties and models (or programs) to be verified. Although a combination of their translation with Kobayashi's reduction from program verification to HORS model checking [17,18] yields an (indirect) translation from *finite-data* programs to pure HFL model checking problems, the combination does not work for infinite-data programs. In contrast, our translation is sound and complete even for infinite-data programs. Among the translations in Sects. 5, 6 and 7, the translation in Sect. 7.2 shares some similarity to their translation, in that functions and their arguments are replicated for each priority. The actual translations are however quite different; ours is type-directed and optimized for a given automaton, whereas their translation is not. This difference comes from the difference of the goals: the goal of [19] was to clarify the relationship between HORS and HFL, hence their translation was designed to be independent of an automaton. The proof of the correctness of our translation in Sect. 7 is much more involved due to the need for dealing with integers. Whilst the proof of [19] could reuse the type-based characterization of HORS model checking [21], we had to generalize arguments in both [19,21] to work on infinite-data programs.

Lange et al. [28] have shown that various process equivalence checking problems (such as bisimulation and trace equivalence) can be reduced to (pure) HFL model checking problems. The idea of their reduction is quite different from ours. They reduce processes to LTSs, whereas we reduce programs to HFL formulas.

Major approaches to automated or semi-automated higher-order program verification have been HORS model checking [17,18,22,27,31,33,43], (refinement) type systems [14,24,34–36,39,41,44], Horn clause solving [2,7], and their combinations. As already discussed in Sect. 1, compared with the HORS model checking approach, our new approach provides more uniform, streamlined methods. Whilst the HORS model checking approach is for fully automated verification, our approach enables various degrees of automation: after verification problems are automatically translated to $\mathrm{HFL}_{\mathbf{Z}}$ formulas, one can prove them (i) interactively using a proof assistant like Coq (see [23]), (ii) semi-automatically, by letting users provide hints for induction/co-induction and discharging the rest of proof obligations by (some extension of) an SMT solver, or (iii) fully automatically by recasting the techniques used in the HORS-based approach; for example, to deal with the ν-only fragment of $\mathrm{HFL}_{\mathbf{Z}}$, we can reuse the technique of predicate abstraction [22]. For a more technical comparison between the HORS-based approach and our HFL-based approach, see [23].

As for type-based approaches [14,24,34–36,39,41,44], most of the refinement type systems are (i) restricted to safety properties, and/or (ii) incomplete. A notable exception is the recent work of Unno et al. [40], which provides a relatively complete type system for the classes of properties discussed in Sect. 5. Our approach deals with a wider class of properties (cf. Sects. 6 and 7). Their "relative completeness" property relies on Godel coding of functions, which cannot be exploited in practice.

The reductions from program verification to Horn clause solving have recently been advocated [2–4] or used [34,39] (via refinement type inference problems) by a number of researchers. Since Horn clauses can be expressed in a fragment of HFL without modal operators, fixpoint alternations (between ν and μ), and higher-order predicates, our reductions to HFL model checking may be viewed as extensions of those approaches. Higher-order predicates and fixpoints over them allowed us to provide sound and complete characterizations of properties of higher-order programs for a wider class of properties. Bjørner et al. [4] proposed an alternative approach to obtaining a complete characterization of safety properties, which defunctionalizes higher-order programs by using algebraic data types and then reduces the problems to (first-order) Horn clauses. A disadvantage of that approach is that control flow information of higher-order programs is also encoded into algebraic data types; hence even for finite-data higher-order programs, the Horn clauses obtained by the reduction belong to an undecidable fragment. In contrast, our reductions yield pure HFL model checking problems for finite-data programs. Burn et al. [7] have recently advocated the use of *higher-order* (constrained) Horn clauses for verification of safety properties (i.e., which correspond to the negation of may-reachability properties discussed in Sect. 5.1 of the present paper) of higher-order programs. They interpret recursion using the least fixpoint semantics, so their higher-order Horn clauses roughly corresponds to a fragment of the $\mathrm{HFL}_{\mathbf{Z}}$ without modal operators and fixpoint alternations. They have not shown a general, concrete reduction from safety property verification to higher-order Horn clause solving.

