



Quantitative Analysis of Smart Contracts

Krishnendu Chatterjee¹, Amir Kafshdar Goharshady^{1(✉)}, and Yaron Velner²

¹ IST Austria (Institute of Science and Technology Austria), Klosterneuburg, Austria
`{krishnendu.chatterjee, amir.goharshady}@ist.ac.at`
² Hebrew University of Jerusalem, Jerusalem, Israel
`yaron.welner@mail.huji.ac.il`

Abstract. Smart contracts are computer programs that are executed by a network of mutually distrusting agents, without the need of an external trusted authority. Smart contracts handle and transfer assets of considerable value (in the form of crypto-currency like Bitcoin). Hence, it is crucial that their implementation is bug-free. We identify the utility (or expected payoff) of interacting with such smart contracts as the basic and canonical quantitative property for such contracts. We present a framework for such quantitative analysis of smart contracts. Such a formal framework poses new and novel research challenges in programming languages, as it requires modeling of game-theoretic aspects to analyze incentives for deviation from honest behavior and modeling utilities which are not specified as standard temporal properties such as safety and termination. While game-theoretic incentives have been analyzed in the security community, their analysis has been restricted to the very special case of stateless games. However, to analyze smart contracts, stateful analysis is required as it must account for the different program states of the protocol. Our main contributions are as follows: we present (i) a simplified programming language for smart contracts; (ii) an automatic translation of the programs to state-based games; (iii) an abstraction-refinement approach to solve such games; and (iv) experimental results on real-world-inspired smart contracts.

1 Introduction

In this work we present a quantitative stateful game-theoretic framework for formal analysis of smart-contracts.

Smart Contracts. Hundreds of crypto-currencies are in use today, and investments in them are increasing steadily [24]. These currencies are not controlled by any central authority like governments or banks, instead they are governed by the *blockchain* protocol, which dictates the rules and determines the outcomes, e.g., the validity of money transactions and account balances. Blockchain was initially used for peer-to-peer Bitcoin payments [43], but recently it is also used for running programs (called smart contracts). A *smart contract* is a program that runs on the blockchain, which enforces its correct execution (i.e., that

A longer version of this article is available in [19].

© The Author(s) 2018

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 739–767, 2018.

https://doi.org/10.1007/978-3-319-89884-1_26

it is running as originally programmed). This is done by encoding semantics in crypto-currency transactions. For example, Bitcoin transaction scripts allow users to specify conditions, or contracts, which the transactions must satisfy prior to acceptance. Transaction scripts can encode many useful functions, such as validating that a payer owns a coin she is spending or enforcing rules for multi-party transactions. The Ethereum crypto-currency [16] allows arbitrary stateful Turing-complete conditions over the transactions which gives rise to smart contracts that can implement a wide range of applications, such as financial instruments (e.g., financial derivatives or wills) or autonomous governance applications (e.g., voting systems). The protocols are globally specified and their implementation is decentralized. Therefore, there is no central authority and they are immutable. Hence, the economic consequences of bugs in a smart contract cannot be reverted.

Types of Bugs. There are two types of bugs with monetary consequences:

1. *Coding errors.* Similar to standard programs, bugs could arise from coding mistakes. At one reported case [33], mistakenly replacing `+=` operation with `==` enabled loss of tokens that were backed by \$800,000 of investment.
2. *Dishonest interaction incentives.* Smart contracts do not fully dictate the behavior of participants. They only specify the outcome (e.g., penalty or rewards) of the behaviors. Hence, a second source for bugs is the high level *interaction aspects* that could give a participant unfair advantage and incentive for dishonest behavior. For example, a naive design of rock-paper-scissors game [29] allows playing sequentially, rather than concurrently, and gives advantage to the second player who can see the opponent's move.

DAO Attack: Interaction of Two Types of Bugs. Quite interestingly a coding bug can incentivize dishonest behavior as in the famous DAO attack [48]. The Decentralized Autonomous Organization (DAO) [38] is an Ethereum smart contract [51]. The contract consists of investor-directed venture capital fund. On June 17, 2016 an attacker exploited a bug in the contract to extract \$80 million [48]. Intuitively, the root cause was that the contract allowed users to first get hold of their funds, and only then updated their balance records while a semantic detail allowed the attacker to withdraw multiple times before the update.

Necessity of Formal Framework. Since bugs in smart contracts have direct economic consequences and are irreversible, they have the same status as safety-critical errors for programs and reactive systems and must be detected before deployment. Moreover, smart contracts are deployed rapidly. There are over a million smart contracts in Ethereum, holding over 15 billion dollars at the time of writing [31]. It is impossible for security researchers to analyze all of them, and lack of automated tools for programmers makes them error prone. Hence, a formal analysis framework for smart contract bugs is of great importance.

Utility Analysis. In verification of programs, specifying objectives is non-trivial and a key goal is to consider specification-less verification, where basic properties are considered canonical. For example, termination is a basic property in

program analysis; and data-race freedom or serializability are basic properties in concurrency. Given these properties, models are verified wrt them without considering any other specification. For smart contracts, describing the correct specification that prevents dishonest behavior is more challenging due to the presence of game-like interactions. We propose to consider the expected user utility (or payoff) that is guaranteed even in presence of adversarial behavior of other agents as a canonical property. Considering malicious adversaries is standard in game theory. For example, the expected utility of a fair lottery is 0. An analysis reporting a different utility signifies a bug.

New Research Challenges. Coding bugs are detected by classic verification, program analysis, and model checking tools [23,39]. However, a formal framework for incentivization bugs presents a new research challenge for the programming language community. Their analysis must overcome two obstacles: (a) the framework will have to handle game-theoretic aspects to model interactions and incentives for dishonest behavior; and (b) it will have to handle properties that cannot be deduced from standard temporal properties such as safety or termination, but require analysis of monetary gains (i.e., quantitative properties).

While game-theoretic incentives are widely analyzed by the security community (e.g., see [13]), their analysis is typically restricted to the very special case of one-shot games that do not consider different states of the program, and thus the consequences of decisions on the next state of the program are ignored. In addition their analysis is typically ad-hoc and stems from brainstorming and special techniques. This could work when very few protocols existed (e.g., when bitcoin first emerged) and deep thought was put into making them elegant and analyzable. However, the fast deployment of smart contracts makes it crucial to automate the process and make it accessible to programmers.

Our Contribution. In this work we present a formal framework for quantitative analysis of utilities in smart contracts. Our contributions are as follows:

1. We present a simplified (loop-free) programming language that allows game-theoretic interactions. We show that many classical smart contracts can be easily described in our language, and conversely, a smart contract programmed in our language can be easily translated to Solidity [30], which is the most popular Ethereum smart contract language.
2. The underlying mathematical model for our language is stateful concurrent games. We automatically translate programs in our language to such games.
3. The key challenge to analyze such game models automatically is to tackle the state-space explosion. While several abstraction techniques have been considered for programs [14,35,45], they do not work for game-theoretic models with quantitative objectives. We present an approach based on interval-abstraction for reducing the states, establish soundness of our abstraction, and present a refinement process. This is our core technical contribution.
4. We present experimental results on several classic real-world smart contracts. We show that our approach can handle contracts that otherwise give rise to games with up to 10^{23} states. While special cases of concurrent games

(namely, turn-based games) have been studied in verification and reactive synthesis, there are no practical methods to solve general concurrent quantitative games. To the best of our knowledge, there are no tools to solve quantitative concurrent games other than academic examples of few states, and we present the first practical method to solve quantitative concurrent games that scales to real-world smart contract analysis.

In summary, our contributions range from (i) modeling of smart contracts as state-based games, to (ii) an abstraction-refinement approach to solve such games, to (iii) experimental results on real-world smart contracts.

2 Background on Ethereum Smart Contracts

2.1 Programmable Smart Contracts

Ethereum [16] is a decentralized virtual machine, which runs programs called contracts. Contracts are written in a Turing-complete bytecode language, called Ethereum Virtual Machine (EVM) bytecode [53]. A contract is invoked by calling one of its functions, where each function is defined by a sequence of instructions. The contract maintains a persistent internal state and can receive (transfer) currency from (to) users and other contracts. Users send transactions to the Ethereum network to invoke functions. Each transaction may contain input parameters for the contract and an associated monetary amount, possibly 0, which is transferred from the user to the contract.

Upon receiving a transaction, the contract collects the money sent to it, executes a function according to input parameters, and updates its internal state. All transactions are recorded on a decentralized ledger, called blockchain. A sequence of transactions that begins from the creation of the network uniquely determines the state of each contract and balances of users and contracts. The blockchain does not rely on a trusted central authority, rather, each transaction is processed by a large network of mutually untrusted peers called miners. Users constantly broadcast transactions to the network. Miners add transactions to the blockchain via a proof-of-work consensus protocol [43].

Subtleties. In this work, for simplicity, we ignore some details in the underlying protocol of Ethereum smart contract. We briefly describe these details below:

- *Transaction fees.* In exchange for including her transactions in the blockchain, a user pays transaction fees to the miners, proportionally to the execution time of her transaction. This fact could slightly affect the monetary analysis of the user gain, but could also introduce bugs in a program, as there is a bound on execution time that cannot be exceeded. Hence, it is possible that some functions could never be called, or even worse, a user could actively give input parameters that would prevent other users from invoking a certain function.

- *Recursive invocation of contracts.* A contract function could invoke a function in another contract, which in turn can have a call to the original contract. The underlying Ethereum semantic in recursive invocation was the root cause for the notorious DAO hack [27].
- *Behavior of the miners.* Previous works have suggested that smart contracts could be implemented to encourage miners to deviate from their honest behavior [50]. This could in theory introduce bugs into a contract, e.g., a contract might give unfair advantage for a user who is a big miner.

2.2 Tokens and User Utility

A user's utility is determined by the Ether she spends and receives, but could also be affected by the state of the contract. Most notably, smart contracts are used to issue *tokens*, which can be viewed as a stake in a company or an organization, in return to an Ether (or tokens) investment (see an example in Fig. 1). These tokens are *transferable* among users and are traded in exchanges in return to Ether, Bitcoin and Fiat money. At the time of writing, smart contracts instantiate tokens worth billions of dollars [32]. Hence, gaining or losing tokens has clear utility for the user. At a larger scope, user utility could also be affected by more abstract storage changes. Some users would be willing to pay to have a contract declare them as Kings of Ether [4], while others could gain from registering their domain name in a smart contract storage [40]. In the examples provided in this work we mainly focus on utility that arises from Ether, tokens and the like. However, our approach is general and can model any form of utility by introducing auxiliary utility variables and definitions.

```

1 contract Token {
2     mapping(address=>uint) balances;
3     function buy() payable {
4         balances[msg.sender] += msg.value;
5     }
6     function transfer( address to, uint amount ) {
7         if(balances[msg.sender]>=amount) {
8             balances[msg.sender] -= amount;
9             balances[to] += amount;
10        }
11    }
12 }

```

Fig. 1. Token contract example.

3 Programming Language for Smart Contracts

In this section we present our programming language for smart contracts that supports concurrent interactions between parties. A party denotes an agent that decides to interact with the contract. A contract is a tuple $C = (N, I, M, R, X_0, F, T)$ where $X := N \cup I \cup M$ is a set of variables, R describes the range of values that can be stored in each variable, X_0 is the initial values stored in variables, F is a list of functions and T describes for each function, the time segment in which it can be invoked. We now formalize these concepts.

Variables. There are three distinct and disjoint types of variables in X :

- N contains “numeric” variables that can store a single integer.
- I contains “identification” (“id”) variables capable of pointing to a party in the contract by her address or storing NULL. The notion of ids is quite flexible in our approach: The only dependence on ids is that they should be distinct and an id should not act on behalf of another id. We simply use different integers to denote distinct ids and assume that a “faking of identity” does not happen. In Ethereum this is achieved by digital signatures.
- M is the set of “mapping” variables. Each $m \in M$ maps parties to integers.

Bounds and Initial Values. The tuple $R = (\underline{R}, \overline{R})$ where $\underline{R}, \overline{R} : N \cup M \rightarrow \mathbb{Z}$ represent lower and upper bounds for integer values that can be stored in a variable. For example, if $n \in N$, then n can only store integers between $\underline{R}(n)$ and $\overline{R}(n)$. Similarly, if $m \in M$ is a mapping and $i \in I$ stores an address to a party in the contract, then $m[i]$ can save integers between $\underline{R}(m)$ and $\overline{R}(m)$. The function $X_0 : X \rightarrow \mathbb{Z} \cup \{\text{NULL}\}$ assigns an initial value to every variable. The assigned value is an integer in case of numeric and mapping variables, i.e., a mapping variable maps everything to its initial value by default. Id variables can either be initialized by NULL or an id used by one of the parties.

Functions and Timing. The sequence $F = \langle f_1, f_2, \dots, f_n \rangle$ is a list of functions and $T = (\underline{T}, \overline{T})$, where $\underline{T}, \overline{T} : F \rightarrow \mathbb{N}$. The function f_i can only be invoked in time-frame $T(f_i) = [\underline{T}(f_i), \overline{T}(f_i)]$. The contract uses a global clock, for example the current block number in the blockchain, to keep track of time.

Note that we consider a single contract, and interaction between multiple contracts is a subject of future work.

3.1 Syntax

We provide a simple overview of our contract programming language. Our language is syntactically similar to Solidity [30], which is a widely used language for writing Ethereum contracts. A translation mechanism for different aspects is discussed in [19]. An example contract, modeling a game of rock-paper-scissors, is given in Fig. 2. Here, a party, called **issuer** has issued the contract and taken the role of **Alice**. Any other party can join the contract by registering as **Bob** and then playing rock-paper-scissors. To demonstrate our language, we use a bidding mechanism.

Declaration of Variables. The program begins by declaring variables¹, their type, name, range and initial value. For example, **Bids** is a map variable that assigns a value between 0 and 100 to every id. This value is initially 0. Line numbers (labels) are defined in Sect. 3.2 below and are not part of the syntax.

Declaration of Functions. After the variables, the functions are defined one-by-one. Each function begins with the keyword **function** followed by its name and

¹ For simplicity, we demonstrate our method with global variables only. However, the method is applicable to general variables as long as their ranges are well-defined at each point of the program.

```

(0) contract RPS {
map Bids[0, 100] = 0;
id Alice = issuer;
id Bob = null;
numeric played[0,1] = 0;
numeric AliceWon[0,1] = 0;
numeric BobWon[0,1] = 0;
numeric bid[0, 100] = 0;
numeric AlicesMove[0,3] = 0;
numeric BobsMove[0,3] = 0;
//0 denotes no choice,
//1 rock, 2 paper,
//3 scissors

(1) function registerBob[1,10]
    (payable bid : caller) {
(2)     if(Bob==null) {
(3)         Bob = caller;
(4)         Bids[Bob]=bid;
        }
        else{
(5)             payout(caller, bid);
        }
(6) }
(7) function play[11, 15]
    (AlicesMove:Alice = 0,
     BobsMove:Bob = 0,
     payable Bids[Alice]: Alice){
(8)     if(played==1)
(9)         return;
        else
(10)            played = 1;

(11)     if(BobsMove==0 and AlicesMove!=0)
(12)         AliceWon = 1;
(13)     else if(AlicesMove==0 and BobsMove!=0)
(14)         BobWon = 1;
(15)     else if(AlicesMove==0 and BobsMove==0)
        {
(16)         AliceWon = 0;
(17)         BobWon = 0;
        }
(18)     else if(AlicesMove==BobsMove+1 or
                AlicesMove==BobsMove-2)
(19)         AliceWon = 1;
        else
(20)             BobWon = 1;
(21) }
(22) function getReward[16,20]() {
(23)     if(caller==Alice and AliceWon==1
        or caller==Bob and BobWon==1)
        {
(24)         payout(caller,
                    Bids[Alice] + Bids[Bob]);
(25)         Bids[Alice] = 0;
(26)         Bids[Bob] = 0;
        }
(27) }
}

```

Fig. 2. A rock-paper-scissors contract.

the time interval in which it can be called by parties. Then comes a list of input parameters. Each parameter is of the form **variable : party** which means that the designated party can choose a value for that variable. The chosen value is required to be in the range specified for that variable. The keyword **caller** denotes the party that has invoked this function and **payable** signifies that the party should not only decide a value, but must also pay the amount she decides. For example, **registerBob** can be called in any time between 1 and 10 by any of the parties. At each such invocation the party that has called this function must pay some amount which will be saved in the variable **bid**. After the decisions and payments are done, the contract proceeds with executing the function.

Types of Functions. There are essentially two types of functions, depending on their parameters. *One-party functions*, such as **registerBob** and **getReward** require parameters from **caller** only, while *multi-party functions*, such as **play** ask several, potentially different, parties for input. In this case all parties provide their input decisions and payments concurrently and without being aware of the choices made by other parties, also a default value is specified for every decision in case a relevant party does not take part.

Summary. Putting everything together, in the contract specified in Fig. 2, any party can claim the role of Bob between time 1 and time 10 by paying a bid to the contract, if the role is not already occupied. Then at time 11 one of the

parties calls `play` and both parties have until time 15 to decide which choice (rock, paper, scissors or none) they want to make. Then the winner can call `getReward` and collect her prize.

3.2 Semantics

In this section we present the details of the semantics. In our programming language there are several key aspects which are non-standard in programming languages, such as the notion of time progress, concurrency, and interactions of several parties. Hence we present a detailed description of the semantics. We start with the requirements.

Requirements. In order for a contract to be considered valid, other than following the syntax rules, a few more requirements must be met, which are as follows:

- We assume that no division by zero or similar undefined behavior happens.
- To have a well-defined message passing, we also assume that no multi-party function has an associated time interval intersecting that of another function.
- Finally, for each non-id variable v , it must hold that $\underline{R}(v) \leq X_0(v) \leq \overline{R}(v)$ and similarly, for every function f_i , we must have $\underline{T}(f_i) < \overline{T}(f_i)$.

Overview of Time Progress. Initially, the time is 0. Let F_t be the set of functions executable at time t , i.e., $F_t = \{f_i \in F \mid t \in T(f_i)\}$, then F_t is either empty or contains one or more one-party functions or consists of a single multi-party function. We consider the following cases:

- *F_t empty.* If F_t is empty, then nothing can happen until the clock ticks.
- *Execution of one-party functions.* If F_t contains one or more one-party functions, then each of the parties can call any subset of these functions at time t . If there are several calls at the same time, the contract might run them in any order. While a function call is being executed, all parties are able to see the full state of the contract, and can issue new calls. When there are no more requests for function calls, the clock ticks and the time is increased to $t + 1$. When a call is being executed and is at the beginning part of the function, its caller can send messages or payments to the contract. Values of these messages and payments will then be saved in designated variables and the execution continues. If the caller fails to make a payment or specify a value for a decision variable or if her specified values/payments are not in the range of their corresponding variables, i.e. they are too small or too big, the call gets canceled and the contract reverts any changes to variables due to the call and continues as if this call had never happened.
- *Execution of multi-party functions.* If F_t contains a single multi-party function f_i and $t < \overline{T}(f_i)$, then any party can send messages and payments to the contract to specify values for variables that are designated to be paid or decided by her. These choices are hidden and cannot be observed by other participants. She can also change her decisions as many times as she sees fit.

The clock ticks when there are no more valid requests for setting a value for a variable or making a payment. This continues until we reach time $\bar{T}(f_i)$. At this time parties can no longer change their choices and the choices become visible to everyone. The contract proceeds with execution of the function. If a party fails to make a payment/decision or if NULL is asked to make a payment or a decision, default behavior will be enforced. Default value for payments is 0 and default behavior for other variables is defined as part of the syntax. For example, in function `play` of Fig. 2, if a party does not choose, a default value of 0 is enforced and given the rest of this function, this will lead to a definite loss.

Given the notion of time progress we proceed to formalize the notion of “runs” of the contract. This requires the notion of labels, control-flow graphs, valuations, and states, which we describe below.

Labels. Starting from 0, we give the contract, beginning and end points of every function, and every command a label. The labels are given in order of appearance. As an example, see the labels in parentheses in Fig. 2.

Entry and Exit Labels. We denote the first (beginning point) label in a function f_i by \square_i and its last (end point) label by \blacksquare_i .

Control Flow Graphs (CFGs). We define the control flow graph CFG_i of the function f_i in the standard manner, i.e. $CFG_i = (V, E)$, where there is a vertex corresponding to every labeled entity inside f_i . Each edge $e \in E$ has a condition $cond(e)$ which is a boolean expression that must be true when traversing that edge. For more details see [19].

Valuations. A valuation is a function val , assigning a value to every variable. Values for numeric variables must be integers in their range, values for identity variables can be party ids or NULL and a value assigned to a map variable m must be a function $val(m)$ such that for each identity i , we have $\underline{R}(m) \leq val(m)(i) \leq \bar{R}(m)$. Given a valuation, we extend it to expressions containing mathematical operations in the straight-forward manner.

States. A state of the contract is a tuple $s = (t, b, l, val, c)$, where t is a time stamp, $b \in \mathbb{N} \cup \{0\}$ is the current balance of the contract, i.e., the total amount of payment to the contract minus the total amount of payouts, l is a label (that is being executed), val assigns values to variables and $c \in P \cup \{\perp\}$, is the caller of the current function. $c = \perp$ corresponds to the case where the caller is undefined, e.g., when no function is being executed. We use S to denote the set of all states that can appear in a run of the contract as defined below.

Runs. A run ρ of the contract is a finite sequence $\{\rho_j = (t_j, b_j, l_j, val_j, c_j)\}_{j=0}^r$ of states, starting from $(0, 0, 0, X_0, \perp)$, that follows all rules of the contract and ends in a state with time-stamp $t_r > \max_{f_i} \bar{T}(f_i)$. These rules must be followed when switching to a new state in a run:

- The clock can only tick when there are no valid pending requests for running a one-party function or deciding or paying in multi-party functions.

- Transitions that happen when the contract is executing a function must follow its control flow graph and update the valuation correctly.
- No variable can contain an out-of-bounds value. If an overflow or underflow happens, the closest possible value will be saved. This rule also ensures that the contract will not create new money, given that paying more than the current balance of the contract results in an underflow.
- Each party can call any set of the functions at any time.

Remark 1. Note that in our semantics each function body completes its execution in a single tick of the clock. However, ticks might contain more than one function call and execution.

Run Prefixes. We use H to mean the set of all prefixes of runs and denote the last state in $\eta \in H$ by $\text{end}(\eta)$. A run prefix η' is an extension of η if it can be obtained by adding one state to the end of η .

Probability Distributions. Given a finite set \mathcal{X} , a probability distribution on \mathcal{X} is a function $\delta : \mathcal{X} \rightarrow [0, 1]$ such that $\sum_{x \in \mathcal{X}} \delta(x) = 1$. Given such a distribution, its support, $\text{Supp}(\delta)$, is the set of all $x \in \mathcal{X}$ such that $\delta(x) > 0$. We denote the set of all probability distributions on \mathcal{X} by $\Delta(\mathcal{X})$.

Typically for programs it suffices to define runs for the semantics. However, given that there are several parties in contracts, their semantics depends on the possible choices of the parties. Hence we need to define policies for parties, and such policies will define probability distribution over runs, which constitute the semantics for contracts. To define policies we first define moves.

Moves. We use \mathcal{M} for the set of all moves. The moves that can be taken by parties in a contract can be summarized as follows:

- Calling a function f_i , we denote this by $\text{call}(f_i)$.
- Making a payment whose amount, y is saved in x , we denote this by $\text{pay}(x, y)$.
- Deciding the value of x to be y , we denote this by $\text{decide}(x, y)$.
- Doing none of the above, we denote this by \boxtimes .

Permitted Moves. We define $P_i : S \rightarrow \mathcal{M}$, so that $P_i(s)$ is the set of permitted moves for the party with identity i if the contract is in state $s = (t, b, l, \text{val}, p_j)$. It is formally defined as follows:

- If f_k is a function that can be called at state s , then $\text{call}(f_k) \in P_i(s)$.
- If $l = \square_q$ is the first label of a function f_q and x is a variable that can be decided by i at the beginning of the function f_q , then $\text{decide}(x, y) \in P_i(s)$ for all permissible values of y . Similarly if x can be paid by i , $\text{pay}(x, y) \in P_i(s)$.
- $\boxtimes \in P_i(s)$.

Policies and Randomized Policies. A policy π_i for party i is a function $\pi_i : H \rightarrow A$, such that for every $\eta \in H$, $\pi_i(\eta) \in P_i(\text{end}(\eta))$. Intuitively, a policy is a way of deciding what move to use next, given the current run prefix. A policy profile

$\pi = (\pi_i)$ is a sequence assigning one policy to each party i . The policy profile π defines a unique run ρ^π of the contract which is obtained when parties choose their moves according to π . A randomized policy ξ_i for party i is a function $\xi_i : H \rightarrow \Delta(\mathcal{M})$, such that $\text{Supp}(\xi_i(s)) \subseteq P_i(s)$. A randomized policy assigns a probability distribution over all possible moves for party i given the current run prefix of the contract, then the party can follow it by choosing a move randomly according to the distribution. We use Ξ to denote the set of all randomized policy profiles, Ξ_i for randomized policies of i and Ξ_{-i} to denote the set of randomized policy profiles for all parties except i . A randomized policy profile ξ is a sequence (ξ_i) assigning one randomized policy to each party. Each such randomized policy profile induces a unique probability measure on the set of runs, which is denoted as $\text{Prob}^\xi[\cdot]$. We denote the expectation measure associated to $\text{Prob}^\xi[\cdot]$ by $\mathbb{E}^\xi[\cdot]$.

3.3 Objective Function and Values of Contracts

As mentioned in the introduction we identify expected payoff as the canonical property for contracts. The previous section defines expectation measure given randomized policies as the basic semantics. Given the expected payoff, we define values of contracts as the worst-case guaranteed payoff for a given party. We formalize the notion of objective function (the payoff function).

Objective Function. An objective o for a party p is in one of the following forms:

- $(p^+ - p^-)$, where p^+ is the total money received by party p from the contract (by “payout” statements) and p^- is the total money paid by p to the contract (as “payable” parameters).
- An expression containing mathematical and logical operations (addition, multiplication, subtraction, integer division, and, or, not) and variables chosen from the set $N \cup \{m[i] \mid m \in M, i \in I\}$. Here N is the set of numeric variables, $m[i]$ ’s are the values that can be saved inside maps.²
- A sum of the previous two cases.

Informally, p is trying to choose her moves so as to maximize o .

Run Outcomes. Given a run ρ of the program and an objective o for party p , the outcome $\kappa(\rho, o, p)$ is the value of o computed using the valuation at $\text{end}(\rho)$ for all variables and accounting for payments in ρ to compute p^+ and p^- .

Contract Values. Since we consider worst-case guaranteed payoff, we consider that there is an objective o for a single party p which she tries to maximize and all other parties are adversaries who aim to minimize o . Formally, given a contract C and an objective o for party p , we define the value of contract as:

$$V(C, o, p) := \sup_{\xi_p \in \Xi_p} \inf_{\xi_{-p} \in \Xi_{-p}} \mathbb{E}^{(\xi_p, \xi_{-p})} [\kappa(\rho, o, p)],$$

² We are also assuming, as in many programming languages, that $\text{TRUE} = 1$ and $\text{FALSE} = 0$.

This corresponds to p trying to maximize the expected value of o and all other parties maliciously colluding to minimize it. In other words, it provides the worst-case guarantee for party p , irrespective of the behavior of the other parties, which in the worst-case is adversarial to party p .

3.4 Examples

One contribution of our work is to present the simplified programming language, and to show that this simple language can express several classical smart contracts. To demonstrate the applicability, we present several examples of classical smart contracts in this section. In each example, we present a contract and a “buggy” implementation of the same contract that has a different value. In Sect. 6 we show that our automated approach to analyze the contracts can compute contract values with enough precision to differentiate between the correct and the buggy implementation. All of our examples are motivated from well-known bugs that have happened in real life in Ethereum.

Rock-Paper-Scissors. Let our contract be the one specified in Fig. 2 and assume that we want to analyze it from the point of view of the issuer p . Also, let the objective function be $(p^+ - p^- + 10 \cdot \text{AliceWon})$. Intuitively, this means that winning the rock-paper-scissors game is considered to have an additional value of 10, other than the spending and earnings. The idea behind this is similar to the case with chess tournaments, in which players not only win a prize, but can also use their wins to achieve better “ratings”, so winning has extra utility.

A common bug in writing rock-paper-scissors is allowing the parties to move sequentially, rather than concurrently [29]. If parties can move sequentially and the issuer moves after Bob, then she can ensure a utility of 10, i.e. her worst-case expected reward is 10. However, in the correct implementation as in Fig. 2, the best strategy for both players is to bid 0 and then Alice can win the game with probability $1/3$ by choosing each of the three options with equal probability. Hence, her worst-case expected reward is $10/3$.