The characterization of the reachability problems in Sect. 5 in terms of formulas without modal operators is a reminiscent of predicate transformers [9,13] used for computing the weakest preconditions of imperative programs. In particular, [5] and [13] respectively used least fixpoints to express weakest preconditions for while-loops and recursions.

9 Conclusion

We have shown that various verification problems for higher-order functional programs can be naturally reduced to (extended) HFL model checking problems. In all the reductions, a program is mapped to an HFL formula expressing the property that the behavior of the program is correct. For developing verification tools for higher-order functional programs, our reductions allow us to focus on the development of (automated or semi-automated) $\mathrm{HFL}_{\mathbf{Z}}$ model checking tools (or, even more simply, theorem provers for $\mathrm{HFL}_{\mathbf{Z}}$ without modal operators, as the reductions of Sects. 5 and 7 yield HFL formulas without modal operators). To this end, we have developed a prototype model checker for pure HFL (without integers), which will be reported in a separate paper. Work is under way to develop $\mathrm{HFL}_{\mathbf{Z}}$ model checkers by recasting the techniques [22,26,27,43] developed for the HORS-based approach, which, together with the reductions presented in this paper, would yield fully automated verification tools. We have also started building a Coq library for interactively proving $\mathrm{HFL}_{\mathbf{Z}}$ formulas,

as briefly discussed in [23]. As a final remark, although one may fear that our reductions may map program verification problems to "harder" problems due to the expressive power of $\mathrm{HFL}_{\mathbf{Z}}$, it is actually not the case at least for the classes of problems in Sects. 5 and 6, which use the only alternation-free fragment of $\mathrm{HFL}_{\mathbf{Z}}$. The model checking problems for μ-only or ν-only $\mathrm{HFL}_{\mathbf{Z}}$ are semi-decidable and co-semi-decidable respectively, like the source verification problems of may/must-reachability and their negations of closed programs.

Acknowledgment. We would like to thank anonymous referees for useful comments. This work was supported by JSPS KAKENHI Grant Number JP15H05706 and JP16K16004.

References

1. Axelsson, R., Lange, M., Somla, R.: The complexity of model checking higher-order fixpoint logic. Logical Methods Comput. Sci. **3**(2), 1–33 (2007)
2. 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
3. Bjørner, N., McMillan, K.L., Rybalchenko, A.: Program verification as satisfiability modulo theories. In: SMT 2012, EPiC Series in Computing, vol. 20, pp. 3–11. EasyChair (2012)
4. Bjørner, N., McMillan, K.L., Rybalchenko, A.: Higher-order program verification as satisfiability modulo theories with algebraic data-types. CoRR, abs/1306.5264 (2013)
5. Blass, A., Gurevich, Y.: Existential fixed-point logic. In: Börger, E. (ed.) Computation Theory and Logic. LNCS, vol. 270, pp. 20–36. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18170-9_151
6. Blume, M., Acar, U.A., Chae, W.: Exception handlers as extensible cases. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 273–289. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_20
7. Burn, T.C., Ong, C.L., Ramsay, S.J.: Higher-order constrained horn clauses for verification. PACMPL **2**(POPL), 11:1–11:28 (2018)
8. Carayol, A., Serre, O.: Collapsible pushdown automata and labeled recursion schemes: equivalence, safety and effective selection. In: LICS 2012, pp. 165–174. IEEE (2012)
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
10. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata Logics, and Infinite Games: A Guide to Current Research. LNCS, vol. 2500. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36387-4
11. Grellois, C., Melliès, P.: Relational semantics of linear logic and higher-order model checking. In: Proceedings of CSL 2015, LIPIcs, vol. 41, pp. 260–276 (2015)
12. Haddad, A.: Model checking and functional program transformations. In: Proceedings of FSTTCS 2013, LIPIcs, vol. 24, pp. 115–126 (2013)
13. Hesselink, W.H.: Predicate-transformer semantics of general recursion. Acta Inf. **26**(4), 309–332 (1989)