Auction. Consider an open auction, in which during a fixed time interval everyone is allowed to bid for the good being sold and everyone can see others’ bids. When the bidding period ends a winner emerges and every other participant can get their money back. Let the variable `HighestBid` store the value of the highest bid made at the auction. Then for a party p , one can define the objective as:

$$p^+ - p^- + (\text{Winner}==p) \times \text{HighestBid}.$$

This is of course assuming that the good being sold is worth precisely as much as the highest bid. A correctly written auction should return a value of 0 to every participant, because those who lose the auction must get their money back and the party that wins pays precisely the highest bid. The contract in Fig. 3 (left) is an implementation of such an auction. However, it has a slight problem. The function `bid` allows the winner to reduce her bid. This bug is fixed in the contract on the right.

```

contract BuggyAuction {
  map Bids[0,1000] = 0;
  numeric HighestBid[0,1000] = 0;
  id Winner = null;
  numeric bid[0,1000] = 0;

  function bid[1,10]
  (payable bid : caller) {
    payout(caller, Bids[caller]);
    Bids[caller]=bid;
    if(bid>HighestBid)
    {
      HighestBid = bid;
      Winner = caller;
    }
  }

  function withdraw[11,20]()
  {
    if(caller!=Winner)
    {
      payout(caller, Bids[caller]);
      Bids[caller]=0;
    }
  }
}

contract Auction {
  map Bids[0,1000] = 0;
  numeric HighestBid[0,1000] = 0;
  id Winner = null;
  numeric bid[0,1000] = 0;

  function bid[1,10]
  (payable bid : caller) {
    if(bid<Bids[caller])
      return;
    payout(caller, Bids[caller]);
    Bids[caller]=bid;
    if(bid>HighestBid)
    {
      HighestBid = bid;
      Winner = caller;
    }
  }

  function withdraw[11,20]()
  {
    if(caller!=Winner)
    {
      payout(caller, Bids[caller]);
      Bids[caller]=0;
    }
  }
}

```

Fig. 3. A buggy auction contract (left) and its fixed version (right).

Three-Way Lottery. Consider a three-party lottery contract issued by a party p . The other two players can sign up by buying tickets worth 1 unit each. Then each of the players is supposed to randomly and uniformly choose a nonce. A combination of these nonces produces the winner with equal probability for all three parties. If a person does not make a choice or pay the fees, she will certainly lose the lottery. The rules are such that if the other two parties choose the same nonce, which is supposed to happen with probability $\frac{1}{3}$, then the issuer wins. Otherwise the winner is chosen according to the parity of sum of nonces. This gives everyone a winning probability of $\frac{1}{3}$ if all sides play uniformly at random. However, even if one of the sides refuses to play uniformly at random, the resulting probabilities of winning stays the same because each side's probability of winning is independent of her own choice assuming that others are playing randomly. We assume that the issuer p has objective $p^+ - p^-$. This is because the winner can take other players' money. In a bug-free contract we will expect the value of this objective to be 0, given that winning has a probability of $\frac{1}{3}$. However, the bug here is due to the fact that other parties can collude. For example, the same person might register as both players and then opt for different nonces. This will ensure that the issuer loses. The bug can be solved by ensuring one's probability of winning is $\frac{1}{3}$ if she honestly plays uniformly at random, no matter what other parties do. For more details about this contract see [19].

Token Sale. Consider a contract that sells *tokens* modeling some aspect of the real world, e.g. shares in a company. At first anyone can buy tokens at a fixed price of 1 unit per token. However, there are a limited number of tokens

available and at most 1000 of them are meant to be sold. The tokens can then be transferred between parties, which is the subject of our next example. For now, Fig. 4 (left) is an implementation of the selling phase. However, there is a big problem here. The problem is that one can buy any number of tokens as long as there is at least one token remaining. For example, one might first buy 999 tokens and then buy another 1000. If we analyze the contract from the point of view of a solo party p with objective `balance[p]`, then it must be capped by 1000 in a bug-free contract, while the process described above leads to a value of 1999. The fixed contract is in Fig. 4 (right). This bug is inspired by a very similar real-world bug described in [52].

Token Transfer. Consider the same bug-free token sale as in the previous example, we now add a function for transferring tokens. An owner can choose a recipient and an amount less than or equal to her balance and transfer that many tokens to the recipient. Figure 5 (left) is an implementation of this concept. Taking the same approach and objective as above, we expect a similar result. However, there is again an important bug in this code. What happens if a party transfers tokens to herself? She gets free extra tokens! This has been fixed in the contract on the right. This example models a real-world bug as in [42].

<pre> contract BuggySale { map balance[0,2000] = 0; numeric remaining[0,2000] = 1000; numeric payment[0,2000] = 0; function buy[1,10] (payable payment:caller) { if(remaining<=0){ payout(caller, payment); return; } balance[caller] += payment; remaining -= payment; }} </pre>	<pre> contract Sale { map balance[0,2000] = 0; numeric remaining[0,2000] = 1000; numeric payment[0,2000] = 0; function buy[1,10] (payable payment:caller) { if(remaining-payment<0){ payout(caller, payment); return; } balance[caller] += payment; remaining -= payment; }} </pre>
---	---

Fig. 4. A buggy token sale (left) and its fixed version (right).

Translation to Solidity. All aspects of our programming language are already present in Solidity, except for the global clock and concurrent interactions. The global clock can be modeled by the number of the current block in the blockchain and concurrent interactions can be implemented using commitment schemes. For more details see [19].

4 Bounded Analysis and Games

Since smart contracts can be easily described in our programming language, and programs in our programming language can be translated to Solidity, the

```

contract BuggyTransfer {
map balance[0,2000] = 0;
numeric remaining[0,2000] = 1000;
numeric payment[0,2000] = 0;
numeric amount[0,2000] = 0;
numeric fromBalance[0,2000] = 0;
numeric toBalance[0,2000] = 0;
id recipient = null;

function buy[1,10]...

function transfer[1,10](
  recipient : caller
  amount : caller) {
  fromBalance = balance[caller];
  toBalance = balance[recipient];
  if(fromBalance < amount)
    return;
  fromBalance -= amount;
  toBalance += amount;
  balance[caller] = fromBalance;
  balance[recipient] = toBalance;
}}

contract Transfer {
map balance[0,2000] = 0;
numeric remaining[0,2000] = 1000;
numeric payment[0,2000] = 0;
numeric amount[0,2000] = 0;

id recipient = null;

function buy[1,10]...

function transfer[1,10](
  recipient : caller
  amount : caller) {

  if(balance[caller] < amount)
    return;
  balance[caller] -= amount;
  balance[recipient] += amount;

}}

```

Fig. 5. A buggy transfer function (left) and its fixed version (right).

main aim to automatically compute values of contracts (i.e., compute guaranteed payoff for parties). In this section, we introduce the bounded analysis problem for our programming language framework, and present concurrent games which is the underlying mathematical framework for the bounded analysis problem.

4.1 Bounded Analysis

As is standard in verification, we consider the bounded analysis problem, where the number of parties and the number of function calls are bounded. In standard program analysis, bugs are often detected with a small number of processes, or a small number of context switches between concurrent threads. In the context of smart contracts, we analogously assume that the number of parties and function calls are bounded.

Contracts with Bounded Number of Parties and Function Calls. Formally, a contract with bounded number of parties and function calls is as follows:

- Let C be a contract and $k \in \mathbb{N}$, we define C_k as an equivalent contract that can have at most k parties. This is achieved by letting $\mathbb{P} = \{p_1, p_2, \dots, p_k\}$ be the set of all possible ids in the contract. The set \mathbb{P} must contain all ids that are in the program source, therefore k is at least the number of such ids. Note that this does not restrict that ids are controlled by unique users, and a real-life user can have several different ids. We only restrict the analysis to bounded number of parties interacting with the smart contract.
- To ensure runs are finite, number of function calls by each party is also bounded. Specifically, each party can call each function at most once during each time frame, i.e. between two consecutive ticks of the clock. This

closely resembles real-life contracts in which one's ability to call many functions is limited by the capacity of a block in the blockchain, given that the block must save all messages.

4.2 Concurrent Games

The programming language framework we consider has interacting agents that act simultaneously, and we have the program state. We present the mathematical framework of concurrent games, which are games played on finite state spaces with concurrent interaction between the players.

Concurrent Game Structures. A concurrent two-player game structure is a tuple $G = (S, s_0, A, \Gamma_1, \Gamma_2, \delta)$, where S is a finite set of states, $s_0 \in S$ is the start state, A is a finite set of actions, $\Gamma_1, \Gamma_2 : S \rightarrow 2^A \setminus \emptyset$ such that Γ_i assigns to each state $s \in S$, a non-empty set $\Gamma_i(s) \subseteq A$ of actions available to player i at s , and finally $\delta : S \times A \times A \rightarrow S$ is a transition function that assigns to every state $s \in S$ and action pair $a_1 \in \Gamma_1(s), a_2 \in \Gamma_2(s)$ a successor state $\delta(s, a_1, a_2) \in S$.

Plays and Histories. The game starts at state s_0 . At each state $s_i \in S$, player 1 chooses an action $a_1^i \in \Gamma_1(s_i)$ and player 2 chooses an action $a_2^i \in \Gamma_2(s_i)$. The choices are made simultaneously and independently. The game subsequently transitions to the new state $s_{i+1} = \delta(s_i, a_1, a_2)$ and the same process continues. This leads to an infinite sequence of tuples $p = (s_i, a_1^i, a_2^i)_{i=0}^\infty$ which is called a *play* of the game. We denote the set of all plays by \mathcal{P} . Every finite prefix $p[..r] := ((s_0, a_1^0, a_2^0), (s_1, a_1^1, a_2^1), \dots, (s_r, a_1^r, a_2^r))$ of a play is called a *history* and the set of all histories is denoted by \mathcal{H} . If $h = p[..r]$ is a history, we denote the last state appearing according to h , i.e. $s_{r+1} = \delta(s_r, a_1^r, a_2^r)$, by $last(h)$. We also define $p[..-1]$ as the empty history.

Strategies and Mixed Strategies. A strategy is a recipe that describes for a player the action to play given the current game history. Formally, a strategy φ_i for player i is a function $\varphi_i : \mathcal{H} \rightarrow A$, such that $\varphi_i(h) \in \Gamma_i(last(h))$. A pair $\varphi = (\varphi_1, \varphi_2)$ of strategies for the two players is called a strategy profile. Each such φ induces a unique play. A mixed strategy $\sigma_i : \mathcal{H} \rightarrow \Delta(A)$ for player i given the history of the game. Intuitively, such a strategy suggests a distribution of actions to player i at each step and then she plays one of them randomly according to that distribution. Of course it must be the case that $Supp(\sigma_i(h)) \subseteq \Gamma_i(last(h))$. A pair $\sigma = (\sigma_1, \sigma_2)$ of mixed strategies for the two players is called a mixed strategy profile. Note that mixed strategies generalize strategies with randomization. Every mixed strategy profile $\sigma = (\sigma_1, \sigma_2)$ induces a unique probability measure on the set of plays, which is denoted as $\text{Prob}^\sigma[\cdot]$, and the associated expectation measure is denoted by $\mathbb{E}^\sigma[\cdot]$.

State and History Utilities. In a game structure G , a state utility function u for player 1 is of the form $u : S \rightarrow \mathbb{R}$. Intuitively, this means that when the game enters state s , player 1 receives a reward of $u(s)$. State utilities can be extended to history utilities. We define the utility of a history to be the sum of utilities of all the states included in that history. Formally, if $h = (s_i, a_1^i, a_2^i)_{i=0}^r$, then

$u(h) = \sum_{i=0}^T u(s_i)$. Given a play $p \in \mathcal{P}$, we denote the utility of its prefix of length L by $u_L(p)$.

Games. A game is a pair (G, u) where G is a game structure and u is a utility function for player 1. We assume that player 1 is trying to maximize u , while player 2's goal is to minimize it.

Values. The L -step finite-horizon value of a game (G, u) is defined as

$$v_L(G, u) := \sup_{\sigma_1} \inf_{\sigma_2} \mathbb{E}^{(\sigma_1, \sigma_2)} [u_L(p)], \quad (1)$$

where σ_i iterates over all possible mixed strategies of player i . This models the fact that player 1 is trying to maximize the utility in the first L steps of the run, while player 2 is minimizing it. The values of games can be computed using the value-iteration algorithm or dynamic programming, which is standard. A more detailed overview of the algorithms for games is provided in [19].

Remark 2. Note that in (1), limiting player 2 to pure strategies does not change the value of the game. Hence, we can assume that player 2 is an arbitrarily powerful nondeterministic adversary and get the exact same results.

4.3 Translating Contracts to Games

The translation from bounded smart contracts to games is straightforward, where the states of the concurrent game encodes the states of the contract. Correspondences between objects in the contract and game are as follows: (a) moves in contracts with actions in games; (b) run prefixes in contracts with histories in games; (c) runs in contracts with plays in games; and (d) policies (resp., randomized policies) in contracts with strategies (resp., mixed strategies) in games. Note that since all runs of the bounded contract are finite and have a limited length, we can apply finite horizon analysis to the resulting game, where L is the maximal length of a run in the contract. This gives us the following theorem:

Theorem 1 (Correspondence). *Given a bounded contract C_k for a party \mathfrak{p} with objective o , a concurrent game can be constructed such that value of this game, $v_L(G, u)$, is equal to the value of the bounded contract, $V(C_k, o, \mathfrak{p})$.*

For details of the translation of smart contracts to games and proof of the theorem above see [19].

Remark 3. In standard programming languages, there are no parties to interact and hence the underlying mathematical models are graphs. In contrast, for smart contracts programming languages, where parties interact in a game-like manner, we have to consider games as the mathematical basis of our analysis.

5 Abstraction for Quantitative Concurrent Games

Abstraction is a key technique to handle large-scale systems. In the previous section we described that smart contracts can be translated to games, but due to state-space explosion (since we allow integer variables), the resulting state space of the game is huge. Hence, we need techniques for abstraction, as well as refinement of abstraction, for concurrent games with quantitative utilities. In this section we present such abstraction refinement for quantitative concurrent games, which is our main technical contribution in this paper. We show the soundness of our approach and its completeness in the limit. Then, we introduce a specific method of abstraction, called interval abstraction, which we apply to the games obtained from contracts and show that soundness and refinement are inherited from the general case. We also provide a heuristic for faster refining of interval abstractions for games obtained from contracts.

5.1 Abstraction for Quantitative Concurrent Games

Abstraction considers a partition of the state space, and reduces the number of states by taking each partition set as a state. In case of transition systems (or graphs) the standard technique is to consider existential (or universal) abstraction to define transitions between the partition sets. However, for game-theoretic interactions such abstraction ideas are not enough. We now describe the key intuition for abstraction in concurrent games with quantitative objectives and formalize it. We also provide a simple example for illustration.

Abstraction Idea and Key Intuition. In an abstraction the state space of the game (G, u) is partitioned into several abstract states, where an abstract state represents a set of states of the original game. Intuitively, an abstract state represents a set of similar states of the original game. Given an abstraction our goal is to define two games that can provide lower and upper bound on the value of the original game. This leads to the concepts of lower and upper abstraction.

- *Lower abstraction.* The lower abstraction $(G^\downarrow, u^\downarrow)$ represents a lower bound on the value. Intuitively, the utility is assigned as minimal utility among states in the partition, and when an action profile can lead to different abstract states, then the adversary, i.e. player 2, chooses the transition.
- *Upper abstraction.* The upper abstraction (G^\uparrow, u^\uparrow) represents an upper bound on the value. Intuitively, the utility is assigned as maximal utility among states in the partition, and when an action profile can lead to different abstract states, then player 1 chooses between the possible states.

Informally, the lower abstraction gives more power to the adversary, player 2, whereas the upper abstraction is favorable to player 1.

General Abstraction for Concurrent Games. Given a game (G, u) consisting of a game structure $G = (S, s_0, A, \Gamma_1, \Gamma_2, \delta)$ and a utility function u , and a partition Π of S , the lower and upper abstractions, $(G^\downarrow = (S^a, s_0^a, A^a, \Gamma_1^\downarrow, \Gamma_2^\downarrow, \delta^\downarrow), u^\downarrow)$ and $(G^\uparrow = (S^a, s_0^a, A^a, \Gamma_1^\uparrow, \Gamma_2^\uparrow, \delta^\uparrow), u^\uparrow)$, of (G, u) with respect to Π are defined as:

- $S^a = \Pi \cup D$, where $D = \Pi \times A \times A$ is a set of dummy states for giving more power to one of the players. Members of S^a are called abstracted states.
- The start state of G is in the start state of G^\uparrow and G^\downarrow , i.e. $s_0 \in s_0^a \in \Pi$.
- $A^a = A \cup \Pi$. Each action in abstracted games either corresponds to an action in the original game or to a choice of the next state.
- If two states $s_1, s_2 \in S$, are in the same abstracted state $s^a \in \Pi$, then they must have the same set of available actions for both players, i.e. $\Gamma_1(s_1) = \Gamma_1(s_2)$ and $\Gamma_2(s_1) = \Gamma_2(s_2)$. Moreover, s^a inherits these action sets. Formally, $\Gamma_1^\downarrow(s^a) = \Gamma_1^\uparrow(s^a) = \Gamma_1(s_1) = \Gamma_1(s_2)$ and $\Gamma_2^\downarrow(s^a) = \Gamma_2^\uparrow(s^a) = \Gamma_2(s_1) = \Gamma_2(s_2)$.
- For all $\pi \in \Pi$ and $a_1 \in \Gamma_1^\downarrow(\pi)$ and $a_2 \in \Gamma_2^\downarrow(\pi)$, we have $\delta^\downarrow(\pi, a_1, a_2) = (\pi, a_1, a_2) \in D$. Similarly for $a_1 \in \Gamma_1^\uparrow(\pi)$ and $a_2 \in \Gamma_2^\uparrow(\pi)$, $\delta^\uparrow(\pi, a_1, a_2) = (\pi, a_1, a_2) \in D$. This means that all transitions from abstract states in Π go to the corresponding dummy abstract state in D .
- If $d = (\pi, a_1, a_2) \in D$ is a dummy abstract state, then let $X_d = \{\pi' \in \Pi \mid \exists s \in \pi \ \delta(s, a_1, a_2) \in \pi'\}$ be the set of all partition sets that can be reached from π by a_1, a_2 in G . Then in G^\downarrow , $\Gamma_1^\downarrow(d)$ is a singleton, i.e., player 1 has no choice, and $\Gamma_2^\downarrow(d) = X_d$, i.e., player 2 can choose which abstract state is the next. Conversely, in G^\uparrow , $\Gamma_2^\uparrow(d)$ is a singleton and player 2 has no choice, while $\Gamma_1^\uparrow(d) = X_d$ and player 1 chooses the next abstract state.
- In line with the previous point, $\delta^\downarrow(d, a_1, a_2) = a_2$ and $\delta^\uparrow(d, a_1, a_2) = a_1$ for all $d \in D$ and available actions a_1 and a_2 .
- We have $u^\downarrow(s^a) = \min_{s \in s^a} \{u(s)\}$ and $u^\uparrow(s^a) = \max_{s \in s^a} \{u(s)\}$. The utility of a non-dummy abstracted state in G^\downarrow , resp. G^\uparrow , is the minimal, resp. maximal, utility among the normal states included in it. Also, for each dummy state $d \in D$, we have $u^\downarrow(d) = u^\uparrow(d) = 0$.

Given a partition Π of S , either (i) there is no lower or upper abstraction corresponding to it because it puts states with different sets of available actions together; or (ii) there is a unique lower and upper abstraction pair. Hence we will refer to the unique abstracted pair of games by specifying Π only.

Remark 4. Dummy states are introduced for conceptual clarity in explaining the ideas because in lower abstraction all choices are assigned to player 2 and upper abstraction to player 1. However, in practice, there is no need to create them, as the choices can be allowed to the respective players in the predecessor state.

Example. Figure 6 (left) shows a concurrent game with (G, u) with 4 states. The utilities are denoted in red. The edges correspond to transitions in δ and each edge is labeled with its corresponding action pair. Here $A = \{a, b\}$, $\Gamma_1(s_0) = \Gamma_2(s_0) = \Gamma_2(s_1) = \Gamma_1(s_2) = \Gamma_2(s_2) = \Gamma_2(s_3) = A$ and $\Gamma_1(s_1) = \Gamma_1(s_3) = \{a\}$. Given that action sets for s_0 and s_2 are equal, we can create abstracted games using the partition $\Pi = \{\pi_0, \pi_1, \pi_2\}$ where $\pi_1 = \{s_0, s_2\}$ and other sets are singletons. The resulting game structure is depicted in Fig. 6 (center). Dummy states are shown by circles and whenever a play reaches a dummy state in G^\downarrow , player 2 chooses which red edge should be taken. Conversely, in G^\uparrow player 1 makes this choice. Also, $u^\uparrow(\pi_0) = \max\{u(s_0), u(s_2)\} = 10$, $u^\downarrow(\pi_0) = \min\{u(s_0), u(s_2)\} = 0$

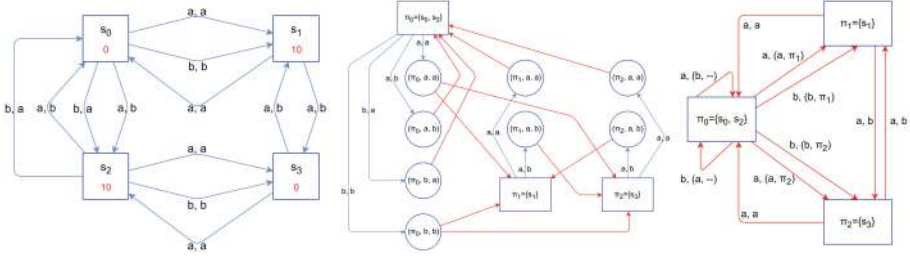


Fig. 6. An example concurrent game (left), abstraction process (center) and the corresponding G^\downarrow without dummy states (right).

and $u^\uparrow(\pi_1)u^\downarrow(\pi_1) = u(s_1) = 10, u^\uparrow(\pi_2) = u^\downarrow(\pi_2) = u(s_3) = 0$. The final abstracted G^\downarrow of the example above, without dummy states, is given in Fig. 6 (right).

5.2 Abstraction: Soundness, Refinement, and Completeness in Limit

For an abstraction we need three key properties: (a) soundness, (b) refinement of the abstraction, and (c) completeness in the limit. The intuitive description is as follows: (a) soundness requires that the value of the games is between the value of the lower and upper abstraction; (b) refinement requires that if the partition is refined, then the values of lower and upper abstraction becomes closer; and (c) completeness requires that if the partitions are refined enough, then the value of the original game can be approximated. We present each of these results below.

Soundness. Soundness means that when we apply abstraction, value of the original game must lie between values of the lower and upper abstractions. Intuitively, this means abstractions must provide us with some interval containing the value of the game. We expect the value of $(G^\downarrow, u^\downarrow)$ to be less than or equal to the value of the original game because in $(G^\downarrow, u^\downarrow)$, the utilities are less than in (G, u) and player 2 has more power, given that she can choose which transition to take. Conversely, we expect (G^\uparrow, u^\uparrow) to have a higher value than (G, u) .

Formal Requirement for Soundness. An abstraction of a game (G, u) leading to abstraction pair $(G^\uparrow, u^\uparrow), (G^\downarrow, u^\downarrow)$ is sound if for every L , we have $v_{2L}(G^\downarrow, u^\downarrow) \leq v_L(G, u) \leq v_{2L}(G^\uparrow, u^\uparrow)$. The factor 2 in the inequalities above is due to the fact that each transition in the original game is modeled by two transitions in abstracted games, one to a dummy state and a second one out of it. We now present our soundness result.

Theorem 2 (Soundness, Proof in [19]). *Given a game (G, u) and a partition Π of its state space, if G^\uparrow and G^\downarrow exist, then the abstraction is sound, i.e. for all L , it is the case that $v_{2L}(G^\downarrow, u^\downarrow) \leq v_L(G, u) \leq v_{2L}(G^\uparrow, u^\uparrow)$.*

Refinement. We say that a partition Π_2 is a refinement of a partition Π_1 , and write $\Pi_2 \sqsubseteq \Pi_1$, if every $\pi \in \Pi_1$ is a union of several π_i 's in Π_2 , i.e. $\pi = \bigcup_{i \in \mathcal{I}} \pi_i$ and for all $i \in \mathcal{I}$, $\pi_i \in \Pi_2$. Intuitively, this means that Π_2 is obtained by further subdividing the partition sets in Π_1 . It is easy to check that \sqsubseteq is a partial order over partitions. We expect that if $\Pi_2 \sqsubseteq \Pi_1$, then the abstracted games resulting from Π_2 give a better approximation of the value of the original game in comparison with abstracted games resulting from Π_1 . This is called the refinement property.

Formal Requirement for the Refinement Property. Two abstractions of a game (G, u) using two partitions Π_1, Π_2 , such that $\Pi_2 \sqsubseteq \Pi_1$, and leading to abstracted games $(G_i^\uparrow, u_i^\uparrow), (G_i^\downarrow, u_i^\downarrow)$ corresponding to each Π_i satisfy the refinement property if for every L , we have $v_{2L}(G_1^\downarrow, u_1^\downarrow) \leq v_{2L}(G_2^\downarrow, u_2^\downarrow) \leq v_{2L}(G_2^\uparrow, u_2^\uparrow) \leq v_{2L}(G_1^\uparrow, u_1^\uparrow)$.

Theorem 3 (Refinement Property, Proof in [19]). *Let $\Pi_2 \sqsubseteq \Pi_1$ be two partitions of the state space of a game (G, u) , then the abstractions corresponding to Π_1, Π_2 satisfy the refinement property.*

Completeness in the Limit. We say that an abstraction is complete in the limit, if by refining it enough the values of upper and lower abstractions get as close together as desired. Equivalently, this means that if we want to approximate the value of the original game within some predefined threshold of error, we can do so by repeatedly refining the abstraction.

Formal Requirement for Completeness in the Limit. Given a game (G, u) , a fixed finite-horizon L and an abstracted game pair corresponding to a partition Π_1 , the abstraction is said to be complete in the limit, if for every $\epsilon \geq 0$ there exists $\Pi_2 \sqsubseteq \Pi_1$, such that if $(G_2^\downarrow, u_2^\downarrow), (G_2^\uparrow, u_2^\uparrow)$ are the abstracted games corresponding to Π_2 , then $v_L(G_2^\uparrow, u_2^\uparrow) - v_L(G_2^\downarrow, u_2^\downarrow) \leq \epsilon$.

Theorem 4 (Completeness in the Limit, Proof in [19]). *Every abstraction on a game (G, u) using a partition Π is complete in the limit for all values of L .*

5.3 Interval Abstraction

In this section, we turn our focus to games obtained from contracts and provide a specific method of abstraction that can be applied to them.

Intuitive Overview. Let (G, u) be a concurrent game obtained from a contract as in the Sect. 4.3. Then the states of G , other than the unique dummy state, correspond to states of the contract C_k . Hence, they are of the form $s = (t, b, l, val, p)$, where t is the time, b the contract balance, l is a label, p is the party calling the current function and val is a valuation. In an abstraction, one cannot put states with different times or labels or callers together, because they might have different moves and hence different action sets in the corresponding game. The main idea in interval abstraction is to break the states according to intervals over their balance and valuations. We can then refine the abstraction by making the intervals smaller. We now formalize this concept.

Objects. Given a contract C_k , let \mathcal{O} be the set of all objects that can have an integral value in a state s of the contract. This consists of the contract balance, numeric variables and $m[\mathfrak{p}]$'s where m is a map variable and \mathfrak{p} is a party. More precisely, $\mathcal{O} = \{\beta\} \cup N \cup \{m[\mathfrak{p}] \mid m \in M, \mathfrak{p} \in \mathbb{P}\}$ where β denotes the balance. For an $o \in \mathcal{O}$, the value assigned to o at state s is denoted by o_s .

Interval Partition. Let C_k be a contract and (G, u) its corresponding game. A partition Π of the state space of G is called an interval partition if:

- The dummy state is put in a singleton set π_d .
- Each $\pi \in \Pi$ except π_d has associated values, $t_\pi, l_\pi, \mathfrak{p}_\pi$ and for each $o \in \mathcal{O}$, $\underline{o}_\pi, \overline{o}_\pi$, such that $\pi = \{s \in S \mid s = (t_\pi, b, l_\pi, val, \mathfrak{p}_\pi)\}$ and for all $o \in \mathcal{O}$, $\underline{o}_\pi \leq s_o \leq \overline{o}_\pi$. Basically, each partition set includes states with the same time, label and caller in which the value of every object o is in an interval $[\underline{o}_\pi, \overline{o}_\pi]$.

We call an abstraction using an interval partition, an interval abstraction.

Refinement Heuristic. We can start with big intervals and continually break them into smaller ones to get refined abstractions and a finer approximation of the game value. We use the following heuristic to choose which intervals to break: Assume that the current abstracted pair of games are $(G^\downarrow, u^\downarrow)$ and (G^\uparrow, u^\uparrow) corresponding to an interval partition Π . Let $d = (\pi_d, a_1, a_2)$ be a dummy state in G^\uparrow and define the skewness of d as $v(G_d^\uparrow, u^\uparrow) - v(G_d^\downarrow, u^\downarrow)$. Intuitively, skewness of d is a measure of how different the outcomes of the games G^\uparrow and G^\downarrow are, from the point when they have reached d . Take a label l with maximal average skewness among its corresponding dummy states and cut all non-unit intervals of it in more parts to get a new partition Π' . Continue the same process until the approximation is as precise as desired. Intuitively, it tries to refine parts of the abstraction that show the most disparity between G^\downarrow and G^\uparrow with the aim to bring their values closer. Our experiments show its effectiveness.

Soundness and Completeness in the Limit. If we restrict our attention to interval abstractions, soundness is inherited from general abstractions and completeness in the limit holds because Π_* is an interval partition. Therefore, using interval abstractions is both sound and complete in the limit.

Interval Refinement. An interval partition Π' is interval refinement of a given interval partition Π if $\Pi' \subseteq \Pi$. Refinement property is inherited from general abstractions. This intuitively means that Π' is obtained by breaking the intervals in some sets of Π into smaller intervals.

Conclusion. We devised a sound abstraction-refinement method for approximating values of contracts. Our method is also complete in the limit. It begins by converting the contract to a game, then applies interval abstraction to the resulting game and repeatedly refines the abstraction using a heuristic until the desired precision is reached.

6 Experimental Results

Implementation and Optimizations. The state-space of the games corresponding to the smart contracts is huge. Hence the original game corresponding to the contract is computationally too expensive to construct. Therefore, we do not first construct the game and then apply abstraction, instead we first apply the interval abstraction, and construct the lower and upper abstraction and compute values in them. We optimized our implementation by removing dummy states and exploiting acyclicity using backward-induction. More details are provided in [19].

Experimental Results. We present our experimental results (Table 1) for the five examples mentioned in Sect. 3.4. In each of the examples, the original game is quite large, and the size of the state space is calculated without creating them. In our experimental results we show the abstracted game size, the refinement of games to larger sizes, and how the lower and upper bound on the values change. We used an Ubuntu machine with 3.2 GHz Intel i7-5600U CPU and 12 GB RAM.

Interpretation of the Experimental Results. Our results demonstrate the effectiveness of our approach in automatically approximating values of large games and real-world smart contracts. Concretely, the following points are shown:

- *Refinement Property.* By repeatedly refining the abstractions, values of lower and upper abstractions get closer at the expense of a larger state space.
- *Distinguishing Correct and Buggy Programs.* Values of the lower and upper abstractions provide an approximation interval containing the contract value. These intervals shrink with refinement until the intervals for correct and buggy programs become disjoint and distinguishable.
- *Bug Detection.* One can anticipate a sensible value for the contract, and an approximation interval not containing the value shows a bug. For example, in token sale, the objective (number of tokens sold) is at most 1000, while results show the buggy program has a value between 1741 and 2000.
- *Quantification of Economic Consequences.* Abstracted game values can also be seen as a method to quantify and find limits to the economic gain or loss of a party. For example, our results show that if the buggy auction contract is deployed, a party can potentially gain no more than 1000 units from it.

7 Comparison with Related Work

Blockchain Security Analysis. The first security analysis of Bitcoin protocol was done by Nakamoto [43] who showed resilience of the blockchain against double-spending. A stateful analysis was done by Sapirshtein et al. [47] and by Sompolinsky and Zohar [49] in which states of the blockchain were considered. It was done using MDPs where only the attacker decides on her actions and the victim follows a predefined protocol. Our paper is the first work that is using two-player and concurrent games to analyze contracts and the first to use stateful analysis on arbitrary smart contracts, rather than a specific protocol.

Table 1. Experimental results for correct and buggy contracts. $l := v(G^\downarrow, u^\downarrow)$ denotes the lower value and $u := v(G^\uparrow, u^\uparrow)$ is the upper value. Times are in seconds.

Rock-Paper-Scissors						
Size	Abstractions					
$> 2.5 \cdot 10^{14}$	Correct Program			Buggy Variant		
	states	$[l, u]$	time	states	$[l, u]$	time
	19440	[0.00, 10.00]	367	25200	[0.00, 10.00]	402
	135945	[1.47, 6.10]	2644	258345	[8.01, 10.00]	4815
	252450	[1.83, 5.59]	3381			

Auction						
Size	Abstractions					
$> 5.2 \cdot 10^{14}$	Correct Program			Buggy Variant		
	states	$[l, u]$	time	states	$[l, u]$	time
	3360	[0, 1000]	68	2880	[0, 1000]	38
	22560	[0, 282]	406	27360	[565, 1000]	552
	272160	[0, 227]	4237	233280	[748, 1000]	3780

Lottery						
Size	Abstractions					
$> 2.5 \cdot 10^8$	Correct Program			Buggy Variant		
	states	$[l, u]$	time	states	$[l, u]$	time
	1539	[-1, 1]	17	1701	[-1, 1]	22
	2457600	[0, 0]	13839	2457600	[-1, -1]	13244

Sale						
Size	Abstractions					
$> 4.6 \cdot 10^{22}$	Correct Program			Buggy Variant		
	states	$[l, u]$	time	states	$[l, u]$	time
	17010	[0, 2000]	226	17010	[0, 2000]	275
	75762	[723, 1472]	1241	81202	[1167, 2000]	1733
	131250	[792, 1260]	2872	124178	[1741, 2000]	2818

Transfer						
Size	Abstractions					
$> 10^{23}$	Correct Program			Buggy Variant		
	states	$[l, u]$	time	states	$[l, u]$	time
	1040	[0, 2000]	20	6561	[0, 2000]	237
	32880	[844, 1793]	562	131520	[1716, 2000]	3979
	148311	[903, 1352]	3740			

Smart Contract Security. Delmolino et al. [29] held a contract programming workshop and showed that even simple contracts can contain incentive misalignment bugs. Luu et al. [41] introduced a symbolic model checker with which they could detect specific erroneous patterns. However the use of model checker cannot be extended to game-theoretic analysis. Bhargavan et al. [9] translated solidity programs to F^* and then used standard verification tools to detect vulnerable code patterns. See [7] for a survey of the known causes for Solidity bugs that result in security vulnerabilities.

Games and Verification. Abstraction for concurrent games has been considered wrt qualitative temporal objectives [3, 22, 28, 44]. Several works considered concurrent games with only pure strategies [28, 36, 37]. Concurrent games with pure strategies are extremely restrictive and effectively similar to turn-based games. The min-max theorem (determinacy) does not hold for them even in special cases of one-shot games or games with qualitative objectives.

Quantitative analysis with games is studied in [12, 17, 21]. However these approaches either consider games without concurrent interactions or do not consider any abstraction-refinement. A quantitative abstraction-refinement framework has been considered in [18]; however, there is no game-theoretic interaction. Abstraction-refinement for games has also been considered [20, 36]; however, these works neither consider games with concurrent interaction, nor quantitative objectives. Moreover, [20, 36] start with a finite-state model without variables, and interval abstraction is not applicable to these game-theoretic frameworks. In contrast, our technical contribution is an abstraction-refinement approach for quantitative games and its application to analysis of smart contracts.

Formal Methods in Security. There is a huge body of work on program analysis for security; see [1, 46] for survey. Formal methods are used to create safe programming languages (e.g., [34, 46]) and to define new logics that can express security properties (e.g., [5, 6, 15]). They are also used to automatically verify security and cryptographic protocols, e.g., [2, 8, 11] for a survey. However, all of these works aimed to formalize qualitative properties such as privacy violation and information leakage. To the best of our knowledge, our framework is the first attempt to use formal methods as a tool for reasoning about monetary losses and identifying them as security errors.

Bounded Model Checking (BMC). BMC was proposed by Biere et al. in 1999 [10]. The idea in BMC is to search for a counterexample in executions whose length is at most k . If no bug is found then one increases k until either a bug is found, the problem becomes intractable, or some pre-known upper bound is reached.

Interval Abstraction. The first infinite abstract domain was introduced in [25]. This was later used to prove that infinite abstract domains can lead to effective static analysis for a given programming language [26]. However, none of the standard techniques is applicable to game analysis.

8 Conclusion

In this work we present a programming language for smart contracts, and an abstraction-refinement approach for quantitative concurrent games to automatically analyze (i.e., compute worst-case guaranteed utilities of) such contracts. This is the first time a quantitative stateful game-theoretic framework is studied for formal analysis of smart contracts. There are several interesting directions of future work. First, we present interval-based abstraction techniques for such games, and whether different abstraction techniques can lead to more scalability or other classes of contracts is an interesting direction of future work. Second, since we consider worst-case guarantees, the games we obtain are two-player zero-sum games. The extension to study multiplayer games and compute values for rational agents is another interesting direction of future work. Finally, in this work we do not consider interaction between smart contracts, and an extension to encompass such study will be a subject of its own.

Acknowledgments. The research was partially supported by Vienna Science and Technology Fund (WWTF) Project ICT15-003, Austrian Science Fund (FWF) NFN Grant No S11407-N23 (RiSE/SHiNE), and ERC Starting grant (279307: Graph Games).

References

1. Abadi, M.: Software security: a formal perspective. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 1–5. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_1
2. Abadi, M., Rogaway, P.: Reconciling two views of cryptography. In: van Leeuwen, J., Watanabe, O., Hagiya, M., Mosses, P.D., Ito, T. (eds.) TCS 2000. LNCS, vol. 1872, pp. 3–22. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44929-9_1
3. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0055622>
4. Anonymous Author: King of the ether (2017). www.kingoftheether.com
5. Arden, O., Liu, J., Myers, A.C.: Flow-limited authorization. In: CSF, pp. 569–583 (2015)
6. Arden, O., Myers, A.C.: A calculus for flow-limited authorization. In: CSF (2016)
7. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts. IACR Cryptology ePrint Archive, 1007 (2016)
8. Avals, M., Pironti, A., Sisto, R.: Formal verification of security protocol implementations: a survey. *Formal Aspects Comput.* **26**(1), 99–123 (2014)
9. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: PLAS. ACM (2016)
10. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
11. Blanchet, B., Chaudhuri, A.: Automated formal analysis of a protocol for secure file sharing on untrusted storage. In: SP, pp. 417–431. IEEE (2008)

12. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_14
13. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: Sok: research perspectives and challenges for bitcoin and cryptocurrencies. In: SP, pp. 104–121. IEEE (2015)
14. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.J.: Symbolic model checking: 1020 states and beyond. *Inf. Comput.* **98**(2), 142–170 (1992)
15. Burrows, M., Abadi, M., Needham, R.M.: A logic of authentication. In: Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, pp. 233–271. The Royal Society (1989)
16. Buterin, V., et al.: Ethereum white paper (2013)
17. Černý, P., Chatterjee, K., Henzinger, T.A., Radhakrishna, A., Singh, R.: Quantitative synthesis for concurrent programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 243–259. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_20
18. Černý, P., Henzinger, T.A., Radhakrishna, A.: Quantitative abstraction refinement. In: POPL (2013)
19. Chatterjee, K., Goharshady, A.K., Velner, Y.: Quantitative analysis of smart contracts (2018). arXiv preprint: [arXiv:1801.03367](https://arxiv.org/abs/1801.03367)
20. Chatterjee, K., Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided planning. In: UAI, pp. 104–111 (2005)
21. Chatterjee, K., Ibsen-Jensen, R.: Qualitative analysis of concurrent mean-payoff games. *Inf. Comput.* **242**, 2–24 (2015)
22. Church, A.: Logic, arithmetic, and automata. In: Proceedings of the International Congress of Mathematicians, pp. 23–35. Institut Mittag-Leffler (1962)
23. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
24. CoinMarketCap: Crypto-currency market capitalizations (2017). coinmarketcap.com
25. Cousot, P., Cousot, R.: Static determination of dynamic properties of generalized type unions. In: ACM Conference on Language Design for Reliable Software, vol. 12, pp. 77–94. ACM (1977)
26. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55844-6_142
27. Daian, P.: Analysis of the DAO exploit (2016). hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit
28. de Alfaro, L., Godefroid, P., Jagadeesan, R.: Three-valued abstractions of games: uncertainty, but with precision. In: LICS. IEEE (2004)
29. Delmolino, K., Arnett, M., Kosba, A.E., Miller, A., Shi, E.: Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. IACR Cryptology ePrint Archive 2015, 460 (2015)
30. Ethereum Foundation: Solidity language documentation (2017)
31. Etherscan: Contract accounts (2017). etherscan.io/accounts/c
32. Etherscan: Token information (2017). etherscan.io/tokens
33. ETHNews: Hkg token has a bug and needs to be reissued (2017). ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued

34. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: automated security certification of android. Technical report (2009)
35. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. LNCS, vol. 1032. Springer, Heidelberg (1996). <https://doi.org/10.1007/3-540-60761-7>
36. Henzinger, T.A., Jhala, R., Majumdar, R.: Counterexample-guided control. In: ICALP (2003)
37. Henzinger, T.A., Majumdar, R., Mang, F., Raskin, J.-F.: Abstract interpretation of game properties. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 220–239. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-45099-3_12
38. Jentzsch, C.: Decentralized autonomous organization to automate governance (2016). <download.slock.it/public/DAO/WhitePaper.pdf>
39. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21:1–21:54 (2009)
40. Johnson, N.: A beginner’s guide to buying an ENS domain (2017)
41. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS, pp. 254–269 (2016)
42. Luu, L., Velner, Y.: Audit report for digix’s smart contract platform (2017)
43. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
44. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190 (1989)
45. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) International Symposium on Programming. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-11494-7_22
46. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
47. Sapirshstein, A., Sompolinsky, Y., Zohar, A.: Optimal selfish mining strategies in bitcoin (2015). arXiv preprint: [arXiv:1507.06183](https://arxiv.org/abs/1507.06183)
48. Simonite, T.: \$80 million hack shows the dangers of programmable money, June 2016. www.technologyreview.com
49. Sompolinsky, Y., Zohar, A.: Bitcoin’s security model revisited. CoRR abs/1605.09193 (2016)
50. Deutsch, J., Jain, S., Saxena, P.: When cryptocurrencies mine their own business? In: Grossklags, J., Preneel, B. (eds.) FC 2016. LNCS, vol. 9603, pp. 499–514. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54970-4_29
51. Toobin, A.: The DAO, Ethereum’s \$150 million blockchain investment fund, has a logic problem (2016). www.inverse.com/article/16314-the-dao-ethereum-s-150-million-blockchain
52. Tran, V., Velner, Y.: Coindash audit report (2017)
53. Wood, G.: Ethereum yellow paper (2014)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Session Types and Concurrency



Session-Typed Concurrent Contracts

Hannah Gommerstadt^(✉), Limin Jia, and Frank Pfenning

Carnegie Mellon University, Pittsburgh, PA, USA
{hgommers,fp}@cs.cmu.edu, liminjia@cmu.edu

Abstract. In sequential languages, dynamic contracts are usually expressed as boolean functions without externally observable effects, written within the language. We propose an analogous notion of concurrent contracts for languages with session-typed message-passing concurrency. Concurrent contracts are partial identity processes that monitor the bidirectional communication along channels and raise an alarm if a contract is violated. Concurrent contracts are session-typed in the usual way and must also satisfy a transparency requirement, which guarantees that terminating compliant programs with and without the contracts are observationally equivalent. We illustrate concurrent contracts with several examples. We also show how to generate contracts from a refinement session-type system and show that the resulting monitors are redundant for programs that are well-typed.

Keywords: Contracts · Session types · Monitors

1 Introduction

Contracts, specifying the conditions under which software components can safely interact, have been used for ensuring key properties of programs for decades. Recently, contracts for distributed processes have been studied in the context of session types [15, 17]. These contracts can enforce the communication protocols, specified as session types, between processes. In this setting, we can assign each channel a monitor for detecting whether messages observed along the channel adhere to the prescribed session type. The monitor can then detect any deviant behavior the processes exhibit and trigger alarms. However, contracts based solely on session types are inherently limited in their expressive power. Many contracts that we would like to enforce cannot even be stated using session types alone. As a simple example, consider a “factorization service” which may be sent a (possibly large) integer x and is supposed to respond with a list of prime factors. Session types can only express that the request is an integer and the response is a list of integers, which is insufficient.

In this paper, we show that by generalizing the class of monitors beyond those derived from session types, we can enforce, for example, that multiplying the numbers in the response yields the original integer x . This paper focuses on monitoring more expressive contracts, specifically those that cannot be expressed with session types, or even refinement types.

© The Author(s) 2018

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 771–798, 2018.

https://doi.org/10.1007/978-3-319-89884-1_27

To handle these contracts, we have designed a model where our monitors execute as transparent processes alongside the computation. They are able to maintain internal state which allows us to check complex properties. These monitoring processes act as partial identities, which do not affect the computation except possibly raising an alarm, and merely observe the messages flowing through the system. They then perform whatever computation is needed, for example, they can compute the product of the factors, to determine whether the messages are consistent with the contract. If the message is not consistent, they stop the computation and blame the process responsible for the mistake. To show that our contracts subsume refinement-based contracts, we encode refinement types in our model by translating refinements into monitors. This encoding is useful because we can show a blame (safety) theorem stating that monitors that enforce a less precise refinement type than the type of the process being monitored will not raise alarms. Unfortunately, the blame theory for the general model is challenging because the contracts cannot be expressed as types.

The main contributions of this paper are:

- A novel approach to contract checking via partial-identity monitors
- A method for verifying that monitors are partial identities, and a proof that the method is correct
- Examples showing the breadth of contracts that our monitors can enforce
- A translation from refinement types to our monitoring processes and a blame theorem for this fragment

The rest of this paper is organized as follows. We first review the background on session types in Sect. 2. Next, we show a range of example contracts in Sect. 3. In Sect. 4, we show how to check that a monitor process is a partial identity and prove the method correct. We then show how we can encode refinements in our system in Sect. 5. We discuss related work in Sect. 6. Due to space constraints, we only present the key theorems. Detailed proofs can be found in our companion technical report [12].

2 Session Types

Session types prescribe the communication behavior of message-passing concurrent processes. We approach them here via their foundation in intuitionistic linear logic [4, 5, 22]. The key idea is that an intuitionistic linear sequent

$$A_1, \dots, A_n \vdash C$$

is interpreted as the interface to a *process expression* P . We label each of the antecedents with a channel name a_i and the succedent with a channel name c . The a_i are the channels *used* and c is the channel *provided* by P .

$$a_1 : A_1, \dots, a_n : A_n \vdash P :: (c : C)$$

We abbreviate the antecedents by Δ . All the channels a_i and c must be distinct, and bound variables may be silently renamed to preserve this invariant in

the rules. Furthermore, the antecedents are considered modulo exchange. Cut corresponds to parallel composition of two processes that communicate along a private channel x , where P is the *provider* along x and Q the *client*.

$$\frac{\Delta \vdash P :: (x : A) \quad x : A, \Delta' \vdash Q :: (c : C)}{\Delta, \Delta' \vdash x:A \leftarrow P ; Q :: (c : C)} \text{ cut}$$

Operationally, the process $x \leftarrow P ; Q$ spawns P as a new process and continues as Q , where P and Q communicate along a fresh channel a , which is substituted for x . We sometimes omit the type A of x in the syntax when it is not relevant.

In order to define the operational semantics rigorously, we use *multiset rewriting* [6]. The configuration of executing processes is described as a collection \mathcal{C} of propositions $\text{proc}(c, P)$ (process P is executing, providing along c) and $\text{msg}(c, M)$ (message M is sent along c). All the channels c provided by processes and messages in a configuration must be distinct.

A cut spawns a new process, and is in fact the only way new processes are spawned. We describe a transition $\mathcal{C} \longrightarrow \mathcal{C}'$ by defining how a subset of \mathcal{C} can be rewritten to a subset of \mathcal{C}' , possibly with a freshness condition that applies to all of \mathcal{C} in order to guarantee the uniqueness of each channel provided.

$$\text{proc}(c, x:A \leftarrow P ; Q) \longrightarrow \text{proc}(a, [a/x]P), \text{proc}(c, [a/x]Q) \quad (a \text{ fresh})$$

Each of the connectives of linear logic then describes a particular kind of communication behavior which we capture in similar rules. Before we move on to that, we consider the identity rule, in logical form and operationally.

$$\frac{}{A \vdash A} \text{ id} \quad \frac{}{b : A \vdash a \leftarrow b :: (a : A)} \text{ id} \quad \text{proc}(a, a \leftarrow b), \mathcal{C} \longrightarrow [b/a]\mathcal{C}$$

Operationally, it corresponds to identifying the channels a and b , which we implement by substituting b for a in the remainder \mathcal{C} of the configuration (which we make explicit in this rule). The process offering a terminates. We refer to $a \leftarrow b$ as *forwarding* since any messages along a are instead “forwarded” to b .

We consider each class of session type constructors, describing their process expression, typing, and asynchronous operational semantics. The linear logical semantics can be recovered by ignoring the process expressions and channels.

Internal and External Choice. Even though we distinguish a *provider* and its *client*, this distinction is orthogonal to the direction of communication: both may either send or receive along a common private channel. Session typing guarantees that both sides will always agree on the direction and kind of message that is sent or received, so our situation corresponds to so-called *binary session types*.

First, the *internal choice* $c : A \oplus B$ requires the provider to send a token inl or inr along c and continue as prescribed by type A or B , respectively. For practical programming, it is more convenient to support n -ary labelled choice $\oplus\{\ell : A_\ell\}_{\ell \in L}$ where L is a set of labels. A process providing $c : \oplus\{\ell : A_\ell\}_{\ell \in L}$ sends a label $k \in L$ along c and continues with type A_k . The client will operate dually, branching on a label received along c .

$$\frac{k \in L \quad \Delta \vdash P :: (c : A_k)}{\Delta \vdash c.k ; P :: (c : \oplus\{\ell : A_\ell\}_{\ell \in L})} \oplus R \quad \frac{\Delta, c : A_\ell \vdash Q_\ell :: (d : D) \quad \text{for every } \ell \in L}{\Delta, c : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \text{case } c (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (d : D)} \oplus L$$

The operational semantics is somewhat tricky, because we communicate asynchronously. We need to spawn a message carrying the label ℓ , but we also need to make sure that the *next* message sent along the same channel does not overtake the first (which would violate session fidelity). Sending a message therefore creates a fresh continuation channel c' for further communication, which we substitute in the continuation of the process. Moreover, the recipient also switches to this continuation channel after the message is received.

$$\begin{aligned} \text{proc}(c, c.k ; P) &\longrightarrow \text{proc}(c', [c'/c]P), \text{msg}(c, c.k ; c \leftarrow c') \quad (c' \text{ fresh}) \\ \text{msg}(c, c.k ; c \leftarrow c'), \text{proc}(d, \text{case } c (\ell \Rightarrow Q_\ell)_{\ell \in L}) &\longrightarrow \text{proc}(d, [c'/c]Q_k) \end{aligned}$$

It is interesting that the message along c , followed by its continuation c' can be expressed as a well-typed process expression using forwarding $c.k ; c \leftarrow c'$. This pattern will work for all other pairs of send/receive operations.

External choice reverses the roles of client and provider, both in the typing and the operational rules. Below are the semantics and the typing is in Fig. 6.

$$\begin{aligned} \text{proc}(d, c.k ; Q) &\longrightarrow \text{msg}(c', c.k ; c' \leftarrow c), \text{proc}(d, [c'/c]Q) \quad (c' \text{ fresh}) \\ \text{proc}(c, \text{case } c (\ell \Rightarrow P_\ell)_{\ell \in L}), \text{msg}(c', c.k ; c' \leftarrow c) &\longrightarrow \text{proc}(c', [c'/c]P_k) \end{aligned}$$

Sending and Receiving Channels. Session types are *higher-order* in the sense that we can send and receive channels along channels. Sending a channel is perhaps less intuitive from the logical point of view, so we show that and just summarize the rules for receiving.

If we provide $c : A \otimes B$, we send a channel $a : A$ along c and continue as B . From the typing perspective, it is a restricted form of the usual two-premise $\otimes R$ rule by requiring the first premise to be an identity. This restriction separates spawning of new processes from the sending of channels.

$$\frac{\Delta \vdash P :: B}{\Delta, a : A \vdash \text{send } c a ; P :: (c : A \otimes B)} \otimes R^* \quad \frac{\Delta, x : A, c : B \vdash Q :: (d : D)}{\Delta, c : A \otimes B \vdash x \leftarrow \text{recv } c ; Q :: (d : D)} \otimes L$$

The operational rules follow the same patterns as the previous case.

$$\begin{aligned} \text{proc}(c, \text{send } c a ; P) &\longrightarrow \text{proc}(c', [c'/c]P), \text{msg}(\text{send } c a ; c \leftarrow c') \quad (c' \text{ fresh}) \\ \text{msg}(c, \text{send } c a ; c \leftarrow c'), \text{proc}(d, x \leftarrow \text{recv } c ; Q) &\longrightarrow \text{proc}(d, [c'/c][a/x]Q) \end{aligned}$$

Receiving a channel (written as a linear implication $A \multimap B$) works symmetrically. Below are the semantics and the typing is shown in Fig. 6.

$$\begin{aligned} \text{proc}(d, \text{send } c a ; Q) &\longrightarrow \text{msg}(c', \text{send } c a ; c' \leftarrow c), \text{proc}(d, [c'/c]Q) \quad (c' \text{ fresh}) \\ \text{proc}(c, x \leftarrow \text{recv } c ; P), \text{msg}(c', \text{send } c a ; c' \leftarrow c) &\longrightarrow \text{proc}(c', [c'/c][a/x]P) \end{aligned}$$

Termination. We have already seen that a process can terminate by forwarding. Communication along a channel ends explicitly when it has type **1** (the unit of \otimes) and is closed. By linearity there must be no antecedents in the right rule.

$$\frac{}{\cdot \vdash \text{close } c :: (c : \mathbf{1})} \mathbf{1}R \quad \frac{\Delta \vdash Q :: (d : D)}{\Delta, c : \mathbf{1} \vdash \text{wait } c ; Q :: (d : D)} \mathbf{1}L$$

Since there cannot be any continuation, the message takes a simple form.

$$\begin{aligned} \text{proc}(c, \text{close } c) &\longrightarrow \text{msg}(c, \text{close } c) \\ \text{msg}(c, \text{close } c), \text{proc}(d, \text{wait } c ; Q) &\longrightarrow \text{proc}(d, Q) \end{aligned}$$

Quantification. First-order quantification over elements of domains such as integers, strings, or booleans allows ordinary basic data values to be sent and received. At the moment, since we have no type families indexed by values, the quantified variables cannot actually appear in their scope. This will change in Sect. 5 so we anticipate this in these rules.

The proof of an existential quantifier contains a witness term, whose value is what is sent. In order to track variables ranging over values, a new context Ψ is added to all judgments and the preceding rules are modified accordingly. All value variables n declared in context Ψ must be distinct. Such variables are not linear, but can be arbitrarily reused, and are therefore propagated to all premises in all rules. We write $\Psi \vdash v : \tau$ to check that value v has type τ in context Ψ .

$$\frac{\Psi \vdash v : \tau \quad \Psi ; \Delta \vdash P :: (c : [v/n]A)}{\Psi ; \Delta \vdash \text{send } c \ v ; P :: (c : \exists n:\tau. A)} \exists R \quad \frac{\Psi, n:\tau ; \Delta, c : A \vdash Q :: (d : D)}{\Psi ; \Delta, c : \exists n:\tau. A \vdash n \leftarrow \text{recv } c ; Q :: (d : D)} \exists L$$

$$\begin{aligned} \text{proc}(c, \text{send } c \ v ; P) &\longrightarrow \text{proc}(c', [c'/c]P), \text{msg}(c, \text{send } c \ v ; c \leftarrow c') \\ \text{msg}(c, \text{send } c \ v ; c \leftarrow c'), \text{proc}(d, n \leftarrow \text{recv } c ; Q) &\longrightarrow \text{proc}(d, [c'/c][v/n]Q) \end{aligned}$$

The situation for universal quantification is symmetric. The semantics are given below and the typing is shown in Fig. 6.

$$\begin{aligned} \text{proc}(d, \text{send } c \ v ; Q) &\longrightarrow \text{msg}(c', \text{send } c \ v ; c' \leftarrow c), \text{proc}(d, [c'/c]Q) \\ \text{proc}(c, x \leftarrow \text{recv } c ; P), \text{msg}(c', \text{send } c \ v ; c' \leftarrow c) &\longrightarrow \text{proc}(c', [c'/c][v/n]P) \end{aligned}$$

Processes may also make internal transitions while computing ordinary values, which we don't fully specify here. Such a transition would have the form

$$\text{proc}(c, P[e]) \longrightarrow \text{proc}(c, P[e']) \quad \text{if } e \mapsto e'$$

where $P[e]$ would denote a process with an ordinary value expression in evaluation position and $e \mapsto e'$ would represent a step of computation.

Shifts. For the purpose of monitoring, it is important to track the direction of communication. To make this explicit, we *polarize* the syntax and use *shifts* to change the direction of communication (for more detail, see prior work [18]).

$$\begin{aligned} \text{Negative types } A^-, B^- &::= \&\{\ell : A_\ell^-\}_{\ell \in L} \mid A^+ \multimap B^- \mid \forall n:\tau. A^- \mid \uparrow A^+ \\ \text{Positive types } A^+, B^+ &::= \oplus\{\ell : A_\ell^+\}_{\ell \in L} \mid A^+ \otimes B^+ \mid 1 \mid \exists n:\tau. A^+ \mid \downarrow A^- \\ \text{Types } A, B, C, D &::= A^- \mid A^+ \end{aligned}$$

From the perspective of the provider, all negative types receive and all positive types send. It is then clear that $\uparrow A$ must receive a **shift** message and then start sending, while $\downarrow A$ must send a **shift** message and then start receiving.