14. Hofmann, M., Chen, W.: Abstract interpretation from Büchi automata. In: Proceedings of CSL-LICS 2014, pp. 51:1–51:10. ACM (2014)
15. Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Trans. Prog. Lang. Syst. **27**(2), 264–313 (2005)
16. Knapik, T., Niwiński, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: Nielsen, M., Engberg, U. (eds.) FoSSaCS 2002. LNCS, vol. 2303, pp. 205–222. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45931-6_15
17. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Proceedings of POPL, pp. 416–428. ACM Press (2009)
18. Kobayashi, N.: Model checking higher-order programs. J. ACM **60**(3), 1–62 (2013)
19. Kobayashi, N., Lozes, É., Bruse, F.: On the relationship between higher-order recursion schemes and higher-order fixpoint logic. In: Proceedings of POPL 2017, pp. 246–259 (2017)
20. Kobayashi, N., Matsuda, K., Shinohara, A., Yaguchi, K.: Functional programs as compressed data. High.-Order Symbolic Comput. **25**(1), 39–84 (2013)
21. Kobayashi, N., Ong, C.H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of LICS 2009, pp. 179–188 (2009)
22. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of PLDI, pp. 222–233. ACM Press (2011)
23. Kobayashi, N., Tsukada, T., Watanabe, K.: Higher-order program verification via HFL model checking. CoRR abs/1710.08614 (2017). http://arxiv.org/abs/1710.08614
24. Koskinen, E., Terauchi, T.: Local temporal reasoning. In: Proceedings of CSL-LICS 2014, pp. 59:1–59:10. ACM (2014)
25. Kozen, D.: Results on the propositional μ-calculus. Theor. Comput. Sci. **27**, 333–354 (1983)
26. Kuwahara, T., Sato, R., Unno, H., Kobayashi, N.: Predicate abstraction and CEGAR for disproving termination of higher-order functional programs. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 287–303. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_17
27. Kuwahara, T., Terauchi, T., Unno, H., Kobayashi, N.: Automatic termination verification for higher-order functional programs. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 392–411. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_21
28. Lange, M., Lozes, É., Guzmán, M.V.: Model-checking process equivalences. Theor. Comput. Sci. **560**, 326–347 (2014)
29. Ledesma-Garza, R., Rybalchenko, A.: Binary reachability analysis of higher order functional programs. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 388–404. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33125-1_26
30. Lozes, É.: A type-directed negation elimination. In: Proceedings FICS 2015, EPTCS, vol. 191, pp. 132–142 (2015)
31. Murase, A., Terauchi, T., Kobayashi, N., Sato, R., Unno, H.: Temporal verification of higher-order functional programs. In: Proceedings of POPL 2016, pp. 57–68 (2016)
32. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS 2006, pp. 81–90. IEEE Computer Society Press (2006)
33. Ong, C.H.L., Ramsay, S.: Verifying higher-order programs with pattern-matching algebraic data types. In: Proceedings of POPL, pp. 587–598. ACM Press (2011)

34. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. PLDI **2008**, 159–169 (2008)
35. Skalka, C., Smith, S.F., Horn, D.V.: Types and trace effects of higher order programs. J. Funct. Program. **18**(2), 179–249 (2008)
36. Terauchi, T.: Dependent types from counterexamples. In: Proceedings of POPL, pp. 119–130. ACM (2010)
37. Tobita, Y., Tsukada, T., Kobayashi, N.: Exact flow analysis by higher-order model checking. In: Schrijvers, T., Thiemann, P. (eds.) FLOPS 2012. LNCS, vol. 7294, pp. 275–289. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29822-6_22
38. Tsukada, T., Ong, C.L.: Compositional higher-order model checking via ω-regular games over Böhm trees. In: Proceedings of CSL-LICS 2014, pp. 78:1–78:10. ACM (2014)
39. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: PPDP 2009, pp. 277–288. ACM (2009)
40. Unno, H., Satake, Y., Terauchi, T.: Relatively complete refinement type system for verification of higher-order non-deterministic programs. PACMPL **2**(POPL), 12:01–12:29 (2018)
41. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: POPL 2013. pp. 75–86. ACM (2013)
42. Viswanathan, M., Viswanathan, R.: A higher order modal fixed point logic. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 512–528. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_33
43. Watanabe, K., Sato, R., Tsukada, T., Kobayashi, N.: Automatically disproving fair termination of higher-order functional programs. In: Proceedings of ICFP 2016, pp. 243–255. ACM (2016)
44. Zhu, H., Nori, A.V., Jagannathan, S.: Learning refinement types. In: Proceedings of ICFP 2015, pp. 400–411. ACM (2015)