For this restricted form of shift, the logical rules are otherwise uninformative. The semantics are given below and the typing is shown in Fig. 6.

$$\begin{aligned}
 \text{proc}(c, \text{send } c \text{ shift} ; P) &\longrightarrow \text{proc}(c', [c'/c]P), \text{msg}(c, \text{send } c \text{ shift} ; c \leftarrow c') \quad (c' \text{ fresh}) \\
 \text{msg}(c, \text{send } c \text{ shift} ; c \leftarrow c'), \text{proc}(d, \text{shift} \leftarrow \text{recv } d ; Q) &\longrightarrow \text{proc}(d, [c'/c]Q) \\
 \text{proc}(d, \text{send } d \text{ shift} ; Q) &\longrightarrow \text{msg}(c', \text{send } c \text{ shift} ; c' \leftarrow c), \text{proc}(d, [c'/c]Q) \\
 \text{proc}(c, \text{shift} \leftarrow \text{recv } c ; P), \text{msg}(c', \text{send } c \text{ shift} ; c' \leftarrow c) &\longrightarrow \text{proc}(c', [c'/c]P)
 \end{aligned}$$

Recursive Types. Practical programming with session types requires them to be recursive, and processes using them also must allow recursion. For example, lists with elements of type `int` can be defined as the purely positive type list^+ .

$$\text{list}^+ = \oplus\{ \text{cons} : \exists n:\text{int}. \text{list}^+ ; \text{nil} : \mathbf{1} \}$$

A provider of type $c : \text{list}$ is required to send a sequence such as $\text{cons} \cdot v_1 \cdot \text{cons} \cdot v_2 \cdots$ where each v_i is an integer. If it is finite, it must be terminated with $\text{nil} \cdot \text{end}$. In the form of a grammar, we could write

$$\text{From} ::= \text{cons} \cdot v \cdot \text{From} \mid \text{nil} \cdot \text{end}$$

A second example is a multiset (bag) of integers, where the interface allows inserting and removing elements, and testing if it is empty. If the bag is empty when tested, the provider terminates after responding with the `empty` label.

$$\begin{aligned}
 \text{bag}^- = \&\{ \text{insert} : \forall n:\text{int}. \text{bag}^-, \text{remove} : \forall n:\text{int}. \text{bag}^-, \\
 &\text{is_empty} : \uparrow \oplus\{ \text{empty} : \mathbf{1}, \text{nonempty} : \downarrow \text{bag}^- \} \}
 \end{aligned}$$

The protocol now describes the following grammar of exchanged messages, where *To* goes to the provider, *From* comes from the provider, and v stands for integers.

$$\begin{aligned}
 \text{To} &::= \text{insert} \cdot v \cdot \text{To} \mid \text{remove} \cdot v \cdot \text{To} \mid \text{is_empty} \cdot \text{shift} \cdot \text{From} \\
 \text{From} &::= \text{empty} \cdot \text{end} \mid \text{nonempty} \cdot \text{shift} \cdot \text{To}
 \end{aligned}$$

For these protocols to be realized in this form and support rich subtyping and refinement types without change of protocol, it is convenient for recursive types to be *equirecursive*. This means a defined type such as list^+ is viewed as *equal* to its definition $\oplus\{\dots\}$ rather than *isomorphic*. For this view to be consistent, we require type definitions to be *contractive* [11], that is, they need to provide at least one send or receive interaction before recursing.

The most popular formalization of equirecursive types is to introduce an explicit μ -constructor. For example, $\text{list} = \mu\alpha. \oplus\{ \text{cons} : \exists n:\text{int}. \alpha, \text{nil} : \mathbf{1} \}$ with rules unrolling the type $\mu\alpha. A$ to $[(\mu\alpha. A)/\alpha]A$. An alternative (see, for example, Balzers and Pfenning 2017 [3]) is to use an explicit definition just as we stated, for example, `list` and `bag`, and consider the left-hand side *equal* to the right-hand side in our discourse. In typing, this works without a hitch. When we consider subtyping explicitly, we need to make sure we view inference systems on types as being defined *co-inductively*. Since a co-inductively defined judgment essentially expresses the absence of a counterexample, this is exactly what we need for

the operational properties like progress, preservation, or absence of blame. We therefore adopt this view.

Recursive Processes. In addition to recursively defined types, we also need recursively defined processes. We follow the general approach of Toninho et al. [23] for the integration of a (functional) data layer into session-typed communication. A process can be named p , ascribed a type, and be defined as follows.

$$\begin{aligned} p &: \forall n_1:\tau_1. \dots, \forall n_k:\tau_k. \{A \leftarrow A_1, \dots, A_m\} \\ x &\leftarrow p n_1 \dots n_k \leftarrow y_1, \dots, y_m = P \end{aligned}$$

where we check $(n_1:\tau_1, \dots, n_k:\tau_k) ; (y_1:A_1, \dots, y_m:A_m) \vdash P :: (x:A)$

We use such process definitions when spawning a new process with the syntax

$$c \leftarrow p e_1 \dots e_k \leftarrow d_1, \dots, d_m ; P$$

which we check with the rule

$$\frac{(\Psi \vdash e_i : \tau_i)_{i \in \{1, \dots, k\}} \quad \Delta' = (d_1:A_1, \dots, d_m:A_m) \quad \Psi ; \Delta, c : A \vdash Q :: (d : D)}{\Psi ; \Delta, \Delta' \vdash c \leftarrow p e_1 \dots e_k \leftarrow d_1, \dots, d_m ; Q :: (d : D)} \text{ pdef}$$

After evaluating the value arguments, the call consumes the channels d_j (which will not be available to the continuation Q , due to linearity). The continuation Q will then be the (sole) client of c and The new process providing c will execute $[c/x][d_1/y_1] \dots [d_m/y_m]P$.

One more quick shorthand used in the examples: a tail-call $c \leftarrow p \bar{e} \leftarrow \bar{d}$ in the definition of a process that provides along c is expanded into $c' \leftarrow p \bar{e} \leftarrow \bar{d} ; c \leftarrow c'$ for a fresh c' . Depending on how forwarding is implemented, however, it may be much more efficient [13].

Stopping Computation. Finally, in order to be able to successfully monitor computation, we need the capability to stop the computation. We add an **abort** l construct that aborts on a particular label. We also add **assert** blocks to check conditions on observable values. The semantics are given below and the typing is in Fig. 6.

$$\text{proc}(c, \text{assert } l \text{ True}; Q) \longrightarrow \text{proc}(c, Q) \qquad \text{proc}(c, \text{assert } l \text{ False}; Q) \longrightarrow \text{abort}(l)$$

Progress and preservation were proven for the above system, with the exception of the **abort** and **assert** rules, in prior work [18]. The additional proof cases do not change the proof significantly.

3 Contract Examples

In this section, we present monitoring processes that can enforce a variety of contracts. The examples will mainly use lists as defined in the previous section. Our monitors are transparent, that is, they do not change the computation. We accomplish this by making them act as partial identities (described in more

detail in Sect. 4). Therefore, any monitor that enforces a contract on a list must peel off each layer of the type one step at a time (by sending or receiving over the channel as dictated by the type), perform the required checks on values or labels, and then reconstruct the original type (again, by sending or receiving as appropriate).

Refinement. The simplest kind of monitoring process we can write is one that models a refinement of an integer type; for example, a process that checks whether every element in the list is positive. This is a recursive process that receives the head of the list from channel b , checks whether it is positive (if yes, it continues to the next value, if not it aborts), and then sends the value along to reconstruct the monitored list a . We show three refinement monitors in Fig. 1. The process pos implements the refinement mentioned above.

<pre>pos : {list ← list} a ← pos_mon ← b = case b of nil ⇒ a.nil ; wait b ; close a cons ⇒ x ← rcv b ; assert(x > 0)^ρ ; a.cons ; send a x ; a ← pos_mon ← b ;</pre>	<pre>empty : {list ← list} a ← empty ← b = case b of nil ⇒ wait b ; a.nil ; close a cons ⇒ abort^ρ ;</pre>	<pre>nempty : {list ← list} a ← nempty ← b = case b of nil ⇒ abort^ρ cons ⇒ a.cons ; x ← rcv b ; send a x ; a ← b ;</pre>
---	--	---

Fig. 1. Refinement examples

Our monitors can also exploit information that is contained in the labels in the external and internal choices. The **empty** process checks whether the list b is empty and aborts if b sends the label **cons**. Similarly, the **nempty** monitor checks whether the list b is not empty and aborts if b sends the label **nil**. These two monitors can then be used by a process that zips two lists and aborts if they are of different lengths. These two monitors enforce the refinements $\{\text{nil}\} \subseteq \{\text{nil}, \text{cons}\}$ and $\{\text{cons}\} \subseteq \{\text{nil}, \text{cons}\}$. We discuss how to generate monitors from refinement types in more detail in Sect. 5.

Monitors with Internal State. We now move beyond refinement contracts, and model contracts that have to maintain some internal state (Fig. 2).

We first present a monitor that checks whether the given list is sorted in ascending order (**ascending**). The monitor's state consists of a lower bound on the subsequent elements in the list. This value has an option type, which can either be **None** if no bound has yet been set, or **Some b** if b is the current bound.

If the list is empty, there is no bound to check, so no contract failure can happen. If the list is nonempty, we check to see if a bound has already been set. If not, we set the bound to be the first received element. If there is already a bound in place, then we check if the received element is greater or equal to the bound. If it is not, then the list must be unsorted, so we abort with a contract

```

ascending : option int → {list ← list};;
m ← ascending bound ← n =
  case n of
  | nil ⇒ m.nil ; wait n ; close m
  | cons ⇒ x ← recv n ;
    case bound of
    | None ⇒ m.cons ; send m x ;
      m ← ascending (Some x) ← n
    | Some a ⇒ assert (x ≥ a)ρ ;
      m.cons ; send m x ;
      m ← ascending (Some x) ← n;;

match : int → {list ← list};;
a ← match count ← b =
  case b of
  | nil ⇒ assert (count = 0)ρ ;
    a.nil ; wait b ; close a
  | cons ⇒ a.cons ; x ← recv b ;
    if (x = 1) then send a x ;
      a ← match (count + 1) ← b;
    else if (x = -1)
      then assert(count > 0)ρ ;
        send a x ;
        a ← match (count - 1) ← b ;
    else abortρ //invalid input

```

Fig. 2. Monitors using internal state

failure. Note that the output list m is the same as the input list n because every element that we examine is then passed along unchanged to m .

We can use the **ascending** monitor to verify that the output list of a sorting procedure is in sorted order. To take the example one step further, we can verify that the elements in the output list are in fact a permutation of the elements in the input list of the sorting procedure as follows. Using a reasonable hash function, we hash each element as it is sent to the sorting procedure. Our monitor then keeps track of a running total of the sum of the hashes, and as elements are received from the sorting procedure, it computes their hash and subtracts it from the total. After all of the elements are received, we check that the total is 0 – if it is, with high probability, the two lists are permutations of each other. This example is an instance of *result checking*, inspired by Wasserman and Blum [26]. The monitor encoding is straightforward and omitted from the paper.

Our next example **match** validates whether a set of right and left parentheses match. The monitor can use its internal state to push every left parenthesis it sees on its stack and to pop it off when it sees a right parenthesis. For brevity, we model our list of parentheses by marking every left parenthesis with a 1 and right parenthesis with a -1. So the sequence $()()$ would look like 1, -1, 1, -1, -1. As we can see, this is not a proper sequence of parenthesis because adding all of the integer representations does not yield 0. In a similar vein, we can implement a process that checks that a tree is serialized correctly, which is related to recent work on context-free session types by Thiemann and Vasconcelos [21].

Mapper. Finally, we can also define monitors that check higher-order contracts, such as a contract for a mapping function (Fig. 3). Consider the mapper which takes an integer and doubles it, and a function **map** that applies this mapper to a list of integers to produce a new list of integers. We can see that any integer that the mapper has produced will be strictly larger than the original integer, assuming the original integer is positive. In order to monitor this contract, it makes sense to impose a contract on the mapper itself. This **mapper_mon** process enforces both the precondition, that the original integer is positive, and the

```

mapper_tp : {&{done : 1 ; next : ∀n : int. ∃n' : int. mapper_tp}}
m ← mapper =
  case m of
  | done ⇒ close m
  | next ⇒ x ← recv m ; send m (2 * x) ; m ← mapper
map : {list ← mapper_tp ; list}
k ← map ← m l =
  case l of
  | nil ⇒ m.done ; k.nil ; wait l ; close k
  | cons ⇒ m' ← mapper_mon ← m ; //run monitor
    x ← recv l ; send m' x ; y ← recv m' ; k.cons ; send k y ; k ← map m' l ;
mapper_mon : {mapper_tp ← mapper_tp}
n ← mapper_mon ← m =
  case n of
  | done ⇒ m.done ; wait m ; close n
  | next ⇒ x ← recv n ; assert(x > 0)ρ1 //checks precondition
    m.next ; send m x ; y ← recv m ; assert(y > x)ρ2 //checks postcondition
    send n y ; n ← mapper_mon ← m

```

Fig. 3. Higher-order monitor

postcondition, that the resulting integer is greater than the original. We can now run the monitor on the mapper, in the `map` process, before applying the mapper to the list `l`.

4 Monitors as Partial Identity Processes

In the literature on contracts, they are often depicted as guards on values sent to and returned from functions. In our case, they really *are* processes that monitor message-passing communications between processes. For us, a central property of contracts is that a program may be executed with or without contract checking and, unless an alarm is raised, the observable outcome should be the same. This means that contract monitors should be *partial identity processes* passing messages back and forth along channels while testing properties of the messages.

This may seem very limiting at first, but session-typed processes can maintain local state. For example, consider the functional notion of a *dependent contract*, where the contract on the result of a function depends on its input. Here, a function would be implemented by a process to which you send the arguments and which sends back the return value *along the same channel*. Therefore, a monitor can remember any (non-linear) “argument values” and use them to validate the “result value”. Similarly, when a list is sent element by element, properties that can be easily checked include constraints on its length, or whether it is in ascending order. Moreover, local state can include additional (private) concurrent processes.

This raises a second question: how can we guarantee that a monitor really is a partial identity? The criterion should be general enough to allow us to naturally

express the contracts from a wide range of examples. A key constraint is that *contracts are expressed as session-typed processes*, just like functional contracts should be expressed within the functional language, or object contracts within the object oriented language, etc.

The purpose of this section is to present and prove the correctness of a criterion on session-typed processes that guarantees that they are observationally equivalent to partial identity processes. All the contracts in this paper can be verified to be partial identities under our definition.

4.1 Buffering Values

As a first simple example let's take a process that receives one positive integer n and factors it into two integers p and q that are sent back where $p \leq q$. The part of the specification that is *not* enforced is that if n is not prime, p and q should be proper factors, but we at least enforce that all numbers are positive and $n = p * q$. We are being very particular here, for the purpose of exposition, marking the place where the direction of communication changes with a shift (\uparrow). Since a minimal number of shifts can be inferred during elaboration of the syntax [18], we suppress it in most examples.

```
factor_t =  $\forall n:\text{int}. \uparrow \exists p:\text{int}. \exists q:\text{int}. \mathbf{1}$ 
factor_monitor : {factor_t  $\leftarrow$  factor_t}
 $c \leftarrow$  factor_monitor  $\leftarrow d =$ 
   $n \leftarrow$  recv  $c$  ; assert  $(n > 0)^{\rho_1}$  ; shift  $\leftarrow$  recv  $c$  ; send  $d$   $n$  ; send  $d$  shift ;
   $p \leftarrow$  recv  $d$  ; assert  $(p > 0)^{\rho_2}$  ;  $q \leftarrow$  recv  $d$  ; assert  $(q > 0)^{\rho_3}$  ; assert  $(p \leq q)^{\rho_4}$  ;
  assert  $(n = p * q)^{\rho_5}$  ; send  $c$   $p$  ; send  $c$   $q$  ;  $c \leftarrow d$ 
```

This is a one-time interaction (the session type `factor_t` is not recursive), so the monitor terminates. It terminates here by forwarding, but we could equally well have replaced it by its identity-expanded version at type `1`, which is `wait d ; close c .`

The contract could be invoked by the provider or by the client. Let's consider how a provider `factor` might invoke it:

```
factor : {factor_t}
 $c \leftarrow$  factor =
   $c' \leftarrow$  factor_raw ;  $c' \leftarrow$  factor_monitor  $\leftarrow c'$  ;  $c \leftarrow c'$ 
```

To check that `factor_monitor` is a partial identity we need to track that p and q are received from the provider, in this order. In general, for any received message, we need to enter it into a message queue q and we need to check that the messages are passed on in the correct order. As a first cut (to be generalized several times), we write for negative types:

$$[q](b : B^-) ; \Psi \vdash P :: (a : A^-)$$

which expresses that the two endpoints of the monitor are $a : A^-$ and $b : B^-$ (both negative), and we have already received the messages in q along a . The context Ψ declares types for local variables.

A monitor, at the top level, is defined with

$$\begin{aligned} \text{mon} : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \{A \leftarrow A\} \\ a \leftarrow \text{mon } x_1 \dots x_n \leftarrow b = P \end{aligned}$$

where context Ψ declares value variables x . The body P here is type-checked as one of (depending on the polarity of A)

$$[] (b : A^-) ; \Psi \vdash P :: (a : A^-) \quad \text{or} \quad (b : A^+) ; \Psi \vdash P :: [] (a : A^+)$$

where $\Psi = (x_1 : \tau_1) \dots (x_n : \tau_n)$. A use such as

$$c \leftarrow \text{mon } e_1 \dots e_n \leftarrow c$$

is transformed into

$$c' \leftarrow \text{mon } e_1 \dots e_n \leftarrow c ; c \leftarrow c'$$

for a fresh c' and type-checked accordingly.

In general, queues have the form $q = m_1 \dots m_n$ with

$$\begin{array}{lll} m ::= l_k & \text{labels} & \oplus, \& \\ | c & \text{channels} & \otimes, \multimap \\ | \text{end close} & \mathbf{1} & \end{array} \quad \begin{array}{ll} | n & \text{value variables } \exists, \forall \\ | \text{shift shifts} & \uparrow, \downarrow \end{array}$$

where m_1 is the front of the queue and m_n the back.

When a process P receives a message, we add it to the end of the queue q . We also need to add it to Ψ context, marked as *unrestricted* (non-linear) to remember its type. In our example $\tau = \text{int}$.

$$\frac{[q \cdot n](b : B) ; \Psi, n : \tau \vdash P :: (a : A^-)}{[q](b : B) ; \Psi \vdash n \leftarrow \text{recv } a ; P :: (a : \forall n : \tau. A^-)} \quad \forall R$$

Conversely, when we *send* along b the message must be equal to the one at the front of the queue (and therefore it must be a variable). The m is a value variable and remains in the context so it can be reused for later assertion checks. However, it could never be sent again since it has been removed from the queue.

$$\frac{[q](b : [m/n]B) ; \Psi, m : \tau \vdash P :: (a : A)}{[m \cdot q](b : \forall n : \tau. B) ; \Psi, m : \tau \vdash \text{send } b \ m ; Q :: (a : A)} \quad \forall L$$

All the other send and receive rules for negative types (\forall , \multimap , $\&$) follow exactly the same pattern. For positive types, a queue must be associated with the channel along which the monitor provides (the succedent of the sequent judgment).

$$(b : B^+) ; \Psi \vdash Q :: [q](a : A^+)$$

Moreover, when *end* has been received along b the corresponding process has terminated and the channel is closed, so we generalize the judgment to

$$\omega ; \Psi \vdash Q :: [q](a : A^+) \quad \text{with } \omega = \cdot \mid (b : B).$$

The shift messages change the direction of communication. They therefore need to switch between the two judgments and also ensure that the queue has been emptied before we switch direction. Here are the two rules for \uparrow , which appears in our simple example:

$$\frac{[q \cdot \text{shift}](b : B^-) ; \Psi \vdash P :: (a : A^+)}{[q](b : B^-) ; \Psi \vdash \text{shift} \leftarrow \text{recv } a ; P :: (a : \uparrow A^+)} \uparrow R$$

We notice that after receiving a *shift*, the channel a already changes polarity (we now have to send along it), so we generalize the judgment, allowing the succedent to be either positive or negative. And conversely for the other judgment.

$$\frac{[q](b : B^-) ; \Psi \vdash P :: (a : A)}{\omega ; \Psi \vdash Q :: [q](a : A^+) \quad \text{where } \omega = \cdot \mid (b : B)}$$

When we *send* the final shift, we initialize a new empty queue. Because the queue is empty the two sides of the monitor must have the same type.

$$\frac{(b : B^+) ; \Psi \vdash Q :: [](a : B^+)}{[\text{shift}](b : \uparrow B^+) ; \Psi \vdash \text{send } b \text{ shift} ; Q :: (a : B^+)} \uparrow L$$

The rules for forwarding are also straightforward. Both sides need to have the same type, and the queue must be empty. As a consequence, the immediate forward is always a valid monitor at a given type.

$$\frac{}{(b : A^+) ; \Psi \vdash a \leftarrow b :: [](a : A^+)} \text{id}^+ \quad \frac{}{[] (b : A^-) ; \Psi \vdash a \leftarrow b :: (a : A^-)} \text{id}^-$$

4.2 Rule Summary

The current rules allow us to communicate *only along the channels a and b that are being monitored*. If we send channels along channels, however, these channels must be recorded in the typing judgment, but we are not allowed to communicate along them directly. On the other hand, if we spawn internal (local) channels, say, as auxiliary data structures, we should be able to interact with them since such interactions are not externally observable. Our judgment thus requires two additional contexts: Δ for channels internal to the monitor, and Γ for externally visible channels that may be sent along the monitored channels. Our full judgments therefore are

$$\frac{[q](b : B^-) ; \Psi ; \Gamma ; \Delta \vdash P :: (a : A)}{\omega ; \Psi ; \Gamma ; \Delta \vdash Q :: [q](a : A^+) \quad \text{where } \omega = \cdot \mid (b : B)}$$

So far, it is given by the following rules

$$\frac{(\forall \ell \in L) \quad (b : B_\ell) ; \Psi ; \Gamma ; \Delta \vdash Q_\ell :: [q \cdot \ell](a : A^+)}{(b : \oplus \{\ell : B_\ell\}_{\ell \in L}) ; \Psi ; \Gamma ; \Delta \vdash \text{case } b \ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: [q](a : A^+)} \oplus L$$

$$\frac{\omega ; \Psi ; \Gamma ; \Delta \vdash P :: [q](a : B_k) \quad (k \in L)}{\omega ; \Psi ; \Gamma ; \Delta \vdash a.k ; P :: [k \cdot q](a : \oplus \{\ell : B_\ell\}_{\ell \in L})} \oplus R$$

$$\begin{array}{c}
\frac{(\forall \ell \in L) \quad [q \cdot \ell](b : B) ; \Psi ; \Gamma ; \Delta \vdash P_\ell :: (a : A_\ell)}{[q](b : B) ; \Psi ; \Gamma ; \Delta \vdash \text{case } a \ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (a : \&\{\ell : A_\ell\}_{\ell \in L})} \&R \\
\frac{[q](b : B_k) ; \Psi ; \Gamma ; \Delta \vdash P :: (a : A) \quad (k \in L)}{[k \cdot q](b : \oplus\{\ell : B_\ell\}_{\ell \in L}) ; \Psi ; \Gamma ; \Delta \vdash b.k ; P :: (a : A)} \&L \\
\frac{(b : B) ; \Psi ; \Gamma, x:C ; \Delta \vdash Q :: [q \cdot x](a : A)}{(b : C \otimes B) ; \Psi ; \Gamma ; \Delta \vdash x \leftarrow \text{recv } b ; Q :: [q](a : A)} \otimes L \\
\frac{\omega ; \Psi ; \Gamma ; \Delta \vdash P :: [q](a : A)}{\omega ; \Psi ; \Gamma, x:C ; \Delta \vdash \text{send } a \ x ; P :: [x \cdot q](a : C \otimes A)} \otimes R \\
\frac{[q \cdot x](b : B) ; \Psi ; \Gamma, x:C ; \Delta \vdash P :: (a : A)}{[q](b : B) ; \Psi ; \Gamma ; \Delta \vdash x \leftarrow \text{recv } a ; P :: (a : C \multimap A)} \multimap R \\
\frac{[q](b : B) ; \Psi ; \Gamma ; \Delta \vdash Q :: (a : A)}{[x \cdot q](b : C \multimap B) ; \Psi ; \Gamma, x:C ; \Delta \vdash \text{send } b \ x ; Q :: (a : A)} \multimap L \\
\frac{\cdot ; \Psi ; \Gamma ; \Delta \vdash Q :: [q \cdot \text{end}](a : A)}{(b : \mathbf{1}) ; \Psi ; \Gamma ; \Delta \vdash \text{wait } b ; Q :: [q](a : A)} \mathbf{1}L \\
\frac{}{\cdot ; \Psi ; \cdot ; \cdot \vdash \text{close } a :: [\text{end}](a : \mathbf{1})} \mathbf{1}R \\
\frac{(b : B) ; \Psi, n:\tau ; \Gamma ; \Delta \vdash Q :: [q \cdot n](a : A)}{(b : \exists n:\tau. B) ; \Psi ; \Gamma ; \Delta \vdash n \leftarrow \text{recv } b ; Q :: [q](a : A)} \exists L \\
\frac{\omega ; \Psi, m:\tau ; \Gamma ; \Delta \vdash P :: [q](a : [m/n]A)}{\omega ; \Psi, m:\tau ; \Gamma ; \Delta \vdash \text{send } a \ m ; P :: [m \cdot q](a : \exists n:\tau. A)} \exists R \\
\frac{[q \cdot n](b : B) ; \Psi, n:\tau ; \Gamma ; \Delta \vdash P :: (a : A^-)}{[q](b : B) ; \Psi ; \Gamma ; \Delta \vdash v \leftarrow \text{recv } a ; P :: (a : \forall n:\tau. A^-)} \forall R \\
\frac{[q](b : [m/n]B) ; \Psi, m:\tau ; \Gamma ; \Delta \vdash P :: (a : A)}{[m \cdot q](b : \forall n:\tau. B) ; \Psi, m:\tau ; \Gamma ; \Delta \vdash \text{send } b \ m ; Q :: (a : A)} \forall L \\
\frac{(b : B^-) ; \Psi ; \Gamma ; \Delta \vdash Q :: [q \cdot \text{shift}](a : A^+)}{(b : \downarrow B^-) ; \Psi ; \Gamma ; \Delta \vdash \text{shift} \leftarrow \text{recv } b ; Q :: [q](a : A^+)} \downarrow L \\
\frac{[] (b : A^-) ; \Psi ; \Gamma ; \Delta \vdash P :: (a : A^-)}{(b : A^-) ; \Psi ; \Gamma ; \Delta \vdash \text{send } a \ \text{shift} ; P :: [\text{shift}](a : \downarrow A^-)} \downarrow R \\
\frac{[q \cdot \text{shift}](b : B^-) ; \Psi ; \Gamma ; \Delta \vdash P :: (a : A^+)}{[q](b : B^-) ; \Psi ; \Gamma ; \Delta \vdash \text{shift} \leftarrow \text{recv } a ; P :: (a : \uparrow A^+)} \uparrow R \\
\frac{(b : B^+) ; \Psi ; \Gamma ; \Delta \vdash Q :: [] (a : B^+)}{[\text{shift}](b : \uparrow B^+) ; \Psi ; \Gamma ; \Delta \vdash \text{send } b \ \text{shift} ; Q :: (a : B^+)} \uparrow L
\end{array}$$

4.3 Spawning New Processes

The most complex part of checking that a process is a valid monitor involves spawning new processes. In order to be able to spawn and use local (private) processes, we have introduced the (so far unused) context Δ that tracks such channels. We use it here only in the following two rules:

$$\frac{\Psi ; \Delta \vdash P :: (c : C) \quad \omega ; \Psi ; \Gamma ; \Delta', c : C \vdash Q :: [q](a : A^+)}{\omega ; \Psi ; \Gamma ; \Delta, \Delta' \vdash (c : C) \leftarrow P ; Q :: [q](a : A^+)} \text{cut}_1^+$$

$$\frac{\Psi ; \Delta \vdash P :: (c : C) \quad [q](b : B^-) ; \Psi ; \Gamma ; \Delta', c : C \vdash Q :: (a : A)}{[q](b : B^-) ; \Psi ; \Gamma ; \Delta, \Delta' \vdash (c : C) \leftarrow P ; Q :: (a : A)} \text{cut}_1^-$$

The second premise (that is, the continuation of the monitor) remains the monitor, while the first premise corresponds to a freshly spawned local progress accessible through channel c . All the ordinary left rules for sending or receiving along channels in Δ are also available for the two monitor validity judgments. By the strong ownership discipline of intuitionistic session types, none of this information can flow out of the monitor.

It is also possible for a single monitor to decompose into two monitors that operate concurrently, in sequence. In that case, the queue q may be split anywhere, as long as the intermediate type has the right polarity. Note that Γ must be chosen to contain all channels in q_2 , while Γ' must contain all channels in q_1 .

$$\frac{\omega ; \Psi ; \Gamma ; \Delta \vdash P :: [q_2](c : C^+) \quad (c : C^+) ; \Psi ; \Gamma' ; \Delta' \vdash Q :: [q_1](a : A^+)}{\omega ; \Psi ; \Gamma, \Gamma' ; \Delta, \Delta' \vdash c : C^+ \leftarrow P ; Q :: [q_1 \cdot q_2](a : A^+)} \text{cut}_2^+$$

Why is this correct? The first messages sent along a will be the messages in q_1 . If we receive messages along c in the meantime, they will be first the messages in q_2 (since P is a monitor), followed by any messages that P may have received along b if $\omega = (b : B)$. The second rule is entirely symmetric, with the flow of messages in the opposite direction.

$$\frac{[q_1](b : B^-) ; \Psi ; \Gamma ; \Delta \vdash P :: (c : C^-) \quad [q_2](c : C^-) ; \Psi' ; \Gamma' ; \Delta' \vdash Q :: (a : A)}{[q_1 \cdot q_2](b : B^-) ; \Psi ; \Gamma, \Gamma' ; \Delta, \Delta' \vdash c : C^- \leftarrow P ; Q :: (a : A)} \text{cut}_2^-$$

The next two rules allow a monitor to be attached to a channel x that is passed between a and b . The monitored version of x is called x' , where x' is chosen fresh. This apparently violates our property that we pass on all messages exactly as received, because here we pass on a monitored version of the original. However, if monitors are partial identities, then the original x and the new x' are indistinguishable (unless a necessary alarm is raised), which will be a tricky part of the correctness proof.

$$\frac{(x : C^+) ; \Psi ; \cdot ; \Delta \vdash P :: [](x' : C^+) \quad \omega ; \Psi ; \Gamma, x' : C^+ ; \Delta' \vdash Q :: [q_1 \cdot x' \cdot q_2](a : A^+)}{\omega ; \Psi ; \Gamma, x : C^+ ; \Delta, \Delta' \vdash x' \leftarrow P ; Q :: [q_1 \cdot x \cdot q_2](a : A^+)} \text{cut}_3^{++}$$

$$\frac{[](x : C^-) ; \Psi ; \cdot ; \Delta \vdash P :: (x' : C^-) \quad [q_1 \cdot x' \cdot q_2](b : B^-) ; \Psi ; \Gamma, x' : C^- ; \Delta' \vdash Q :: (a : A)}{[q_1 \cdot x \cdot q_2](b : B^-) ; \Psi ; \Gamma ; \Delta, \Delta' \vdash x' \leftarrow P ; Q :: (a : A)} \text{cut}_3^{--}$$

There are two more versions of these rules, depending on whether the types of x and the monitored types are positive or negative. These rules play a critical role in monitoring higher-order processes, because monitoring $c : A^+ \multimap B^-$ may require us to monitor the continuation $c : B^-$ (already covered) but also communication along the channel $x : A^+$ received along c .

In actual programs, we mostly use cut $x \leftarrow P ; Q$ in the form $x \leftarrow p \bar{e} \leftarrow \bar{d} ; Q$ where p is a defined process. The rules are completely analogous, except that for those rules that require splitting a context in the conclusion, the arguments \bar{d} will provide the split for us. When a new sub-monitor is invoked in this way, we remember and eventually check that the process p must also be a partial identity process, unless we are already checking it. This has the effect that recursively defined monitors with proper recursive calls are in fact allowed. This is important, because monitors for recursive types usually have a recursive structure. An illustration of this can be seen in `pos` in Fig. 1.

4.4 Transparency

We need to show that monitors are *transparent*, that is, they are indeed observationally equivalent to partial identity processes. Because of the richness of types and process expressions and the generality of the monitors allowed, the proof has some complexities. First, we define the configuration typing, which consists of just three rules. Because we also send and receive ordinary values, we also need to type (closed) substitutions $\sigma = (v_1/n_1, \dots, v_k/n_k)$ using the judgment $\sigma :: \Psi$.

$$\frac{}{(\cdot) :: (\cdot)} \quad \frac{\cdot \vdash v : \tau}{(v/n) :: (n : \tau)} \quad \frac{\sigma_1 :: \Psi_1 \quad \sigma_2 :: \Psi_2}{(\sigma_1, \sigma_2) :: (\Psi_1, \Psi_2)}$$

For configurations, we use the judgment

$$\Delta \vdash \mathcal{C} :: \Delta'$$

which expresses that process configuration \mathcal{C} *uses* the channels in Δ and *provides* the channels in Δ' . Channels that are neither used nor offered by \mathcal{C} are “passed through”. Messages are just a restricted form of processes, so they are typed exactly the same way. We write *pred* for either `proc` or `msg`.

$$\frac{}{\Delta \vdash (\cdot) :: \Delta} \quad \frac{\Delta_0 \vdash \mathcal{C}_1 :: \Delta_1 \quad \Delta_1 \vdash \mathcal{C}_2 :: \Delta_2}{\Delta_0 \vdash \mathcal{C}_1, \mathcal{C}_2 :: \Delta_2} \\ \frac{\Psi ; \Delta \vdash P :: (c : A) \quad \sigma :: \Psi}{\Delta', \Delta[\sigma] \vdash \text{pred}(c, P[\sigma]) :: (\Delta', c : A[\sigma])} \quad \text{pred} ::= \text{proc} \mid \text{msg}$$

To characterize observational equivalence of processes, we need to first characterize the possible messages and the direction in which they flow: towards the client (channel type is positive) or towards the provider (channel type is negative). We summarize these in the following table. In each case, c is the channel along with the message is transmitted, and c' is the continuation channel.

Message to client of c		Message to provider of c	
$\text{msg}^+(c, c.k ; c \leftarrow c')$	(\oplus)	$\text{msg}^-(c', c.k ; c' \leftarrow c)$	$(\&)$
$\text{msg}^+(c, \text{send } c\ d ; c \leftarrow c')$	(\otimes)	$\text{msg}^-(c', \text{send } c\ d ; c' \leftarrow c)$	$(-\circ)$
$\text{msg}^+(c, \text{close } c)$	$(\mathbf{1})$		
$\text{msg}^+(c, \text{send } c\ v ; c \leftarrow c')$	(\exists)	$\text{msg}^-(c', \text{send } c\ v ; c' \leftarrow c)$	(\forall)
$\text{msg}^+(c, \text{send } c\ \text{shift} ; c \leftarrow c')$	(\downarrow)	$\text{msg}^-(c', \text{send } c\ \text{shift} ; c' \leftarrow c)$	(\uparrow)

The notion of observational equivalence we need does not observe “nontermination”, that is, it only compares messages that are actually received. Since messages can flow in two directions, we need to observe messages that arrive at either end. We therefore do *not* require, as is typical for bisimulation, that if one configuration takes a step, another configuration can also take a step. Instead we say if both configurations send an externally visible message, then the messages must be equivalent.

Supposing $\Gamma \vdash \mathcal{C} : \Delta$ and $\Gamma \vdash \mathcal{D} :: \Delta$, we write $\Gamma \vdash \mathcal{C} \sim \mathcal{D} :: \Delta$ for our notion of observational equivalence. It is the largest relation satisfying that $\Gamma \vdash \mathcal{C} \sim \mathcal{D} : \Delta$ implies

1. If $\Gamma' \vdash \text{msg}^+(c, P) :: \Gamma$ then $\Gamma' \vdash (\text{msg}^+(c, P), \mathcal{C}) \sim (\text{msg}^+(c, P), \mathcal{D}) :: \Delta$.
2. If $\Delta \vdash \text{msg}^-(c, P) :: \Delta'$ then $\Gamma \vdash (\mathcal{C}, \text{msg}^-(c, P)) \sim (\mathcal{D}, \text{msg}^-(c, P)) :: \Delta'$.
3. If $\mathcal{C} = (\mathcal{C}', \text{msg}^+(c, P))$ with $\Gamma \vdash \mathcal{C}' :: \Delta'_1$ and $\Delta'_1 \vdash \text{msg}^+(c, P) :: \Delta$ and $\mathcal{D} = (\mathcal{D}', \text{msg}^+(c, Q))$ with $\Gamma \vdash \mathcal{D}' :: \Delta'_2$ and $\Delta'_2 \vdash \text{msg}^+(c, Q) :: \Delta$ then $\Delta'_1 = \Delta'_2 = \Delta'$ and $P = Q$ and $\Gamma \vdash \mathcal{C}' \sim \mathcal{D}' :: \Delta'$.
4. If $\mathcal{C} = (\text{msg}^-(c, P), \mathcal{C}')$ with $\Gamma \vdash \text{msg}^-(c, P) :: \Gamma'_1$ and $\Gamma'_1 \vdash \mathcal{C}' :: \Delta$ and $\mathcal{D} = (\text{msg}^-(c, Q), \mathcal{D}')$ with $\Gamma \vdash \text{msg}^-(c, Q) :: \Gamma'_2$ and $\Gamma'_2 \vdash \mathcal{D}' :: \Delta$ then $\Gamma'_1 = \Gamma'_2 = \Gamma'$ and $P = Q$ and $\Gamma' \vdash \mathcal{C}' \sim \mathcal{D}' :: \Delta$.
5. If $\mathcal{C} \longrightarrow \mathcal{C}'$ then $\Gamma \vdash \mathcal{C}' \sim \mathcal{D} :: \Delta$.
6. If $\mathcal{D} \longrightarrow \mathcal{D}'$ then $\Gamma \vdash \mathcal{C} \sim \mathcal{D}' :: \Delta$.

Clauses (1) and (2) correspond to absorbing a message into a configuration, which may later be received by a process according to clauses (5) and (6).

Clauses (3) and (4) correspond to observing messages, either by a client (clause (3)) or provider (clause (4)).

In clause (3) we take advantage of the property that a new continuation channel in the message P (one that does not appear already in Γ) is always chosen fresh when created, so we can consistently (and silently) rename it in \mathcal{C}' , Δ'_1 , and P (and \mathcal{D}' , Δ'_2 , and Q , respectively). This slight of hand allows us to match up the context and messages exactly. An analogous remark applies to clause (4). A more formal description would match up the contexts and messages modulo two renaming substitution which allow us to leave Γ and Δ fixed.

Clauses (5) and (6) make sense because a transition never changes the interface to a configuration, except when executing a forwarding $\text{proc}(a, a \leftarrow b)$ which substitutes b for a in the remaining configuration. We can absorb this renaming into the renaming substitution. Cut creates a new channel, which remains internal since it is linear and will have one provider and one client within the new configuration. Unfortunately, our notation is already somewhat unwieldy

and carrying additional renaming substitutions further obscures matters. We therefore omit them in this presentation.

We now need to define a relation \sim_M such that (a) it satisfies the closure conditions of \sim and is therefore an observational equivalence, and (b) allows us to conclude that monitors satisfying our judgment are partial identities. Unfortunately, the theorem is rather complex, so we will walk the reader through a sequence of generalizations that account for various phenomena.

The $\oplus, \&$ Fragment. For this fragment, we have no value variables, nor are we passing channels. Then the top-level properties we would like to show are

- (1⁺) If $(y : A^+) ; \cdot ; \cdot \vdash P :: (x : A^+)[\]$
 then $y : A^+ \vdash \text{proc}(x, x \leftarrow y) \sim_M P :: (x : A^+)$
 (1⁻) If $(y : A^-) ; \cdot ; \cdot \vdash P :: (x : A^-)$
 then $y : A^- \vdash \text{proc}(x, x \leftarrow y) \sim_M P :: (x : A^-)$

Of course, asserting that $\text{proc}(x, x \leftarrow y) \sim_M P$ will be insufficient, because this relation is not closed under the conditions of observational equivalence. For example, if we add a message along y to both sides, P will change its state once it receives the message, and the queue will record that this message still has to be sent. To generalize this, we need to define the queue that corresponds to a sequence of messages. First, a single message:

Message to client of c		Message to provider of c	
$\langle\langle \text{msg}^+(c, c.k ; c \leftarrow c') \rangle\rangle$	$= k \quad (\oplus)$	$\langle\langle \text{msg}^-(c', c.k ; c' \leftarrow c) \rangle\rangle$	$= k \quad (\&)$
$\langle\langle \text{msg}^+(c, \text{send } c \ d ; c \leftarrow c') \rangle\rangle$	$= d \quad (\otimes)$	$\langle\langle \text{msg}^-(c', \text{send } c \ d ; c' \leftarrow c) \rangle\rangle$	$= d \quad (\neg\otimes)$
$\langle\langle \text{msg}^+(c, \text{close } c) \rangle\rangle$	$= \text{end} \quad (\mathbf{1})$		
$\langle\langle \text{msg}^+(c, \text{send } c \ v ; c \leftarrow c') \rangle\rangle$	$= v \quad (\exists)$	$\langle\langle \text{msg}^-(c', \text{send } c \ v ; c' \leftarrow c) \rangle\rangle$	$= v \quad (\forall)$
$\langle\langle \text{msg}^+(c, \text{send } c \ \text{shift} ; c \leftarrow c') \rangle\rangle$	$= \text{shift} \quad (\downarrow)$	$\langle\langle \text{msg}^-(c', \text{send } c \ \text{shift} ; c' \leftarrow c) \rangle\rangle$	$= \text{shift} \quad (\uparrow)$

We extend this to message sequences with $\langle\langle \cdot \rangle\rangle = (\cdot)$ and $\langle\langle \mathcal{E}_1, \mathcal{E}_2 \rangle\rangle = \langle\langle \mathcal{E}_1 \rangle\rangle \cdot \langle\langle \mathcal{E}_2 \rangle\rangle$, provided $\Delta_0 \vdash \mathcal{E}_1 : \Delta_1$ and $\Delta_1 \vdash \mathcal{E}_2 : \Delta_2$.

Then we build into the relation that sequences of messages correspond to the queue.

- (2⁺) If $(y : B^+) ; \cdot ; \cdot ; \cdot \vdash P :: (x : A^+)[\langle\langle \mathcal{E} \rangle\rangle]$ then $y : B^+ \vdash \mathcal{E} \sim_M \text{proc}(x, P) :: (x : A^+)$.
 (2⁻) If $(y : B^-) ; \cdot ; \cdot ; \cdot \vdash P :: (x : A^-)$ then $y : B^- \vdash \mathcal{E} \sim_M \text{proc}(x, P) :: (x : A^-)$.

When we add shifts the two propositions become mutually dependent, but otherwise they remain the same since the definition of $\langle\langle \mathcal{E} \rangle\rangle$ is already general enough. But we need to generalize the type on the opposite side of queue to be either positive or negative, because it switches polarity after a shift has been received. Similarly, the channel might terminate when receiving $\mathbf{1}$, so we also need to allow ω , which is either empty or of the form $y : B$.

- (3⁺) If $\omega ; \cdot ; \cdot ; \cdot \vdash P :: (x : A^+)[\langle\langle \mathcal{E} \rangle\rangle]$ then $\omega \vdash \mathcal{E} \sim_M \text{proc}(x, P) :: (x : A^+)$.
 (3⁻) If $(y : B^-) ; \cdot ; \cdot ; \cdot \vdash P :: (x : A)$ then $y : B^- \vdash \mathcal{E} \sim_M \text{proc}(x, P) :: (x : A)$.

Next, we can permit local state in the monitor (rules cut_1^+ and cut_1^-). The fact that neither of the two critical endpoints y and x , nor any (non-local) channels, can appear in the typing of the local process is key. That local process will evolve to a local configuration, but its interface will not change and it cannot access externally visible channels. So we generalize to allow a configuration \mathcal{D} that does not use any channels, and any channels it offers are used by P .

- (4⁺) If $\omega ; \cdot ; \cdot ; \Delta \vdash P :: [\langle\langle\mathcal{E}\rangle\rangle](x : A^+)$ and $\cdot \vdash \mathcal{D} :: \Delta$ then $\omega \vdash \mathcal{E} \sim_M \mathcal{D}, \text{proc}(x, P) :: [q](x : A^+)$.
- (4⁻) If $[\langle\langle\mathcal{E}\rangle\rangle](y : B^-) ; \cdot ; \cdot ; \Delta \vdash P :: (x : A)$ and $\cdot \vdash \mathcal{D} :: \Delta$ then $\Gamma, y : B^- \vdash \mathcal{E} \sim_M \mathcal{D}, \text{proc}(x, P) :: (x : A)$.

Next, we can allow value variables necessitated by the universal and existential quantifiers. Since they are potentially dependent, we need to apply the closing substitution σ to a number of components in our relation.

- (5⁺) If $\omega ; \Psi ; \cdot ; \Delta \vdash P :: [q](x : A^+)$ and $\sigma : \Psi$ and $q[\sigma] = \langle\langle\mathcal{E}\rangle\rangle$ and $\cdot \vdash \mathcal{D} :: \Delta[\sigma]$ then $\omega[\sigma] \vdash \mathcal{E} \sim_M \mathcal{D}, \text{proc}(x, P[\sigma]) :: (x : A^+[\sigma])$.
- (5⁻) If $[q](y : B^-) ; \Psi ; \cdot ; \Delta \vdash P :: (x : A)$ and $\sigma : \Psi$ and $q[\sigma] = \mathcal{E}$ and $\cdot \vdash \mathcal{D} :: \Delta[\sigma]$ then $y : B^-[\sigma] \vdash \mathcal{E} \sim_M \mathcal{D}, \text{proc}(x, P[\sigma]) :: (x : A[\sigma])$.

Breaking up the queue by spawning a sequence of monitors (rule cut_2^+ and cut_2^-) just comes down to the compositionality of the partial identity property. This is a new and separate way that two configurations might be in the \sim_M relation, rather than a replacement of a previous definition.

- (6) If $\omega \vdash \mathcal{E}_1 \sim_M \mathcal{D}_1 :: (z : C)$ and $(z : C) \vdash \mathcal{E}_2 \sim_M \mathcal{D}_2 :: (x : A)$ then $\omega \vdash (\mathcal{E}_1, \mathcal{E}_2) \sim_M (\mathcal{D}_1, \mathcal{D}_2) :: (x : A)$.

At this point, the only types that have not yet accounted for are \otimes and \multimap . If these channels were only “passed through” (without the four cut_3 rules), this would be rather straightforward. However, for higher-order channel-passing programs, a monitor must be able to spawn a monitor on a channel that it receives before sending on the monitored version. First, we generalize properties (5) to allow the context Γ of channels that may occur in the queue q and the process P , but that P may not interact with.

- (7⁺) If $\omega ; \Psi ; \Gamma ; \Delta \vdash P :: [q](x : A^+)$ and $\sigma : \Psi$ and $q[\sigma] = \langle\langle\mathcal{E}\rangle\rangle$ and $\cdot \vdash \mathcal{D} :: \Delta[\sigma]$ then $\Gamma[\sigma], \omega[\sigma] \vdash \mathcal{E} \sim_M \mathcal{D}, \text{proc}(x, P[\sigma]) :: (x : A^+[\sigma])$.
- (7⁻) If $[q](y : B^-) ; \Psi ; \Gamma ; \Delta \vdash P :: (x : A)$ and $\sigma : \Psi$ and $q[\sigma] = \mathcal{E}$ and $\cdot \vdash \mathcal{D} :: \Delta[\sigma]$ then $\Gamma[\sigma], y : B^-[\sigma] \vdash \mathcal{E} \sim_M \mathcal{D}, \text{proc}(x, P[\sigma]) :: (x : A[\sigma])$.

In addition we need to generalize property (6) into (8) and (9) to allow multiple monitors to run concurrently in a configuration.

- (8) If $\Gamma \vdash \mathcal{E} \sim_M \mathcal{D} :: \Delta$ then $(\Gamma', \Gamma) \vdash \mathcal{E} \sim_M \mathcal{D} :: (\Gamma', \Delta)$.
- (9) If $\Gamma_1 \vdash \mathcal{E}_1 \sim_M \mathcal{D}_1 :: \Gamma_2$ and $\Gamma_2 \vdash \mathcal{E}_2 \sim_M \mathcal{D}_2 :: \Gamma_3$ then $\Gamma_1 \vdash (\mathcal{E}_1, \mathcal{E}_2) \sim_M (\mathcal{D}_1, \mathcal{D}_2) :: \Gamma_3$.

At this point we can state the main theorem regarding monitors.

Theorem 1. *If $\Gamma \vdash \mathcal{E} \sim_M \mathcal{D} :: \Delta$ according to properties (7⁺), (7⁻), (8), and (9) then $\Gamma \vdash \mathcal{E} \sim \mathcal{D} :: \Delta$.*

Proof. By closure under conditions 1–6 in the definition of \sim .

By applying it as in equations (1⁺) and (1⁻), generalized to include value variables as in (5⁺) and (5⁻) we obtain:

Corollary 1. *If $[](b : A^-) ; \Psi \vdash P :: (a : A^-)$ or $(b : A^+) ; \Psi \vdash P :: [](a : A^+)$ then P is a partial identity process.*

5 Refinements as Contracts

In this section we show how to check refinement types dynamically using our contracts. We encode refinements as type casts, which allows processes to remain well-typed with respect to the non-refinement type system (Sect. 2). These casts are translated at run time to monitors that validate whether the cast expresses an appropriate refinement. If so, the monitors behave as identity processes; otherwise, they raise an alarm and abort. For refinement contracts, we can prove a safety theorem, analogous to the classic “Well-typed Programs Can’t be Blamed” [25], stating that if a monitor enforces a contract that casts from type A to type B , where A is a subtype of B , then this monitor will never raise an alarm.

5.1 Syntax and Typing Rules

We first augment messages and processes to include casts as follows. We write $\langle A \Leftarrow B \rangle^\rho$ to denote a cast from type B to type A , where ρ is a unique label for the cast. The cast for values is written as $(\langle \tau \Leftarrow \tau' \rangle^\rho)$. Here, the types τ' and τ are refinement types of the form $\{n:t \mid b\}$, where b is a boolean expression that expresses simple properties of the value n .

$$P ::= \dots \mid x \leftarrow \langle \tau \Leftarrow \tau' \rangle^\rho v ; Q \mid a:A \leftarrow \langle A \Leftarrow B \rangle^\rho b$$

Adding casts to forwarding is expressive enough to encode a more general cast $\langle A \Leftarrow B \rangle^\rho P$. For instance, the process $x:A \leftarrow \langle A \Leftarrow B \rangle^\rho P ; Q_x$ can be encoded as: $y:B \leftarrow P ; x:A \leftarrow \langle A \Leftarrow B \rangle^\rho y ; Q_x$.

One of the additional rules to type casts is shown below (both rules can be found in Fig. 6). We only allow casts between two types that are compatible with each other (written $A \sim B$), which is co-inductively defined based on the structure of the types (the full definition is omitted from the paper).

$$\frac{A \sim B}{\Psi ; b : B \vdash a \leftarrow \langle A \Leftarrow B \rangle^\rho b :: (a : A)} \text{id_cast}$$

5.2 Translation to Monitors

At run time, casts are translated into monitoring processes. A cast $a \leftarrow \langle A \Leftarrow B \rangle^\rho b$ is implemented as a monitor. This monitor ensures that the process that offers a service on channel b behaves according to the prescribed type A . Because of the typing rules, we are assured that channel b must adhere to the type B .

Figure 4 is a summary of all the translation rules, except recursive types. The translation is of the form: $\llbracket \langle A \Leftarrow B \rangle^\rho \rrbracket_{a,b} = P$, where A, B are types; the channels a and b are the offering channel and monitoring channel (respectively) for the resulting monitoring process P ; and ρ is a label of the monitor (i.e., the contract).

Note that this differs from blame labels for high-order functions, where the monitor carries two labels, one for the argument, and one for the body of the function. Here, the communication between processes is bi-directional. Though the blame is always triggered by processes sending messages to the monitor, our contracts may depend on a set of the values received so far, so it does not make sense to blame one party. Further, in the case of forwarding, the processes at either end of the channel are behaving according to the types (contracts) assigned to them, but the cast may forcefully connect two processes that have incompatible types. In this case, it is unfair to blame either one of the processes. Instead, we raise an alarm of the label of the failed contract.

The translation is defined inductively over the structure of the types. The **tensor** rule generates a process that first receives a channel (x) from the channel being monitored (b). It then spawns a new monitor (denoted by the **@monitor** keyword) to monitor channel x , making sure that it behaves as type A_1 , and passes the new monitor's offering channel y to channel a . Finally, the monitor continues to monitor b to make sure that it behaves as type A_2 . The **lolti** rule is similar to the **tensor** rule, except that the monitor first receives a channel from its offering channel. Similar to the higher-order function case, the argument position is contravariant, so the newly spawned monitor checks that the received channel behaves as type B_1 . The **exists** rule generates a process that first receives a value from the channel b , then checks the boolean condition e to validate the contract. The **forall** rule is similar, except the argument position is contravariant, so the boolean expression e' is checked on the offering channel a . The **with** rule generates a process that checks that all of the external choices promised by the type $\&\{\ell : A_\ell\}_{\ell \in I}$ are offered by the process being monitored. If a label in the set I is not implemented, then the monitor aborts with the label ρ . The **plus** rule requires that, for internal choices, the monitor checks that the monitored process only offers choices within the labels in the set $\oplus\{\ell : A_\ell\}_{\ell \in I}$.

For ease of explanation, we omit details for translating casts involving recursive types. Briefly, these casts are translated into recursive processes. For each pair of compatible recursive types A and B , we generate a unique monitor name f and record its type $f : \{A \Leftarrow B\}$ in a context Ψ . The translation algorithm needs to take additional arguments, including Ψ to generate and invoke the appropriate recursive process when needed. For instance, when generating the monitor process for $f : \{\text{list} \leftarrow \text{list}\}$, we follow the rule for translating internal

$$\begin{array}{c}
\frac{}{\llbracket \langle \mathbf{1} \Leftarrow \mathbf{1} \rangle^\rho \rrbracket_{a,b} = \text{wait } b; \text{close } a} \text{ one} \\
\\
\frac{}{\llbracket \langle A_1 \multimap A_2 \Leftarrow B_1 \multimap B_2 \rangle^\rho \rrbracket_{a,b} = } \multimap \quad \frac{}{\llbracket \langle A_1 \otimes A_2 \Leftarrow B_1 \otimes B_2 \rangle^\rho \rrbracket_{a,b} = } \otimes \\
\begin{array}{l}
x \leftarrow \text{recv } a; \\
@monitor \ y \leftarrow \llbracket \langle B_1 \Leftarrow A_1 \rangle^\rho \rrbracket_{y,x} \leftarrow x \\
\text{send } b \ y; \\
\llbracket \langle A_2 \Leftarrow B_2 \rangle^\rho \rrbracket_{a,b}
\end{array} \quad \begin{array}{l}
x \leftarrow \text{recv } b; \\
@monitor \ y \leftarrow \llbracket \langle A_1 \Leftarrow B_1 \rangle^\rho \rrbracket_{y,x} \leftarrow x \\
\text{send } a \ y; \\
\llbracket \langle A_2 \Leftarrow B_2 \rangle^\rho \rrbracket_{a,b}
\end{array} \\
\\
\frac{}{\llbracket \langle \forall \{n : \tau \mid e\}. A \Leftarrow \forall \{n : \tau' \mid e'\}. B \rangle^\rho \rrbracket_{a,b} = x \leftarrow \text{recv } a; } \forall \\
\text{assert } \rho e'(x) (\text{send } b \ x; \llbracket \langle A \Leftarrow B \rangle^\rho \rrbracket_{a,b}) \\
\\
\frac{}{\llbracket \langle \exists \{n : \tau \mid e\}. A \Leftarrow \exists \{n : \tau' \mid e'\}. B \rangle^\rho \rrbracket_{a,b} = x \leftarrow \text{recv } b; } \exists \\
\text{assert } \rho e(x) (\text{send } a \ x; \llbracket \langle A \Leftarrow B \rangle^\rho \rrbracket_{a,b}) \\
\\
\frac{\forall \ell, \ell \in I \cap J, \ a.\ell; \llbracket \langle A_\ell \Leftarrow B_\ell \rangle^\rho \rrbracket_{a,b} = Q_\ell \quad \forall \ell, \ell \in J \wedge \ell \notin I, \ Q_\ell = \text{abort } \rho}{\llbracket \langle \oplus \{\ell : A_\ell\}_{\ell \in I} \Leftarrow \oplus \{\ell : B_\ell\}_{\ell \in J} \rangle^\rho \rrbracket_{a,b} = \text{case } b \ (\ell \Rightarrow Q_\ell)_{\ell \in I}} \oplus \\
\\
\frac{\forall \ell, \ell \in I \cap J, \ b.\ell; \llbracket \langle A_\ell \Leftarrow B_\ell \rangle^\rho \rrbracket_{a,b} = Q_\ell \quad \forall \ell, \ell \in I \wedge \ell \notin J, \ Q_\ell = \text{abort } \rho}{\llbracket \langle \& \{\ell : A_\ell\}_{\ell \in I} \Leftarrow \& \{\ell : B_\ell\}_{\ell \in J} \rangle^\rho \rrbracket_{a,b} = \text{case } a \ (\ell \Rightarrow Q_\ell)_{\ell \in I}} \& \\
\\
\frac{}{\llbracket \langle \uparrow A \Leftarrow \uparrow B \rangle^\rho \rrbracket_{a,b} = } \uparrow \quad \frac{}{\llbracket \langle \downarrow A \Leftarrow \downarrow B \rangle^\rho \rrbracket_{a,b} = } \downarrow \\
\begin{array}{l}
\text{shift } \leftarrow \text{recv } b; \\
\text{send } a \ \text{shift}; \llbracket \langle A \Leftarrow B \rangle^\rho \rrbracket_{a,b}
\end{array} \quad \begin{array}{l}
\text{shift } \leftarrow \text{recv } a; \\
\text{send } b \ \text{shift}; \llbracket \langle A \Leftarrow B \rangle^\rho \rrbracket_{a,b}
\end{array}
\end{array}$$

Fig. 4. Cast translation

choices. For $\llbracket \langle \text{list} \Leftarrow \text{list} \rangle^\rho \rrbracket_{y,x}$ we apply the **cons** case in the translation to get $@monitor \ y \leftarrow f \leftarrow x$.

5.3 Metatheory

We prove two formal properties of cast-based monitors: safety and transparency.

Because of the expressiveness of our contracts, a general safety (or blame) theorem is difficult to achieve. However, for cast-based contracts, we can prove that a cast which enforces a subtyping relation, and the corresponding monitor, will not raise an alarm. We first define our subtyping relation in Fig. 5. In addition to the subtyping between refinement types, we also include label subtyping for our session types. A process that offers more external choices can always be used as a process that offers fewer external choices. Similarly, a process that offers fewer internal choices can always be used as a process that offers more internal choices (e.g., non-empty list can be used as a list). The subtyping rules for internal and external choices are drawn from work by Acay and Pfenning [1].

$$\begin{array}{c}
\frac{}{1 \leq 1} \quad 1 \quad \frac{A \leq A' \quad B \leq B'}{A \otimes B \leq A' \otimes B'} \otimes \quad \frac{A' \leq A \quad B \leq B'}{A \multimap B \leq A' \multimap B'} \multimap \\
\\
\frac{A_k \leq A'_k \text{ for } k \in J \quad J \subseteq I}{\oplus \{lab_k : A_k\}_{k \in J} \leq \oplus \{lab_k : A'_k\}_{k \in I}} \oplus \quad \frac{A_k \leq A'_k \text{ for } k \in J \quad I \subseteq J}{\& \{lab_k : A_k\}_{k \in J} \leq \& \{lab_k : A'_k\}_{k \in I}} \& \\
\\
\frac{A \leq B}{\downarrow A \leq \downarrow B} \downarrow \quad \frac{A \leq B}{\uparrow A \leq \uparrow B} \uparrow \quad \frac{A \leq B \quad \tau_1 \leq \tau_2}{\exists n : \tau_1.A \leq \exists n : \tau_2.B} \exists \quad \frac{A \leq B \quad \tau_2 \leq \tau_1}{\forall n : \tau_1.A \leq \forall n : \tau_2.B} \forall \\
\\
\frac{\text{def}(A) \leq \text{def}(B)}{A \leq B} \text{def} \quad \frac{\forall v:\tau, [v/x]b_1 \mapsto^* \text{true} \text{ implies } [v/x]b_2 \mapsto^* \text{true}}{\{x:\tau \mid b_1\} \leq \{x:\tau \mid b_2\}} \text{refine}
\end{array}$$

Fig. 5. Subtyping

For recursive types, we directly examine their definitions. Because of these recursive types, our subtyping rules are co-inductively defined.

We prove a safety theorem (i.e., well-typed casts do not raise alarms) via the standard preservation theorem. The key is to show that the monitor process generated from the translation algorithm in Fig. 4 is well-typed under a typing relation which guarantees that no **abort** state can be reached. We refer to the type system presented thus far in the paper as T , where monitors that may evaluate to **abort** can be typed. We define a stronger type system S which consists of the rules in T with the exception of the **abort** rule and we replace the **assert** rule with the **assert.strong** rule. The new rule for **assert**, which semantically verifies that the condition b is true using the fact that the refinements are stored in the context Ψ , is shown below. The two type systems are summarized in Fig. 6.

Theorem 2 (Monitors are well-typed). *Let Ψ be the context containing the type bindings of all recursive processes.*

1. $\Psi ; b : B \vdash_T \llbracket \langle A \Leftarrow B \rangle^\rho \rrbracket_{a,b}^\Psi :: (a : A)$.
2. If $B \leq A$, then $\Psi ; b : B \vdash_S \llbracket \langle A \Leftarrow B \rangle^\rho \rrbracket_{a,b}^\Psi :: (a : A)$.

Proof. The proof is by induction over the monitor translation rules. For 2, we need to use the sub-typing relation to show that (1) for the internal and external choice cases, no branches that include **abort** are generated; and (2) for the forall and exists cases, the **assert** never fails (i.e., the **assert.strong** rule applies). \square

As a corollary, we can show that when executing in a well-typed context, a monitor process translated from a well-typed cast will never raise an alarm.

Corollary 2 (Well-typed casts cannot raise alarms). $\vdash \mathcal{C} :: b : B$ and $B \leq A$ implies $\mathcal{C}, \text{proc}(a, \llbracket \langle A \Leftarrow B \rangle^\rho \rrbracket_{a,b}) \not\mapsto^* \text{abort}(\rho)$.

Finally, we prove that monitors translated from casts are partial identify processes.

Both System T and S

$$\begin{array}{c}
\frac{}{\Psi; b : A \vdash a \leftarrow b :: (a : A)} \text{id} \qquad \frac{\Psi; \Delta \vdash P :: (x : A) \quad x : A, \Delta' \vdash Q :: (c : C)}{\Psi; \Delta, \Delta' \vdash x:A \leftarrow P; Q :: (c : C)} \text{cut} \\
\\
\frac{\Psi; \Delta \vdash P :: (c : A^+)}{\Psi; \Delta \vdash \text{shift} \leftarrow \text{recv } c; P :: (c : \uparrow A^+)} \uparrow R \qquad \frac{\Psi; \Delta, c : A^+ \vdash Q :: (d : D)}{\Psi; \Delta, c : \uparrow A^+ \vdash \text{send } c \text{ shift}; Q :: (d : D)} \uparrow L \\
\\
\frac{\Psi; \Delta \vdash P :: (c : A^-)}{\Psi; \Delta \vdash \text{send } c \text{ shift}; P :: (c : \downarrow A^-)} \downarrow R \qquad \frac{\Psi; \Delta, c : A^- \vdash Q :: (d : D)}{\Psi; \Delta, c : \downarrow A^- \vdash \text{shift} \leftarrow \text{recv } c; Q :: (d : D)} \downarrow L \\
\\
\frac{}{\cdot \vdash \text{close } c :: (c : \mathbf{1})} \mathbf{1}R \qquad \frac{\Psi; \Delta \vdash Q :: (d : D)}{\Psi; \Delta, c : \mathbf{1} \vdash \text{wait } c; Q :: (d : D)} \mathbf{1}L \\
\\
\frac{\Psi; \Delta \vdash P :: (c : B)}{\Psi; \Delta, a : A \vdash \text{send } c \ a; P :: (c : A \otimes B)} \otimes R \qquad \frac{\Psi; \Delta, x : A, c : B \vdash Q :: (d : D)}{\Psi; \Delta, c : A \otimes B \vdash x \leftarrow \text{recv } c; Q :: (d : D)} \otimes L \\
\\
\frac{\Psi; \Delta, x : A \vdash P :: (c : B)}{\Psi; \Delta \vdash x \leftarrow \text{recv } c; P :: (c : A \multimap B)} \multimap R \qquad \frac{\Psi; \Delta, c : B \vdash Q :: (d : D)}{\Psi; \Delta, a : A, c : A \multimap B \vdash \text{send } c \ a; Q :: (d : D)} \multimap L \\
\\
\frac{\Psi; \Delta \vdash P_\ell :: (c : A_\ell) \text{ for every } \ell \in L}{\Psi; \Delta \vdash \text{case } c (\ell \Rightarrow P_\ell)_{\ell \in L} :: (c : \&\{\ell : A_\ell\}_{\ell \in L})} \&R \qquad \frac{k \in L \quad \Psi; \Delta, c : A_k \vdash Q :: (d : D)}{\Psi; \Delta, c : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c.k; Q :: (d : D)} \&L \\
\\
\frac{k \in L \quad \Psi; \Delta \vdash P :: (c : A_k)}{\Psi; \Delta \vdash c.k; P :: (c : \oplus\{\ell : A_\ell\}_{\ell \in L})} \oplus R \qquad \frac{\Psi; \Delta, c : A_\ell \vdash Q_\ell :: (d : D) \text{ for every } \ell \in L}{\Psi; \Delta, c : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \text{case } c (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (d : D)} \oplus L \\
\\
\frac{\Psi \vdash v : \tau \quad \Psi; \Delta \vdash P :: (c : [v/n]A)}{\Psi; \Delta \vdash \text{send } c \ v; P :: (c : \exists n:\tau. A)} \exists R \qquad \frac{\Psi, n:\tau; \Delta, c : A \vdash Q :: (d : D)}{\Psi; \Delta, c : \exists n:\tau. A \vdash n \leftarrow \text{recv } c; Q :: (d : D)} \exists L \\
\\
\frac{\Psi, n:\tau; \Delta \vdash P :: (c : A)}{\Psi; \Delta \vdash n \leftarrow \text{recv } c; P :: (c : \forall n:\tau. A)} \forall R \qquad \frac{\Psi \vdash v : \tau \quad \Psi; \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi; \Delta, c : \forall n:\tau. A \vdash \text{send } c \ v; Q :: (d : D)} \forall L \\
\\
\frac{\Psi \vdash v : \tau' \quad \Psi, x:\tau; \Delta \vdash Q :: (c : C) \quad \tau \sim \tau'}{\Psi; \Delta \vdash x \leftarrow \langle \tau \Leftarrow \tau' \rangle^\rho v; Q :: (c : C)} \text{val_cast} \qquad \frac{A \sim B}{\Psi; b : B \vdash a \leftarrow \langle A \Leftarrow B \rangle^\rho b :: (a : A)} \text{id_cast}
\end{array}$$

System T only

$$\frac{\Psi \vdash b : \text{bool} \quad \Psi; \Delta \vdash Q :: (x : A)}{\Psi; \Delta \vdash \text{assert } \rho \ b; Q :: (x : A)} \text{assert} \qquad \frac{}{\Psi; \Delta \vdash \text{abort } \rho :: (x : A)} \text{abort}$$

System S only

$$\frac{\Psi \Vdash b \text{ true} \quad \Psi; \Delta \vdash Q :: (x : A)}{\Psi; \Delta \vdash \text{assert } \rho \ b; Q :: (x : A)} \text{assert_strong}$$

Fig. 6. Typing process expressions

Theorem 3 (Casts are transparent).

$b : B \vdash \text{proc}(b, a \leftarrow b) \sim \text{proc}(a, \llbracket \langle A \Leftarrow B \rangle^\rho \rrbracket_{a,b}) :: (a : A).$

Proof. We just need to show that the translated process passes the partial identity checks. We can show this by induction over the translation rules and by applying the rules in Sect. 4. We note that rules in Sect. 4 only consider identical types; however, our casts only cast between two compatible types. Therefore, we

can lift A and B to their super types (i.e., insert abort cases for mismatched labels), and then apply the checking rules. This does not change the semantics of the monitors.

6 Related Work

There is a rich body of work on higher-order contracts and the correctness of blame assignments in the context of the lambda calculus [2, 7, 8, 10, 16, 24, 25]. The contracts in these papers are mostly based on refinement or dependent types. Our contracts are more expressive than the above, and can encode refinement-based contracts. While our monitors are similar to reference monitors (such as those described by Schneider [19]), they have a few features that are not inherent to reference monitors such as the fact that our monitors are written in the target language. Our monitors are also able to monitor contracts in a higher-order setting by spawning a separate monitor for the sent/received channel.

Disney et al.'s [9] work, which investigates behavioral contracts that enforce temporal properties for modules, is closely related to our work. Our contracts (i.e., session types) also enforce temporal properties; the session types specify the order in which messages are sent and received by the processes. Our contracts can also make use of internal state, as those of Disney et al, but our system is concurrent, while their system does not consider concurrency.

Recently, gradual typing for two-party session-type systems has been developed [14, 20]. Even though this formalism is different from our contracts, the way untyped processes are gradually typed at run time resembles how we monitor type casts. Because of dynamic session types, their system has to keep track of the linear use of channels, which is not needed for our monitors.

Most recently, Melgratti and Padovani have developed chaperone contracts for higher-order session types [17]. Their work is based on a classic interpretation of session types, instead of an intuitionistic one like ours, which means that they do not handle spawning or forwarding processes. While their contracts also inspect messages passed between processes, unlike ours, they cannot model contracts which rely on the monitor making use of internal state (e.g., the parenthesis matching). They proved a blame theorem relying on the notion of locally correct modules, which is a semantic categorization of whether a module satisfies the contract. We did not prove a general blame theorem; instead, we prove a somewhat standard safety theorem for cast-based contracts.

The Whip system [27] addresses a similar problem as our prior work [15], but does not use session types. They use a dependent type system to implement a contract monitoring system that can connect services written in different languages. Their system is also higher order, and allows processes that are monitored by Whip to interact with unmonitored processes. While Whip can express dependent contracts, Whip cannot handle stateful contracts. Another distinguishing feature of our monitors is that they are partial identity processes encoded in the same language as the processes to be monitored.

7 Conclusion

We have presented a novel approach for contract-checking for concurrent processes. Our model uses partial identity monitors which are written in the same language as the original processes and execute transparently. We define what it means to be a partial identity monitor and prove our characterization correct. We provide multiple examples of contracts we can monitor including ones that make use of the monitor's internal state, ones that make use of the idea of probabilistic result checking, and ones that cannot be expressed as dependent or refinement types. We translate contracts in the refinement fragment into monitors, and prove a safety theorem for that fragment.

Acknowledgment. This research was supported in part by NSF grant CNS1423168 and a Carnegie Mellon University Presidential Fellowship.

References

1. Acay, C., Pfenning, F.: Intersections and unions of session types. In: Proceedings Eighth Workshop on Intersection Types and Related Systems, ITRS 2016, Porto, Portugal, pp. 4–19, 26 June 2016. <https://dx.doi.org/10.4204/EPTCS.242.3>
2. Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011) (2011). <https://doi.acm.org/10.1145/1570506.1570507>
3. Balzer, S., Pfenning, F.: Manifest sharing with session types. *Proc. ACM Program. Lang.* **1**(ICFP), 37:1–37:29 (2017). <https://doi.org/10.1145/3110281>
4. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_16
5. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. *Math. Struct. Comput. Sci.* **26**(3), 367–423 (2016)
6. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. *Inf. Comput.* **207**(10), 1044–1077 (2009). <https://doi.org/10.1016/j.ic.2008.11.006>
7. Dimoulas, C., Findler, R.B., Flanagan, C., Felleisen, M.: Correct blame for contracts: no more scapegoating. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 215–226. ACM, New York (2011). <https://doi.acm.org/10.1145/1926385.1926410>
8. Dimoulas, C., Tobin-Hochstadt, S., Felleisen, M.: Complete monitors for behavioral contracts. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 214–233. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_11
9. Disney, T., Flanagan, C., McCarthy, J.: Temporal higher-order contracts. In: 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011) (2011). <https://doi.acm.org/10.1145/2034773.2034800>
10. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, ICFP 2002, pp. 48–59. ACM, New York (2002). <https://doi.acm.org/10.1145/581478.581484>

11. Gay, S.J., Hole, M.: Subtyping for session types in the π -calculus. *Acta Informatica* **42**(2–3), 191–225 (2005). <https://doi.org/10.1007/s00236-005-0177-z>
12. Gommerstadt, H., Jia, L., Pfenning, F.: Session-typed concurrent contracts. Technical report CMU-CyLab-17-004, CyLab, Carnegie Mellon University, February 2018
13. Griffith, D.: Polarized Substructural Session Types. Ph.D. thesis, University of Illinois at Urbana-Champaign, April 2016
14. Igarashi, A., Thiemann, P., Vasconcelos, V.T., Wadler, P.: Gradual session types. *Proc. ACM Program. Lang.* **1**(ICFP), 38:1–38:28 (2017). <https://doi.org/10.1145/3110282>
15. Jia, L., Gommerstadt, H., Pfenning, F.: Monitors and blame assignment for higher-order session types. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*, pp. 582–594. ACM, New York (2016). <https://doi.acm.org/10.1145/2837614.2837662>
16. Keil, M., Thiemann, P.: Blame assignment for higher-order contracts with intersection and union. In: *20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)* (2015). <https://doi.acm.org/10.1145/2784731.2784737>
17. Melgratti, H., Padovani, L.: Chaperone contracts for higher-order sessions. *Proc. ACM Program. Lang.* **1**(ICFP), 35:1–35:29 (2017). <https://doi.org/10.1145/3110279>
18. Pfenning, F., Griffith, D.: Polarized substructural session types. In: Pitts, A. (ed.) *FoSSaCS 2015. LNCS*, vol. 9034, pp. 3–22. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46678-0_1
19. Schneider, F.B.: Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* **3**(1), 30–50 (2000). <https://doi.org/10.1145/353323.353382>
20. Thiemann, P.: Session types with gradual typing. In: Maffei, M., Tuosto, E. (eds.) *TGC 2014. LNCS*, vol. 8902, pp. 144–158. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45917-1_10
21. Thiemann, P., Vasconcelos, V.T.: Context-free session types. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016*, pp. 462–475. ACM, New York (2016). <https://acm.doi.org/10.4230/LIPIcs.ECOOP.2016.9>
22. Toninho, B.: A Logical Foundation for Session-based Concurrent Computation. Ph.D. thesis, Carnegie Mellon University and New University of Lisbon (2015)
23. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: a monadic integration. In: Felleisen, M., Gardner, P. (eds.) *ESOP 2013. LNCS*, vol. 7792, pp. 350–369. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_20
24. Wadler, P.: A complement to blame. In: *1st Summit on Advances in Programming Languages (SNAPL 2015)* (2015). <https://doi.acm.org/10.4230/LIPIcs.SNAPL.2015.309>
25. Wadler, P., Findler, R.B.: Well-typed programs can’t be blamed. In: Castagna, G. (ed.) *ESOP 2009. LNCS*, vol. 5502, pp. 1–16. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_1
26. Wasserman, H., Blum, M.: Software reliability via run-time result-checking. *J. ACM* **44**(6), 826–849 (1997). <https://doi.org/10.1145/268999.269003>
27. Waye, L., Chong, S., Dimoulas, C.: Whip: higher-order contracts for modern services. *Proc. ACM Program. Lang.* **1**(ICFP), 36:1–36:28 (2017). <https://doi.org/10.1145/3110280>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems

Malte Viering^{1(✉)}, Tzu-Chun Chen¹, Patrick Eugster^{1,2,3}, Raymond Hu⁴,
and Lukasz Ziarek⁵

¹ Department of Computer Science, TU Darmstadt, Darmstadt, Germany
viering@dsp.tu-darmstadt.de

² Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland

³ Department of Computer Science, Purdue University, West Lafayette, USA

⁴ Department of Computing, Imperial College London, London, UK

⁵ Department of Computer Science and Engineering, SUNY Buffalo, Buffalo, USA

Abstract. A key requirement for many distributed systems is to be resilient toward partial failures, allowing a system to progress despite the failure of some components. This makes programming of such systems daunting, particularly in regards to avoiding inconsistencies due to failures and asynchrony. This work introduces a formal model for crash failure handling in asynchronous distributed systems featuring a lightweight coordinator, modeled in the image of widely used systems such as ZooKeeper and Chubby. We develop a typing discipline based on multiparty session types for this model that supports the specification and static verification of multiparty protocols with explicit failure handling. We show that our type system ensures subject reduction and progress in the presence of failures. In other words, in a well-typed system even if some participants crash during execution, the system is guaranteed to progress in a consistent manner with the remaining participants.

1 Introduction

Distributed Programs, Partial Failures, and Coordination. Developing programs that execute across a set of physically remote, networked processes is challenging. The correct operation of a *distributed program* requires correctly designed protocols by which concurrent processes interact asynchronously, and correctly implemented processes according to their roles in the protocols. This becomes particularly challenging when distributed programs have to be resilient to *partial failures*, where some processes crashes while others remain operational. Partial failures affect both *safety* and *liveness* of applications. Asynchrony is the key

Financially supported by ERC grant FP7-617805 “LiVeSoft - Lightweight Verification of Software”, NSF grants CNS-1405614 and IIS-1617586, and EPSRC EP/K034413/1 and EP/K011715/1.

© The Author(s) 2018

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 799–826, 2018.

https://doi.org/10.1007/978-3-319-89884-1_28

issue, resulting in the inability to distinguish slow processes from failed ones. In general, this makes it impossible for processes to reach agreement, even when only a single process can crash [19].

In practice, such impasses are overcome by making appropriate assumptions for the considered infrastructure and applications. One common approach is to assume the presence of a highly available *coordination service* [26] – realized using a set of replicated processes large enough to survive common rates of process failures (e.g., 1 out of 3, 2 out of 5) – and delegating critical decisions to this service. While this *coordinator model* has been in widespread use for many years (cf. *consensus service* [22]), the advent of cloud computing has recently brought it further into the mainstream, via instances like Chubby [4] and ZooKeeper [26]. Such systems are used not only by end applications but also by a variety of frameworks and middleware systems across the layers of the protocol stack [11, 20, 31, 40].

Typing Disciplines for Distributed Programs. Typing disciplines for distributed programs is a promising and active research area towards addressing the challenges in the correct development of distributed programs. See Hüttel et al. [27] for a broad survey. *Session types* are one of the established typing disciplines for message passing systems. Originally developed in the π -calculus [23], these have been later successfully applied to a range of practical languages, e.g., Java [25, 41], Scala [39], Haskell [34, 38], and OCaml [28, 37]. *Multiparty session types* (MPSTs) [15, 24] generalize session types beyond two participants. In a nutshell, a standard MPST framework takes (1) a specification of the whole multiparty message protocol as a *global type*; from which (2) *local types*, describing the protocol from the perspective of each participant, are derived; these are in turn used to (3) statically *type check* the I/O actions of endpoint programs implementing the session participants. A well-typed system of session endpoint programs enjoys important safety and liveness properties, such as *no reception errors* (only expected messages are received) and *session progress*. A basic intuition behind MPSTs is that the design (i.e., restrictions) of the type language constitutes a class of distributed protocols for which these properties can be statically guaranteed by the type system.

Unfortunately, *no* MPST work supports protocols for asynchronous distributed programs dealing with *partial failures due to process crashes*, so the aforementioned properties no longer hold in such an event. Several MPST works have treated communication patterns based on *exception messages* (or *interrupts*) [6, 7, 16]. In these works, such messages may convey exceptional states in an *application* sense; from a protocol compliance perspective, however, these messages are the same as any other message communicated during a *normal* execution of the session. This is in contrast to *process* failures, which may invalidate already in-transit (*orphan*) messages, and where the task of agreeing on the concerted handling of a crash failure is itself prone to such failures.

Outside of session types and other type-based approaches, there have been a number of advances on verifying fault tolerant distributed protocols and applications (e.g., based on model checking [29], proof assistants [44]); however, little

work exists on providing direct compile-time support for *programming* such applications in the spirit of MPSTs.

Contributions and Challenges. This paper puts forward a new typing discipline for safe specification and implementation of distributed programs prone to process crash failures based on MPSTs. The following summarizes the key challenges and contributions.

Multiparty session calculus with coordination service. We develop an extended multiparty session calculus as a formal model of processes prone to crash failures in asynchronous message passing systems. Unlike standard session calculi that reflect only “minimal” networking infrastructures, our model introduces a practically-motivated *coordinator* artifact and explicit, asynchronous messages for run-time crash notifications and failure handling.

MPSTs with explicit failure handling. We introduce new global and local type constructs for *explicit failure handling*, designed for specifying protocols tolerating partial failures. Our type system carefully reworks many of the key elements in standard MPSTs to manage the intricacies of handling crash failures. These include the well-formedness of failure-prone global types, and the crucial *coherence* invariant on MPST typing environments to reflect the notion of system consistency in the presence of crash failures and the resulting errors. We show safety and progress for a well-typed MPST session despite potential failures.

To fit our model to practice, we introduce programming constructs similar to well-known and intuitive exception handling mechanisms, for handling concurrent and asynchronous process crash failures in sessions. These constructs serve to integrate user-level session control flow in endpoint processes and the underlying communications with the coordination service, used by the target applications of our work to outsource critical failure management decisions (see Fig. 1). It is important to note that the coordinator does *not* magically solve all problems. Key design challenges are to ensure that communication with it is fully asynchronous as in real-life, and that it is involved only in a “minimal” fashion. Thus we treat the coordinator as a first-class, asynchronous network artifact, as opposed to a convenient but impractical global “oracle” (cf. [6]), and our operational semantics of multiparty sessions remains primarily *choreographic* in the original spirit of distributed MPSTs, unlike works that resort to a centralized *orchestrator* to conduct all actions [5, 8]. As depicted in Fig. 1, application-specific communication does not involve the coordinator. Our model lends itself to common practical scenarios where processes monitor each other in a peer-based fashion to detect failures, and rely on a coordinator only to establish agreement on which processes have failed, and when.

A long version of this paper is available online [43]. The long version contains: full formal definitions, full proofs, and a prototype implementation in Scala.

Example. As a motivating example, Fig. 2 gives a global formal specification for a big data streaming task between a distributed file system (DFS) *dfs*, and two

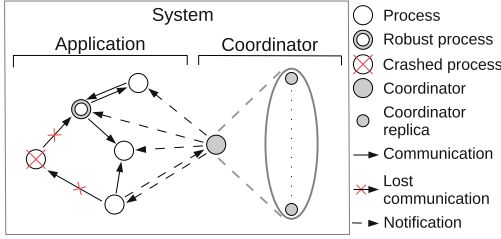


Fig. 1. Coordinator model for asynchronous distributed systems. The coordinator is implemented by replicated processes (internals omitted).

$$\begin{aligned}
 [dfs]G = & \mathbf{t}(\mu t. \\
 & dfs \rightarrow w_1 \ l_{d_1}(S). dfs \rightarrow w_2 \ l_{d_2}(S). \\
 & w_1 \rightarrow dfs \ l_{r_1}(S'). w_2 \rightarrow dfs \ l_{r_2}(S'). t \\
) & \mathbf{h}(\\
 & \{w_1\} : \mu t'. dfs \rightarrow w_2 \ l'_{d_1}(S). \\
 & w_2 \rightarrow dfs \ l'_{r_1}(S'). t', \\
 & \{w_2\} : \dots, \{w_1, w_2\} : \text{end})
 \end{aligned}$$

Fig. 2. Global type for a big data streaming task with failure handling capabilities.

workers $w_{1,2}$. The DFS streams data to two workers, which process the data and write the result back. Most DFSs have built-in fault tolerance mechanisms [20], so we consider dfs to be *robust*, denoted by the annotation $[dfs]$; the workers, however, may individually fail. In the *try-handle* construct $\mathbf{t}(\dots)\mathbf{h}(\dots)$, the *try-block* $\mathbf{t}(\dots)$ gives the *normal* (i.e., failure-free) flow of the protocol, and $\mathbf{h}(\dots)$ contains the explicit *handlers* for potential crashes. In the try-block, the workers receive data from the DFS ($dfs \rightarrow w_i$), perform local computations, and send back the result ($w_i \rightarrow dfs$). If a worker crashes ($\{w_i\} : \dots$), the other worker will also take over the computation of the crashed worker, allowing the system to still produce a valid result. If both workers crash (by any interleaving of their concurrent crash events), the global type specifies that the DFS should safely terminate its role in the session.

We shall refer to this basic example, that focuses on the new failure handling constructs, in explanations in later sections. We also give many further examples throughout the following sections to illustrate the potential session errors due to failures exposed by our model, and how our framework resolves them to recover MPST safety and progress.

Roadmap. Section 2 describes the adopted system and failure model. Section 3 introduces global types for guiding failure handling. Section 4 introduces our process calculus with failure handling capabilities and a coordinator. Section 5 introduces local types, derived from global types by projection. Section 6 describes typing rules, and defines *coherence* of session environments with respect to end-point crashes. Section 7 states properties of our model. Section 8 discusses related work. Sect. 9 draws conclusions.

2 System and Failure Model

In distributed systems care is required to avoid partial failures affecting liveness (e.g., waiting on messages from crashed processes) or safety (e.g., when processes manage to communicate with some peers but not others before crashing) properties of applications. Based on the nature of the infrastructure and application,

appropriate *system and failure models* are chosen along with judiciously made assumptions to overcome such impasses in practice.

We pinpoint the key characteristics of our model, according to our practical motivations and standard distributed systems literature, that shape the design choices we make later for the process calculus and types. As it is common we augment our system with a *failure detector* (FD) to allow for distinguishing slow and failed processes. The advantage of the FD (1) in terms of reasoning is that it concentrates all assumptions to solve given problems and (2) implementation-wise it yields a single main module where time-outs are set and used.

Concretely we make the following assumptions on failures and the system:

- (1) **Crash-stop failures:** Application processes fail by crashing (halting), and do not recover.
- (2) **Asynchronous system:** Application processes and the network are asynchronous, meaning that there are no upper bounds on processes' relative speeds or message transmission delays.
- (3) **Reliable communication:** Messages transmitted between correct (i.e., non-failed) participants are eventually received.
- (4) **Robust coordinator:** The coordinator (coordination service) is permanently available.
- (5) **Asynchronous reliable failure detection:** Application processes have access to local FDs which eventually detect all failed peers and do not falsely suspect peers.

(1)–(3) are standard in literature on fault-tolerant distributed systems [19].

Note that processes can still recover but will not do so *within* sessions (or will not be re-considered for those). Other failure models, e.g., network partitions [21] or Byzantine failures [32], are subject of future work. The former are not tolerated by ZooKeeper et al., and the latter have often been argued to be a too generic failure model (e.g., [3]).

The assumption on the coordinator (4) implicitly means that the number of concomitant failures among the coordinator replicas is assumed to remain within a minority, and that failed replicas are replaced in time (to tolerate further failures). Without loss of validity, the coordinator internals can be treated as a blackbox. The final assumption (5) on failure detection is backed in practice by the concept of *program-controlled* crash [10], which consists in communicating decisions to disregard supposedly failed processes also to those processes, prompting them to reset themselves upon false suspicion. In practice systems can be configured to minimize the probability of such events, and by a “two-level” membership consisting in evicting processes from *individual* sessions (cf. recovery above) more quickly than from a system as a whole; several authors have also proposed network support to entirely avoid false suspicions (e.g., [33]).

These assumptions do not make handling of failures trivial, let alone mask them. For instance, the network can arbitrarily delay messages and thus reorder them with respect to their real sending times, and (so) different processes can detect failures at different points in time and in different orders.

(Basic type) $S ::= \text{bool} \mid \text{str} \mid \text{int}$
 (Global type) $G ::= p \rightarrow q\{l_i(S_i).G_i\}_{i \in I} \mid \mu t.G \mid t \mid \text{end} \mid \mathbf{t}(G_1)\mathbf{h}(H)^\kappa.G_2$
 (Handling env.) $H ::= F:G \mid H, H$ (Handler sig.) $F ::= \{p_i\}_{i \in I}$

Fig. 3. Syntax of global types with explicit handling of partial failures.

3 Global Types for Explicit Handling of Partial Failures

Based on the foundations of MPSTs, we develop *global types* to formalize specifications of distributed protocols with explicit handling of *partial failures due to role crashes*, simply referred to as *failures*. We present global types before introducing the process calculus to provide a high-level intuition of how failure handling works in our model.

The syntax of *global types* is depicted in Fig. 3. We use the following base notations: p, q, \dots for *role* (i.e., participant) names; l_1, l_2, \dots for message *labels*; and t, t', \dots for type variables. *Base types* S may range over, bool, int , etc.

Global types are denoted by G . We first summarize the constructs from standard MPST [15, 24]. A *branch* type $p \rightarrow q\{l_i(S_i).G_i\}_{i \in I}$ means that p can send to q *one* of the messages of type S_k with label l_k , where k is a member of the non-empty index set I . The protocol then proceeds according to the continuation G_k . When I is a singleton, we may simply write $p \rightarrow q\ l(S).G$. We use t for type variables and take an equi-recursive view, i.e., $\mu t.G$ and its unfolding $[\mu t.G/t]$ are equivalent. We assume type variable occurrences are bound and guarded (e.g., $\mu t.t$ is not permitted). **end** is for termination.

We now introduce our extensions for partial failure handling. A *try-handle* $\mathbf{t}(G_1)\mathbf{h}(H)^\kappa.G_2$ describes a “failure-atomic” protocol unit: all *live* (i.e., non-crashed) roles will eventually reach a consistent protocol state, despite any concurrent and asynchronous role crashes. The try-block G_1 defines the *default* protocol flow, and H is a *handling environment*. Each element of H maps a *handler signature* F , that specifies a set of *failed* roles $\{p_i\}_{i \in I}$, to a *handler body* specified by a G . The handler body G specifies how the live roles should proceed given the failure of roles F . The protocol then proceeds (for live roles) according to the continuation G_2 after the default block G_1 or failure handling defined in H has been completed as appropriate.

To simplify later technical developments, we annotate each try-handle term in a given G by a unique $\kappa \in \mathbb{N}$ that lexically identifies the term within G . These annotations may be assigned mechanically. As a short hand, we refer to the try-block and handling environment of a particular try-handle by its annotation; e.g., we use κ to stand for $\mathbf{t}(G_1)\mathbf{h}(H)^\kappa$. In the running examples (e.g., Fig. 2), if there exists only one try-handle, we omit κ for simplicity.

Top-Level Global Types and Robust Roles. We use the term *top-level* global type to mean the source protocol specified by a user, following a typical top-down interpretation of MPST frameworks [15, 24]. We allow top-level global types to be optionally annotated $[\tilde{p}]G$, where $[\tilde{p}]$ specifies a set of *robust* roles—i.e., roles

that can be assumed to never fail. In practice, a participant may be robust if it is replicated or is made inherently fault tolerant by other means (e.g., the participant that represents the distributed file system in Fig. 2).

Well-Formedness. The first stage of validation in standard MPSTs is to check that the top-level global type satisfies the supporting criteria used to ensure the desired properties of the type system. We first list basic syntactic conditions which we assume on any given G : (i) each F is non-empty; (ii) a role in a F cannot occur in the corresponding handler body (a failed role cannot be involved in the handling of its own failure); and (iii) every occurrence of a non-robust role p must be contained within a, possibly outer, try-handle that has a handler signature $\{p\}$ (the protocol must be able to handle its potential failure). Lastly, to simplify the presentation without loss of generality, we impose that separate branch types *not* defined in the same default block or handler body must have disjoint label sets. This can be implicitly achieved by combining label names with try-handle annotations.

Assuming the above, we define *well-formedness* for our extended global types. We write $G' \in G$ to mean that G' syntactically occurs in G (\in is reflexive); similarly for the variations $\kappa \in G$ and $\kappa \in \kappa'$. Recall κ is shorthand for $\mathbf{t}(G_1)\mathbf{h}(H)^\kappa$. We use a lookup function $\text{outer}_G(\kappa)$ for the set of all try-handles in G that enclose a given κ , including κ itself, defined by $\text{outer}_G(\kappa) = \{\kappa' \mid \kappa \in \kappa' \wedge \kappa' \in G\}$.

Definition 1 (Well-formedness). Let κ stand for $\mathbf{t}(G_1)\mathbf{h}(H)^\kappa$, and κ' for $\mathbf{t}(G'_1)\mathbf{h}(H')^{\kappa'}$. A global type G is *well-formed* if both of the following conditions hold. For all $\kappa \in G$:

1. $\forall F_1 \in \text{dom}(H). \forall F_2 \in \text{dom}(H). \exists \kappa' \in \text{outer}_G(\kappa) \text{ s.t. } F_1 \cup F_2 \in \text{dom}(H')$
2. $\nexists F \in \text{dom}(H). \exists \kappa' \in \text{outer}_G(\kappa). \exists F' \in \text{dom}(H') \text{ s.t. } \kappa' \neq \kappa \wedge F' \subseteq F$

The first condition asserts that for any two separate handler signatures of a handling environment of κ , there always exists a handler whose handler signature matches the union of their respective failure sets – this handler is either inside the handling environment of κ itself, or in the handling environment of an outer try-handle. This ensures that if roles are active in different handlers of the same try-handle then there is a handler whose signature corresponds to the union over the signatures of those different handlers. Example 2 together with Example 3 in Sect. 4 illustrate a case where this condition is needed. The second condition asserts that if the handling environment of a try-handle contains a handler for F , then there is no outer try-handle with a handler for F' such that $F' \subseteq F$. The reason for this condition is that in the case of *nested* try-handles, our communication model allows separate try-handles to start failure handling independently (the operational semantics will be detailed in the next section; see (TryHdl) in Fig. 6). The aim is to have the relevant roles eventually converge on performing the handling of the outermost try-handle, possibly by interrupting the handling of an inner try-handle. Consider the following example:

Example 1. $G = \mathbf{t}(\mathbf{t}(G')\mathbf{h}(\{p_1, p_2\} : G_1)^2)\mathbf{h}(\{p_1\} : G'_1)^1$ violates condition 2 because, when p_1 and p_2 both failed, the handler signature $\{p_1\}$ will still be triggered (i.e., the outer try-handle will eventually take over). It is not sensible to run G'_1 instead of G_1 (which is for the crashes of p_1 and p_2).

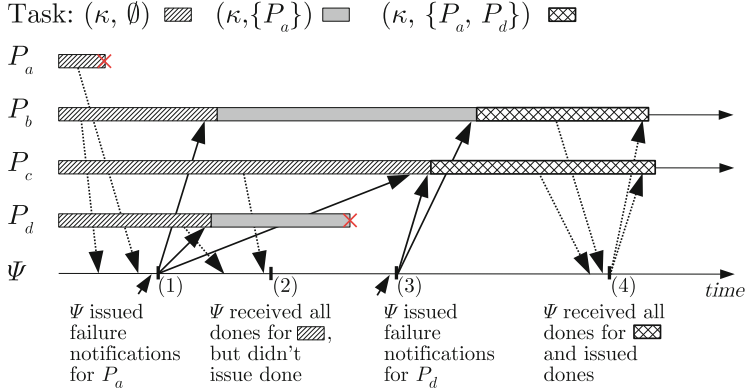


Fig. 4. Challenges under pure asynchronous interactions with a coordinator. Between time (1) and time (2), the task $\phi = (\kappa, \emptyset)$ is interrupted by the crash of P_a . Between time (3) and time (4), due to asynchrony and multiple crashes, P_c starts handling the crash of $\{P_a, P_d\}$ without handling the crash of $\{P_a\}$. Finally after (4) P_b and P_c finish their common task.

4 A Process Calculus for Coordinator-Based Failure Handling

Figure 4 depicts a scenario that can occur in practical asynchronous systems with coordinator-based failure handling through frameworks such as ZooKeeper (Sect. 2). Using this scenario, we first illustrate challenges, formally define our model, and then develop a safe type system.

The scenario corresponds to a global type of the form $\mathbf{t}(G)\mathbf{h}(\{P_a\} : G_a, \{P_a, P_d\} : G_{ad}, \dots)^\kappa$, with processes $P_{a..d}$ and a coordinator Ψ . We define a *task* to mean a unit of interactions, which includes failure handling behaviors. Initially all processes are collaborating on a task ϕ , which we label (κ, \emptyset) (identifying the task context, and the set of failed processes). The shaded boxes signify which task each process is working on. Dotted arrows represent notifications between processes and Ψ related to task completion, and solid arrows for failure notifications from Ψ to processes. During the scenario, P_a first fails, then P_d fails: the execution proceeds through failure handling for $\{P_a\}$ and $\{P_a, P_d\}$.

- (I) When P_b reaches the end of its part in ϕ , the application has P_b notify Ψ . P_b then remains in the context of ϕ (the continuation of the box after notifying) in consideration of other non-robust participants still working on ϕ — P_b may yet need to handle their potential failure(s).

(Expression)	$e ::= v \mid x \mid e + e \mid -e \mid \dots$	(Channel)	$c ::= s[p] \mid y$
(Process)	$P ::= a[p](y).P \mid c : \eta$	(Level)	$\phi ::= (\kappa, F)$
(Statement)	$\eta ::= t(\eta)h(H)^\phi.\eta \mid \underline{0} \mid 0 \mid p! l(e).\eta$ $\quad \mid p?\{l_i(x_i).\eta_i\}_{i \in I} \mid X\langle e \rangle$ $\quad \mid \text{def } D \text{ in } \eta \mid \text{if } e \text{ then } \eta \text{ else } \eta$	(Declaration)	$D ::= X(x) = \eta$
(Application)	$N ::= P \mid N \mid N \mid s : h$	(Handling)	$H ::= F : \eta \mid H, H$
(Message)	$m ::= \langle p, q, l(v) \rangle \mid \langle p, \text{crash } F \rangle \mid dn$	(Queue)	$h ::= \emptyset \mid h \cdot m$
(System)	$S ::= \Psi \blacklozenge N \mid (\nu s)S \mid S \mid S$	(Done)	$dn ::= \langle p, q \rangle^\phi$
(Context)	$E ::= t(E)h(H)^\phi.\eta \mid \text{def } D \text{ in } E \mid []$	(Coordinator)	$\Psi ::= G : (F, d)$
		(Done Queue)	$d ::= \emptyset \mid d \cdot dn$

Fig. 5. Grammar for processes, applications, systems, and evaluation contexts.

- (II) The processes of synchronizing on the completion of a task or performing failure handling *are themselves subject to failures* that may arise concurrently. In Fig. 4, all processes reach the end of ϕ (i.e., four dotted arrows from ϕ), but P_a fails. Ψ determines this failure and it initiates failure handling at time (1), while *done* notifications for ϕ continue to arrive asynchronously at time (2). The failure handling for crash of P_a is itself interrupted by the second failure at time (3).
- (III) Ψ can receive notifications that are no longer relevant. For example, at time (2), Ψ has received all *done* notifications for ϕ , but the failure of P_a has already triggered failure handling from time (1).
- (IV) Due to multiple concurrent failures, interacting participants may end up in different tasks: around time (2), P_b and P_d are in task $\phi' = (\kappa, \{P_a\})$, whereas P_c is still in ϕ (and asynchronously sending or receiving messages with the others). Moreover, P_c never executes ϕ' because of delayed notifications, so it goes from ϕ directly to $(\kappa, \{P_a, P_d\})$.

Processes. Figure 5 defines the grammar of processes and (distributed) applications. Expressions e, e_i, \dots can be values v, v_i, \dots , variables x, x_i, \dots , and standard operations. (Application) processes are denoted by P, P_i, \dots . An initialization $a[p](y).P$ agrees to play role p via shared name a and takes actions defined in P ; actions are executed on a session channel $c : \eta$, where c ranges over $s[p]$ (session name and role name) and session variables y ; η represents action statements.

A try-handle $t(\eta)h(H)^\phi$ attempts to execute the local action η , and can handle failures occurring therein as defined in the handling environment H , analogously to global types. H thus also maps a handler signature F to a handler body η defining how to handle F . Annotation $\phi = (\kappa, F)$ is composed of two elements: an identity κ of a *global* try-handle, and an indication of the *current* handler signature which can be empty. $F = \emptyset$ means that the default try-block is executing, whereas $F \neq \emptyset$ means that the handler body for F is executing. Term $\underline{0}$ only occurs in a try-handle during runtime. It denotes a *yielding* for a *notification* from a *coordinator* (introduced shortly).

Other statements are similar to those defined in [15, 24]. Term 0 represents an *idle* action. For convention, we omit 0 at the end of a statement. Action $p! l(e).\eta$ represents a sending action that sends p a label l with content e , then

it continues as η . Branching $p?\{l_i(x_i).\eta_i\}_{i \in I}$ represents a receiving action from p with several possible branches. When label l_k is selected, the transmitted value v is saved in x_k , and $\eta_k\{v/x_k\}$ continues. For convenience, when there is only one branch, the curly brackets are omitted, e.g., $c : p?l(x).P$ means there is only one branch $l(x)$. $X\langle e \rangle$ is for a statement variable with one parameter e , and $\text{def } D \text{ in } \eta$ is for recursion, where declaration D defines the recursive body that can be called in η . The conditional statement is standard.

The structure of processes ensures that failure handling is not interleaved between different sessions. However, we note that in standard MPSTs [15, 24], session interleaving must anyway be prohibited for the basic progress property. Since our aim will be to show progress, we disallow session interleaving within process bodies. Our model does allow parallel sessions at the top-level, whose actions may be concurrently interleaved during execution.

(Distributed) Systems. A (distributed) *system* in our programming framework is a composition of an application, which contains more than one process, and a coordinator (cf. Fig. 1). A system can be running within a private session \mathbf{s} , represented by $(\nu \mathbf{s})\mathcal{S}$, or $\mathcal{S} \mid \mathcal{S}'$ for systems running in different sessions independently and in parallel (i.e., no session interleaving). The job of the coordinator is to ensure that even in the presence of failures there is consensus on whether all participants in a given try-handle completed their local actions, or whether failures need to be handled, and which ones. We use $\Psi = G : (F, d)$ to denote a (robust) coordinator for the global type G , which stores in (F, d) the failures F that occurred in the application, and in d done notifications sent to the coordinator. The coordinator is denoted by ψ when viewed as a role.

A (distributed) *application*¹ is a process P , a parallel composition $N \mid N'$, or a global queue carrying messages $\mathbf{s} : h$. A global queue $\mathbf{s} : h$ carries a sequence of messages m , sent by participants in session \mathbf{s} . A message is either a regular message $\langle p, q, l(v) \rangle$ with label l and content v sent from p to q or a *notification*. A notification may contain the role of a coordinator. There are *done* and *failure* notifications with two kinds of done notifications dn used for coordination: $\langle p, \psi \rangle^\phi$ notifies ψ that p has finished its local actions of the try-handle ϕ ; $\langle \psi, p \rangle^\phi$ is sent from ψ to notify p that ψ has received all done notifications for the try-handle ϕ so that p shall end its current try-handle and move to its next task. For example, in Fig. 4 at time (4) the coordinator will inform P_b and P_c via $\langle \psi, P_b \rangle^{\langle \kappa, \{P_a, P_d\} \rangle} . \langle \psi, P_c \rangle^{\langle \kappa, \{P_a, P_d\} \rangle}$ that they can finish the try-handle $(\kappa, \{P_a, P_d\})$. Note that the appearance of $\langle \psi, p \rangle^\phi$ implies that the coordinator has been informed that all participants in ϕ have completed their local actions. We define two kinds of *failure* notifications: $\langle \psi, \text{crash } F \rangle$ notifies ψ that F occurred, e.g., $\{q\}$ means q has failed; $\langle p, \text{crash } F \rangle$ is sent from ψ to notify p about the failure F for possible handling. We write $\langle \tilde{p}, \text{crash } F \rangle$, where $\tilde{p} = p_1, \dots, p_n$ short for $\langle p_1, \text{crash } F \rangle \cdot \dots \cdot \langle p_n, \text{crash } F \rangle$; similarly for $\langle \psi, \tilde{p} \rangle^\phi$.

¹ Other works use the term *network* which is the reason why we use N instead of, e.g., A . We call it application to avoid confusion with the physical network which interconnects all processes as well as the coordinator.

$$\begin{array}{l}
a[p_1](y_1).P_1 \mid \dots \mid a[p_n](y_n).P_n \rightarrow \\
(\nu s)(G : (\emptyset, \emptyset) \blacklozenge P_1\{s[p_1]/y_1\} \mid \dots \mid P_n\{s[p_n]/y_n\} \mid s : \emptyset) \quad a : G \quad \textbf{(Link)} \\
s[p] : E[q! l(e).\eta] \mid s : h \rightarrow s[p] : E[\eta] \mid s : h \cdot \langle p, q, l(v) \rangle \quad e \Downarrow v \quad \textbf{(Snd)} \\
s[p] : E[q?\{l_i(x_i).\eta_i\}_{i \in I}] \mid s : \langle q, p, l_k(v_k) \rangle \cdot h \rightarrow \\
s[p] : E[\eta_k\{v_k/x_k\}] \mid s : h \quad k \in I \quad \textbf{(Rcv)} \\
s[p] : E[\text{def } X(x) = \eta \text{ in } X\langle e \rangle] \rightarrow s[p] : E[\text{def } X(x) = \eta \text{ in } \eta\{v/x\}] \quad e \Downarrow v \quad \textbf{(Rec)} \\
\frac{N_1 \equiv N_3 \rightarrow N_4 \equiv N_2}{N_1 \rightarrow N_2} \quad \frac{N_1 \rightarrow N_2}{N_1 \mid N \rightarrow N_2 \mid N} \quad \textbf{(Str, Par)} \\
\frac{N_1 \rightarrow N_2}{\psi \blacklozenge N_1 \rightarrow \psi \blacklozenge N_2} \quad \frac{\mathcal{S} \rightarrow \mathcal{S}'}{(\nu s)S \rightarrow (\nu s)S'} \quad \textbf{(Sys, New)} \\
N \mid s : h \rightarrow N \setminus s[p] : \eta \mid s : \text{remove}(h, p) \cdot \langle \psi, \text{crash } \{p\} \rangle \\
s[p] : \eta \text{ non-robust} \quad \textbf{(Crash)}
\end{array}$$

Fig. 6. Operational semantics of distributed applications, for local actions.

Following the tradition of other MPST works the global queue provides an abstraction for multiple FIFO queues, each queue being between two endpoints (cf. TCP) with no global ordering. Therefore $m_i \cdot m_j$ can be permuted to $m_j \cdot m_i$ in the global queue if the sender or the receiver differ. For example the following messages are permutable: $\langle p, q, l(v) \rangle \cdot \langle p, q', l(v) \rangle$ if $q \neq q'$ and $\langle p, q, l(v) \rangle \cdot \langle \psi, p \rangle^\phi$ and $\langle p, q, l(v) \rangle \cdot \langle q, \text{crash } F \rangle$. But $\langle \psi, p \rangle^\phi \cdot \langle p, \text{crash } F \rangle$ is not permutable, both have the same sender and receiver (ψ is the sender of $\langle p, \text{crash } F \rangle$).

Basic Dynamic Semantics for Applications. Figure 6 shows the operational semantics of applications. We use evaluation contexts as defined in Fig. 5. Context E is either a hole $[]$, a default context $t(E)h(H)^\phi.\eta$, or a recursion context $\text{def } D \text{ in } E$. We write $E[\eta]$ to denote the action statement obtained by filling the hole in $E[\cdot]$ with η .

Rule **(Link)** says that (local) processes who agree on shared name a , obeying to some protocol (global type), playing certain roles p_i represented by $a[p_i](y_i).P$, together will start a private session s ; this will result in replacing every variable y_i in P_i and, at the same time, creating a new global queue $s : \emptyset$, and appointing a coordinator $G : (\emptyset, \emptyset)$, which is novel in our work.

Rule **(Snd)** in Fig. 6 reduces a sending action $q! l(e)$ by emitting a message $\langle p, q, l(v) \rangle$ to the global queue $s : h$. Rule **(Rcv)** reduces a receiving action if the message arriving at its end is sent from the expected sender with an expected label. Rule **(Rec)** is for recursion. When the recursive body, defined inside η , is called by $X\langle e \rangle$ where e is evaluated to v , it reduces to the statement $\eta\{v/x\}$ which will again implement the recursive body. Rule **(Str)** says that processes which are structurally congruent have the same reduction. Processes, applications, and systems are considered modulo structural congruence, denoted by \equiv , along with α -renaming. Rule **(Par)** and **(Str)** together state that a parallel composition has a reduction if its sub-application can reduce. Rule **(Sys)** states that a system has a reduction if its application has a reduction, and **(New)**

says a reduction can proceed under a session. Rule (**Crash**) states that a process on channel $s[p]$ can fail at any point in time. (**Crash**) also adds a notification $\langle\psi, \text{crash } F\rangle$ which is sent to ψ (the coordinator). This is an abstraction for the failure detector described in Sect. 2 (5), the notification $\langle\psi, \text{crash } F\rangle$ is the first such notification issued by a participant based on its local failure detector. Adding the notification into the global queue instead of making the coordinator immediately aware of it models that failures are only detected eventually. Note that a failure is not annotated with a level because failures transcend all levels, and asynchrony makes it impossible to identify “where” exactly they occurred. As a failure is permanent it can affect multiple try-handles. The (**Crash**) rule does not apply to participants which are robust, i.e., that conceptually cannot fail (e.g., *dfs* in Fig. 2). Rule (**Crash**) removes channel $s[p]$ (the failed process) from application N , and removes messages and notifications delivered from, or heading to, the failed p by function $\text{remove}(h, p)$. Function $\text{remove}(h, p)$ returns a new queue after removing all regular messages and notifications that contain p , e.g., let $h = \langle p_2, p_1, l(v) \rangle \cdot \langle p_3, p_2, l'(v') \rangle \cdot \langle p_3, p_4, l'(v') \rangle \cdot \langle p_2, \psi \rangle^\phi \cdot \langle p_2, \text{crash } \{p_3\} \rangle \cdot \langle \psi, p_2 \rangle^\phi$ then $\text{remove}(h, p_2) = \langle p_3, p_4, l'(v') \rangle$. Messages are removed to model that in a real system send/receive does *not* constitute an atomic action.

Handling at Processes. Failure handling, defined in Fig. 7, is based on the observations that (i) a process that fails stays down, and (ii) multiple processes can fail. As a consequence a failure can trigger multiple failure handlers either because these handlers are in different (subsequent) try-handles or because of additional failures. Therefore a process needs to retain the information of *who* failed. For simplicity we do not model state at processes, but instead processes read but do not remove failure notifications from the global queue. We define $Fset(h, p)$ to return the union of failures for which there are notifications heading to p , i.e., $\langle p, \text{crash } F \rangle$, issued by the coordinator in queue h up to the first done notification heading to p :

Definition 2 (Union of Existing Failures $Fset(h, p)$)

$$Fset(\emptyset, p) = \emptyset \quad Fset(h, p) = \begin{cases} F \cup Fset(h', p) & \text{if } h = \langle p, \text{crash } F \rangle \cdot h' \\ \emptyset & \text{if } h = \langle \psi, p \rangle^\phi \cdot h' \\ Fset(h', p) & \text{otherwise} \end{cases}$$

In short, if the global queue is \emptyset , then naturally there are no failure notifications. If the global queue contains a failure notification sent from the coordinator, say $\langle p, \text{crash } F \rangle$, we collect the failure. If the global queue contains done notification $\langle \psi, p \rangle^\phi$ sent from the coordinator then *all* participants in ϕ have finished their local actions, which implies that the try-handle ϕ can be completed. Our failure handling semantics, (**TryHdl**), allows a try-handle $\phi = (\kappa, F)$ to handle different failures or sets of failures by allowing a try-handle to switch between different handlers. F thus denotes the current set of handled failures. For simplicity we refer to this as the *current(ly handled) failure set*. This is a slight abuse of terminology, done for brevity, as obviously failures are only detected with a

$$\begin{array}{c}
\frac{F' = \cup\{A \mid A \in \text{dom}(\mathbf{H}) \wedge F \subset A \subseteq \text{Fset}(h, p)\} \quad F': \eta' \in \mathbf{H}}{\text{s}[p] : E[\text{t}(\eta)\mathbf{h}(\mathbf{H})^{(\kappa, F')}. \eta''] \mid \text{s} : h \rightarrow \text{s}[p] : E[\text{t}(\eta')\mathbf{h}(\mathbf{H})^{(\kappa, F')}. \eta''] \mid \text{s} : h} \quad (\text{TryHdl}) \\
\text{s}[p] : E[\text{t}(0)\mathbf{h}(\mathbf{H})^\phi. \eta] \mid \text{s} : h \rightarrow \text{s}[p] : E[\text{t}(\underline{0})\mathbf{h}(\mathbf{H})^\phi. \eta] \mid \text{s} : h \cdot \langle p, \psi \rangle^\phi \quad (\text{SndDone}) \\
\frac{\langle \psi, p \rangle^\phi \in h}{\text{s}[p] : E[\text{t}(\underline{0})\mathbf{h}(\mathbf{H})^\phi. \eta] \mid \text{s} : h \rightarrow \text{s}[p] : E[\eta] \mid \text{s} : h \setminus \{\langle \psi, p \rangle^\phi\}} \quad (\text{RcvDone}) \\
\text{s}[p] : E[\eta] \mid \text{s} : \langle q, p, l(v) \rangle \cdot h \rightarrow \text{s}[p] : E[\eta] \mid \text{s} : h \quad l \notin \text{labels}(E[\eta]) \quad (\text{Cln}) \\
\frac{\langle \psi, p \rangle^\phi \in h \quad \phi \notin E[\eta]}{\text{s}[p] : E[\eta] \mid \text{s} : h \rightarrow \text{s}[p] : E[\eta] \mid \text{s} : h \setminus \langle \psi, p \rangle^\phi} \quad (\text{ClnDone})
\end{array}$$

Fig. 7. Operational semantics of distributed applications, for endpoint handling.

certain lag. The handling strategy for a process is to handle the—currently—largest set of failed processes that this process has been informed of and is able to handle. This largest set is calculated by $\cup\{A \mid A \in \text{dom}(\mathbf{H}) \wedge F \subset A \subseteq \text{Fset}(h, p)\}$, that selects all failure sets which are larger than the current one ($A \in \text{dom}(\mathbf{H}) \wedge F \subset A$) if they are also triggered by known failures ($A \subseteq \text{Fset}(h, p)$). Condition $F' : \eta' \in \mathbf{H}$ in **(TryHdl)** ensures that there exists a handler for F' . The following example shows how **(TryHdl)** is applied to switch handlers.

Example 2. Take h such that $\text{Fset}(h, p) = \{p_1\}$ and $\mathbf{H} = \{p_1\} : \eta_1, \{p_2\} : \eta_2, \{p_1, p_2\} : \eta_{12}$ in process $P = \text{s}[p] : \text{t}(\eta_1)\mathbf{h}(\mathbf{H})^{(\kappa, \{p_1\})}$, which indicates that P is handling failure $\{p_1\}$. Assume now one more failure occurs and results in a new queue h' such that $\text{Fset}(h', p) = \{p_1, p_2\}$. By **(TryHdl)**, the process acting at $\text{s}[p]$ is handling the failure set $\{p_1, p_2\}$ such that $P = \text{s}[p] : \text{t}(\eta_{12})\mathbf{h}(\mathbf{H})^{(\kappa, \{p_1, p_2\})}$ (also notice the η_{12} inside the try-block). A switch to only handling $\{p_2\}$ does not make sense, since, e.g., η_2 can contain p_1 . Figure 2 shows a case where the handling strategy differs according to the number of failures.

In Sect. 3 we formally define well-formedness conditions, which guarantee that if there exist two handlers for two different handler signatures in a try-handle, then a handler exists for their union. The following example demonstrates why such a guarantee is needed.

Example 3. Assume a slightly different P compared to the previous examples (no handler for the union of failures): $P = \text{s}[p] : E[\text{t}(\eta)\mathbf{h}(\mathbf{H})^{(\kappa, \emptyset)}]$ with $\mathbf{H} = \{p_1\} : \eta_1, \{p_2\} : \eta_2$. Assume also that $\text{Fset}(h, p) = \{p_1, p_2\}$. Here **(TryHdl)** will not apply since there is no failure handling for $\{p_1, p_2\}$ in P . If we would allow a handler for either $\{p_1\}$ or $\{p_2\}$ to be triggered we would have no guarantee that other participants involved in this try-handle will all select the same failure set. Even with a deterministic selection, i.e., all participants in that try-handle selecting the same handling activity, there needs to be a handler with handler signature $= \{p_1, p_2\}$ since it is possible that p_1 is involved in η_2 . Therefore the type system will ensure that there is a handler for $\{p_1, p_2\}$ either at this level or at an outer level.

(I) explains that a process finishing its default action (P_b) cannot leave its current try-handle (κ, \emptyset) immediately because other participants may fail (P_a failed). Below Eq. 1 also shows this issue from the perspective of semantics:

$$\begin{aligned} s[p] : \mathbf{t}(0)h(F : q!l(10).q?l'(x))^{(\kappa, \emptyset)}. \eta' \mid s[q] : \mathbf{t}(p?l(x').p!l'(x' + 10))h(H)^{(\kappa, F)}. \eta'' \\ \mid s : \langle q, \text{crash } F \rangle \cdot \langle p, \text{crash } F \rangle \cdot h \end{aligned} \quad (1)$$

In Eq. 1 the process acting on $s[p]$ ended its try-handle (i.e., the action is 0 in the try-block), and if $s[p]$ finishes its try-handle the participant acting on $s[q]$ which started handling F would be stuck.

To solve the issue, we use (**SndDone**) and (**RcvDone**) for completing a local try-handle with the help of a coordinator. The rule (**SndDone**) sends out a done notification $\langle p, \psi \rangle^\phi$ if the current action in ϕ is 0 and sets the action to $\underline{0}$, indicating that a done notification from the coordinator is needed for ending the try-handle.

Assume process on channel $s[p]$ finished its local actions in the try-block (i.e., as in Eq. 1 above), then by (**SndDone**), we have

$$\begin{aligned} (1) \rightarrow s : \langle q, \text{crash } F \rangle \cdot \langle p, \text{crash } F \rangle \cdot \langle p, \psi \rangle^{(\kappa, \emptyset)} \cdot h \mid \\ s[p] : \mathbf{t}(\underline{0})h(F : q!l(10).q?l'(x))^{(\kappa, \emptyset)}. \eta' \mid s[q] : \mathbf{t}(p?l(x').p!l'(x' + 10))h(H)^{(\kappa, F)}. \eta'' \end{aligned}$$

where notification $\langle p, \psi \rangle^{(\kappa, \emptyset)}$ is added to inform the coordinator. Now the process on channel $s[p]$ can still handle failures defined in its handling environment. This is similar to the case described in (II).

Rule (**RcvDone**) is the counterpart of (**SndDone**). Once a process receives a done notification for ϕ from the coordinator it can finish the try-handle ϕ and reduces to the continuation η . Consider Eq. 2 below, which is similar to Eq. 1 but we take a case where the try-handle can be reduced with (**RcvDone**). In Eq. 2 (**SndDone**) is applied:

$$\begin{aligned} s[p] : \mathbf{t}(\underline{0})h(F : q!l(10).q?l'(x))^{(\kappa, \emptyset)}. \eta' \mid \\ s[q] : \mathbf{t}(\underline{0})h(F : p?l(x').p!l'(x' + 10))^{(\kappa, \emptyset)}. \eta'' \mid s : h \end{aligned} \quad (2)$$

With $h = \langle \psi, q \rangle^{(\kappa, \emptyset)} \cdot \langle \psi, p \rangle^{(\kappa, \emptyset)} \cdot \langle q, \text{crash } F \rangle \cdot \langle p, \text{crash } F \rangle$ both processes can apply (**RcvDone**) and safely terminate the try-handle (κ, \emptyset) . Note that $Fset(h, p) = Fset(h, q) = \emptyset$ (by Definition 2), i.e., rule (**TryHdl**) can not be applied since a done notification suppresses the failure notification. Thus Eq. 2 will reduce to:

$$(2) \rightarrow^* s[p] : \eta' \mid s[q] : \eta'' \mid s : \langle q, \text{crash } F \rangle \cdot \langle p, \text{crash } F \rangle$$

It is possible that η' or η'' have handlers for F . Note that once a queue contains $\langle \psi, p \rangle^{(\kappa, \emptyset)}$, all non-failed process in the try-handle (κ, \emptyset) have sent done notifications to ψ (i.e. applied rule (**SndDone**)). The coordinator which will be introduced shortly ensures this.

$$\begin{array}{c}
\frac{\tilde{p} = \text{roles}(G) \setminus F' \quad F' = F \cup \{p\} \quad m = \langle \tilde{p}, \text{crash } \{p\} \rangle}{G : (F, d) \blacklozenge N \mid s : \langle \psi, \text{crash } \{p\} \rangle \cdot h \rightarrow G : (F', d) \blacklozenge N \mid s : h \cdot m} \text{ (F)} \\
\\
\frac{d' = d \cdot \langle p, \psi \rangle^\phi}{G : (F, d) \blacklozenge s : \langle p, \psi \rangle^\phi \cdot h \rightarrow G : (F, d') \blacklozenge s : h} \text{ (CollectDone)} \\
\\
\frac{\text{roles}(d, \phi) \supseteq \text{roles}(G, \phi) \setminus F \quad \forall F' \in \text{hdl}(G, \phi). (F' \not\subseteq F)}{G : (F, d) \blacklozenge s : h \rightarrow G : (F, \text{remove}(d, \phi)) \blacklozenge s : h \cdot \langle \psi, \text{roles}(G, \phi) \setminus F \rangle^\phi} \text{ (IssueDone)}
\end{array}$$

Fig. 8. Operational semantics for the coordinator.

Rule **(CIn)** removes a normal message from the queue if the label in the message does not exist in the target process, which can happen when a failure handler was triggered. The function $\text{labels}(\eta)$ returns all labels of receiving actions in η which are able to receive messages now or possible later. This removal based on the syntactic process is safe because in a global type separate branch types *not* defined in the same default block or handler body must have disjoint sets of labels (c.f., Sect. 3). Let $\phi \in P$ if try-handle ϕ appears inside P . Rule **(CInDone)** removes a done notification of ϕ from the queue if no try-handle ϕ exists, which can happen in case of nesting when a handler of an outer try-handle is triggered.

Handling at Coordinator. Figure 8 defines the semantics of the coordinator. We firstly give the auxiliary definition of $\text{roles}(G)$ which gives the set of *all* roles appearing in G .

In rule **(F)**, F represents the failures that the coordinator is aware of. This rule states that the coordinator collects and removes a failure notification $\langle \psi, \text{crash } p \rangle$ heading to it, retains this notification by $G : (F', d)$, $F' = F \cup \{p\}$, and issues failure notifications to all non-failed participants.

Rules **(CollectDone, IssueDone)**, in short inform all participants in $\phi = (\kappa, F)$ to finish their try-handle ϕ if the coordinator has received sufficient done notifications of ϕ and did not send out failure notifications that interrupt the task (κ, F) (e.g. see (III)). Rule **(CollectDone)** collects done notifications, i.e., $\langle p, \psi \rangle^\phi$, from the queue and retains these notification; they are used in **(IssueDone)**. For introducing **(IssueDone)**, we first introduce $\text{hdl}(G, (\kappa, F))$ to return a set of handler signatures which can be triggered with respect to the current handler:

Definition 3. $\text{hdl}(G, (\kappa, F)) = \text{dom}(H) \setminus \mathcal{P}(F)$ if $\mathbf{t}(G_0)\mathbf{h}(H)^\kappa \in G$ where $\mathcal{P}(F)$ represents a powerset of F .

Also, we abuse the function roles to collect the non-coordinator roles of ϕ in d , written $\text{roles}(d, \phi)$; similarly, we write $\text{roles}(G, \phi)$ where $\phi = (\kappa, F)$ to collect the roles appearing in the handler body F in the try-handle of κ in G . Remember that d only contains done notifications sent by participants.

Rule **(IssueDone)** is applied for some ϕ when conditions $\forall F' \in \text{hdl}(G, \phi). (F' \not\subseteq F)$ and $\text{roles}(d, \phi) \supseteq \text{roles}(G, \phi) \setminus F$ are both satisfied, where F contains all failures the coordinator is aware of. Intuitively, these two conditions ensure that (1) the coordinator only issues done notifications to the participants

in the try-handle ϕ if it did not send failure notifications which will trigger a handler of the try-handle ϕ ; (2) the coordinator has received all done notifications from all non-failed participants of ϕ . We further explain both conditions in the following examples, starting from condition $\forall F' \in hdl(G, \phi). (F' \not\subseteq F)$, which ensures no handler in ϕ can be triggered based on the failure notifications F sent out by the coordinator.

Example 4. Assume a process playing role p_i is $P_i = s[p_i] : t(\eta_i)h(H_i)^{\phi_i}$. Where $i \in \{1, 2, 3\}$ and $H_i = \{p_2\} : \eta_{i2}, \{p_3\} : \eta_{i3}, \{p_2, p_3\} : \eta_{i23}$ and the coordinator is $G : (\{p_2, p_3\}, d)$ where $t(\dots)h(H)^\kappa \in G$ and $dom(H) = dom(H_i)$ for any $i \in \{1, 2, 3\}$ and $d = \langle p_1, \psi \rangle^{(\kappa, \{p_2\})} . \langle p_1, \psi \rangle^{(\kappa, \{p_2, p_3\})} . d'$. For any ϕ in d , the coordinator checks if it has issued any failure notification that can possibly trigger a new handler of ϕ :

1. For $\phi = (\kappa, \{p_2\})$ the coordinator issued failure notifications that can interrupt a handler since

$$hdl(G, (\kappa, \{p_2\})) = dom(H) \setminus \mathcal{P}(\{p_2\}) = \{\{p_3\}, \{p_2, p_3\}\}$$

and $\{p_2, p_3\} \subseteq \{p_2, p_3\}$. That means the failure notifications issued by the coordinator, i.e., $\{p_2, p_3\}$, can trigger the handler with signature $\{p_2, p_3\}$. Thus the coordinator will not issue done notifications for $\phi = (\kappa, \{p_2\})$. A similar case is visualized in Fig. 4 at time (2).

2. For $\phi = (\kappa, \{p_2, p_3\})$ the coordinator did not issue failure notifications that can interrupt a handler since

$$hdl(G, (\kappa, \{p_2, p_3\})) = dom(H) \setminus \mathcal{P}(\{p_2, p_3\}) = \emptyset$$

so that $\forall F' \in hdl(G, (\kappa, \{p_2, p_3\})). (F' \not\subseteq \{p_2, p_3\})$ is true. The coordinator will issue done notifications for $\phi = (\kappa, \{p_2, p_3\})$.

Another condition $roles(d, \phi) \supseteq roles(G, \phi) \setminus F$ states that only when the coordinator sees sufficient done notifications (in d) for ϕ , it issues done notifications to *all* non-failed participants in ϕ , i.e., $\langle \psi, roles(G, \phi) \setminus F \rangle^\phi$. Recall that $roles(d, \phi)$ returns all roles which have sent a done notification for the handling of ϕ and $roles(G, \phi)$ returns all roles involving in the handling of ϕ . Intuitively one might expect the condition to be $roles(d, \phi) = roles(G, \phi)$; the following example shows why this would be wrong.

Example 5. Consider a process P acting on channel $s[p]$ and $\{q\} \notin dom(H)$:

$$P = s[p] : t(\dots t(\dots)h(\{q\} : \eta, H')^{\phi'} . \eta')h(H)^\phi$$

Assume P has already reduced to:

$$P = s[p] : t(\underline{0})h(H)^\phi$$

We show why $roles(d, \phi) \supseteq roles(G, \phi) \setminus F$ is necessary. We start with the simple cases and then move to the more involving ones.

$$\begin{aligned}
T &::= p!\{l_i(S_i).T_i\}_{i \in I} \mid p?\{l_i(S_i).T_i\}_{i \in I} \mid t \mid \mu t.T \mid \text{end} \mid \underline{\text{end}} \mid t(T)\mathbf{h}(\mathcal{H})^\phi.T \\
\mathcal{H} &::= F:T \mid \mathcal{H}, \mathcal{H}
\end{aligned}$$

Fig. 9. The grammar of local types.

- (a) Assume q did not fail, the coordinator is $G : (\emptyset, d)$, and all roles in ϕ issued a done notification. Then $\text{roles}(d, \phi) = \text{roles}(G, \phi)$ and $F = \emptyset$.
- (b) Assume q failed in the try-handle ϕ' , the coordinator is $G : (\{q\}, d)$, and all roles except q in ϕ issued a done notification. $\text{roles}(d, \phi) \neq \text{roles}(G, \phi)$ however $\text{roles}(d, \phi) = \text{roles}(G, \phi) \setminus \{q\}$. Cases like this are the reason why **(IssueDone)** only requires done notifications from non-failed roles.
- (c) Assume q failed after it has issued a done notification for ϕ (i.e., q finished try-handle ϕ') and the coordinator collected it (by **(CollectDone)**), so we have $G : (\{q\}, d)$ and $q \in \text{roles}(d, \phi)$. Then $\text{roles}(d, \phi) \supset \text{roles}(G, \phi) \setminus \{q\}$. i.e. **(IssueDone)** needs to consider done notifications from failed roles.

Thus rule **(IssueDone)** has the condition $\text{roles}(d, \phi) \supseteq \text{roles}(G, \phi) \setminus F$ because of cases like (b) and (c).

The interplay between issuing of done notification (**(IssueDone)**) and issuing of failure notifications (**(F)**) is non-trivial. The following proposition clarifies that the participants in the same try-handle ϕ will never get confused with handling failures or completing the try-handle ϕ .

Proposition 1. *Given $s : h$ with $h = h' \cdot \langle \psi, p \rangle^\phi \cdot h''$ and $F\text{set}(h, p) \neq \emptyset$, the rule **(TryHdl)** is not applicable for the try-handle ϕ at the process playing role p .*

5 Local Types

Figure 9 defines local types for typing behaviors of endpoint processes with failure handling. Type $p!$ is the primitive for a sending type, and $p?$ is the primitive for a receiving type, derived from global type $p \rightarrow q\{l_i(S_i).G_i\}_{i \in I}$ by projection. Others correspond straightforwardly to process terms. Note that type $\underline{\text{end}}$ only appears in *runtime* type checking. Below we define $G \upharpoonright p$ to project a global type G on p , thus generating p 's local type.

Definition 4 (Projection). Consider a well-formed top-level global type $[\tilde{q}]G$. Then $G \upharpoonright p$ is defined as follows:

$$\begin{aligned}
(1) \quad G \upharpoonright p \text{ where } G &= t(G_0)\mathbf{h}(F_1 : G_1, \dots, F_n : G_n)^\kappa.G' = \\
&\begin{cases} t(G_0 \upharpoonright p)\mathbf{h}(F_1 : G_1 \upharpoonright p, \dots, F_n : G_n \upharpoonright p)^{(\kappa, \emptyset)}.G' \upharpoonright p & \text{if } p \in \text{roles}(G) \\ G' \upharpoonright p & \text{otherwise} \end{cases} \\
(2) \quad p_1 \rightarrow p_2\{l_i(S_i).G_i\}_{i \in I} \upharpoonright p &= \begin{cases} p_2!\{l_i(S_i).G_i \upharpoonright p\}_{i \in I} & \text{if } p = p_1 \\ p_1?\{l_i(S_i).G_i \upharpoonright p\}_{i \in I} & \text{if } p = p_2 \\ G_1 \upharpoonright p & \text{if } \forall i, j \in I. G_i \upharpoonright p = G_j \upharpoonright p \end{cases}
\end{aligned}$$

- (3) $(\mu t.G)\downarrow p = \mu t.(G\downarrow p)$ if $\exists t(G')h(H) \in G$ and $G\downarrow p \neq t'$ for any t'
 (4) $t\downarrow p = t$ (5) $\text{end}\downarrow p = \text{end}$

Otherwise it is undefined.

The main rule is (1): if p appears somewhere in the target try-handle global type then the endpoint type has a try-handle annotated with κ and the default logic (i.e., $F = \emptyset$). Note that even if $G_0\downarrow p = \text{end}$ the endpoint still gets such a try-handle because it needs to be ready for (possible) failure handling; if p does not appear anywhere in the target try-handle global type, then the projection skips to the continuation.

Rule (2) produces local types for interaction endpoints. If the endpoint is a sender (i.e., $p = p_1$), then its local type abstracts that it will send something from one of the possible internal choices defined in $\{l_i(S_i)\}_{i \in I}$ to p_2 , then continue as $G_k\downarrow p$, gained from the projection, if $k \in I$ is chosen. If the endpoint is a receiver (i.e., $p = p_2$), then its local type abstracts that it will receive something from one of the possible external choices defined in $\{l_i(S_i)\}_{i \in I}$ sent by p_1 ; the rest is similarly as for the sender. However, if p is not in this interaction, then its local type starts from the next interaction which p is in; moreover, because p does not know what choice that p_1 has made, every path $G_i\downarrow p$ lead by branch l_i shall be the same for p to ensure that interactions are consistent. For example, in $G = p_1 \rightarrow p_2\{l_1(S_1).p_3 \rightarrow p_1\ l_3(S),\ l_2(S_2).p_3 \rightarrow p_1\ l_4(S)\}$, interaction $p_3 \rightarrow p_1$ continues after $p_1 \rightarrow p_2$ takes place. If $l_3 \neq l_4$, then G is not projectable for p_3 because p_3 does not know which branch that p_1 has chosen; if p_1 chose branch l_1 , but p_3 (blindly) sends out label l_4 to p_1 , for p_1 it is a mistake (but it is not a mistake for p_3) because p_1 is expecting to receive label l_3 . To prevent such inconsistencies, we adopt the projection algorithm proposed in [24]. Other session type works [17, 39] provide ways to weaken the classical restriction on projection of branching which we use.

Rule (3) forbids a try-handle to appear in a recursive body, e.g., $\mu t.t(G)h(F : t)^\kappa.G$ is not allowed, but $t(\mu t.G)h(H)^\kappa$ and $t(G)h(F : \mu t.G', H)^\kappa$ are allowed. This is because κ is used to avoid confusion of messages from different try-handles. If a recursive body contains a try-handle, we have to dynamically generate different levels to maintain interaction consistency, so static type checking does not suffice. We are investigating alternative runtime checking mechanisms, but this is beyond the scope of this paper. Other rules are straightforward.

Example 6. Recall the global type G from Fig. 2 in Sect. 1. Applying projection rules defined in Definition 4 to G on every role in G we obtain the following:

$$\begin{aligned}
 T_{dfs} &= G\downarrow dfs = t(\mu t.w_1!l_{d_1}(S).w_2!l_{d_2}(S).w_1?l_{r_1}(S').w_2?l_{r_2}(S').t)h(\mathcal{H}_{dfs})^{(1,\emptyset)} \\
 \mathcal{H}_{dfs} &= \{w_1\} : \mu t'.w_2!l'_{d_1}(S).w_2?l'_{r_1}(S').t', \\
 &\quad \{w_2\} : \mu t''.w_1!l'_{d_2}(S).w_1?l'_{r_2}(S').t'', \{w_1, w_2\} : \text{end} \\
 T_{w_1} &= G\downarrow w_1 = t(\mu t.dfs?l_{d_1}(S).dfs!l_{r_1}(S').t)h(\mathcal{H}_{w_1})^{(1,\emptyset)} \\
 \mathcal{H}_{w_1} &= \{w_1\} : \text{end}, \{w_2\} : \mu t'.dfs?l'_{d_2}(S).dfs!l'_{r_2}(S').t', \{w_1, w_2\} : \text{end} \\
 T_{w_2} &= G\downarrow w_2 = t(\mu t.dfs?l_{d_2}(S).dfs!l_{r_2}(S').t)h(\mathcal{H}_{w_2})^{(1,\emptyset)} \\
 \mathcal{H}_{w_2} &= \{w_1\} : \mu t''.dfs?l'_{d_1}(S).dfs!l'_{r_1}(S').t'', \{w_2\} : \text{end}, \{w_1, w_2\} : \text{end}
 \end{aligned}$$

$$\begin{array}{c}
\frac{\Gamma \vdash a : \langle G \rangle}{\Gamma \vdash P \triangleright \{c : G[p]\}} \quad \frac{k \in I \quad \Gamma \vdash e : S_k}{\Gamma \vdash c : \eta_k \triangleright \{c : T_k\}} \quad \frac{}{\Gamma \vdash a[p].P \triangleright \emptyset} \quad \frac{}{\Gamma \vdash c : p! l_k(e).\eta_k \triangleright \{c : p! \{l_i(S_i).T_i\}_{i \in I}\}} \quad [\text{T-ini/T-snd}] \\
\\
\frac{\forall i \in I. \Gamma, x_i : S_i \vdash c : \eta_i \triangleright \{c : T_i\}}{\Gamma \vdash c : p? \{l_i(x_i).\eta_i\}_{i \in I} \triangleright \{c : p? \{l_i(S_i).T_i\}_{i \in I}\}} \quad [\text{T-rcv}] \\
\\
\frac{\Delta \text{ end-only}}{\Gamma \vdash c : \underline{0} \triangleright \Delta} \quad \frac{\Gamma \vdash c : \eta \triangleright \{c : \text{end}\}}{\Gamma \vdash c : \underline{0}.\eta \triangleright \{c : \underline{\text{end}}.\text{end}\}} \quad [\text{T-0/T-yd}] \\
\\
\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash c : \text{if } e \eta_1 \text{ else } \eta_2 \triangleright \Delta} \quad \frac{\Gamma \vdash e : S}{\Gamma, X : S \quad T \vdash c : X\langle e \rangle \triangleright \{c : T\}} \quad [\text{T-if/T-var}] \\
\\
\frac{\Gamma, X : S \quad t, x : S \vdash c : \eta_1 \triangleright \{c : T'\} \quad \Gamma, X : S \quad \mu t.T' \vdash c : \eta_2 \triangleright \{c : T\}}{\Gamma \vdash c : \text{def } X(x) = \eta_1 \text{ in } \eta_2 \triangleright \{c : T\}} \quad [\text{T-def}] \\
\\
\frac{\Gamma \vdash c : \eta \triangleright \{c : T\} \quad \Gamma \vdash c : \eta' \triangleright \{c : T'\} \quad \text{dom}(\mathbf{H}) = \text{dom}(\mathcal{H}) \quad \forall F \in \text{dom}(\mathbf{H}). \Gamma \vdash c : \mathbf{H}(F) \triangleright \{c : \mathcal{H}(F)\}}{\Gamma \vdash c : \mathbf{t}(\eta)\mathbf{h}(\mathbf{H})^\phi.\eta' \triangleright \{c : \mathbf{t}(T)\mathbf{h}(\mathcal{H})^\phi.T'\}} \quad [\text{T-th}]
\end{array}$$

Fig. 10. Typing rules for processes

6 Type System

Next we introduce our type system for typing processes. Figures 10 and 11 present typing rules for endpoints processes, and typing judgments for applications and systems respectively.

We define shared environments Γ to keep information on variables and the coordinator, and session environments Δ to keep information on endpoint types:

$$\begin{array}{ll}
\Gamma ::= \emptyset \mid \Gamma, X : S \mid \Gamma, x : S \mid \Gamma, a : G \mid \Gamma, \Psi & \Delta ::= \emptyset \mid \Delta, c : T \mid \Delta, s : \mathbf{h} \\
\mathbf{m} ::= \langle p, q, l(S) \rangle \mid \langle p, \text{crash } F \rangle \mid \langle p, q \rangle^\phi & \mathbf{h} ::= \emptyset \mid \mathbf{h} \cdot \mathbf{m}
\end{array}$$

Γ maps process variables X and content variables x to their types, shared names a to global types G , and a coordinator $\Psi = G : (F, d)$ to failures and done notifications it has observed. Δ maps session channels c to local types and session queues to queue types. We write $\Gamma, \Gamma' = \Gamma \cup \Gamma'$ when $\text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset$; same for Δ, Δ' . Queue types \mathbf{h} are composed of message types \mathbf{m} . Their permutation is defined analogously to the permutation for messages. The typing judgment for local processes $\Gamma \vdash P \triangleright \Delta$ states that process P is well-typed by Δ under Γ .

Since we do not define sequential composition for processes, our type system implicitly forbids session interleaving by $[\text{T-ini}]$. This is different from other session type works [15, 24], where session interleaving is prohibited for the progress property; here the restriction is inherent to the type system.

Figure 10 lists our typing rules for endpoint processes. Rule $[\text{T-ini}]$ says that if a process's set of actions is well-typed by $G[p]$ on some c , this process can play role p in a , which claims to have interactions obeying behaviors defined in G . $\langle G \rangle$ means that G is closed, i.e., devoid of type variables. This rule forbids

$$\begin{array}{c}
\Gamma \vdash s : \emptyset \triangleright \{s : \emptyset\} \quad \frac{\Gamma \vdash s : h \triangleright \{s : h\} \quad \Gamma \vdash e : S}{\Gamma \vdash s : h \cdot \langle p, q, l(e) \rangle \triangleright \{s : h \cdot \langle p, q, l(S) \rangle\}} \quad [\text{T-}\emptyset/\text{T-m}] \\
\\
\frac{(p_1, p_2) \in \{(p, \psi), (\psi, p)\} \quad \Gamma \vdash s : h \triangleright \{s : h\}}{\Gamma \vdash s : h \cdot \langle p_1, p_2 \rangle^\phi \triangleright \{s : h \cdot \langle p_1, p_2 \rangle^\phi\}} \quad [\text{T-D}] \\
\\
\frac{p \in \{q, \psi\} \quad m = \langle p, \text{crash } F \rangle \quad \Gamma \vdash N_1 \triangleright \Delta_1 \quad \Gamma \vdash N_2 \triangleright \Delta_2 \quad \Gamma \vdash s : h \triangleright \{s : h\}}{\Gamma \vdash s : h \cdot \langle p, \text{crash } F \rangle \triangleright \{s : h \cdot m\}} \quad \frac{\text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \emptyset}{\Gamma \vdash N_1 \mid N_2 \triangleright \Delta_1, \Delta_2} \quad [\text{T-F/T-pa}] \\
\\
\frac{\Gamma \vdash \mathcal{S} \triangleright \Delta \quad \Gamma \vdash \Delta_s \text{ coherent}}{\Gamma \vdash (\nu s)\mathcal{S} \triangleright \Delta \setminus \Delta_s} \quad \frac{\Gamma' = \Gamma, \Psi \quad \Gamma \vdash N \triangleright \Delta}{\Gamma' \vdash \Psi \blacklozenge N \triangleright \Delta} \quad [\text{T-s/T-sys}]
\end{array}$$

Fig. 11. Typing rules for applications and systems.

$a[p].b[q].P$ because a process can only use one session channel. Rule $[\text{T-snd}]$ states that an action for sending is well-typed to a sending type if the label and the type of the content are expected; $[\text{T-rcv}]$ states that an action for branching (i.e., for receiving) is well-typed to a branching type if all labels and the types of contents are as expected. Their follow-up actions shall also be well-typed. Rule $[\text{T-0}]$ types an idle process. Predicate end-only Δ is defined as stating whether all endpoints in Δ have type end:

Definition 5 (End-only Δ). We say Δ is end-only if and only if $\forall s[p] \in \text{dom}(\Delta), \Delta(s[p]) = \text{end}$.

Rule $[\text{T-yd}]$ types yielding actions, which only appear at runtime. Rule $[\text{T-if}]$ is standard in the sense that the process is well-typed by Δ if e has boolean type and its sub-processes (i.e., η_1 and η_2) are well-typed by Δ . Rules $[\text{T-var}, \text{T-def}]$ are based on a recent summary of MPSTs [14]. Note that $[\text{T-def}]$ forbids the type $\mu t.t$. Rule $[\text{T-th}]$ states that a try-handle is well-typed if it is annotated with the expected level ϕ , its default statement is well-typed, \mathcal{H} and \mathbf{H} have the same handler signatures, and all handling actions are well-typed.

Figure 11 shows typing rules for applications and systems. Rule $[\text{T-}\emptyset]$ types an empty queue. Rules $[\text{T-m}, \text{T-D}, \text{T-F}]$ simply type messages based on their shapes. Rule $[\text{T-pa}]$ says two applications composed in parallel are well-typed if they do not share any session channel. Rule $[\text{T-s}]$ says a part of a system \mathcal{S} can start a private session, say s , if \mathcal{S} is well-typed according to a $\Gamma \vdash \Delta_s$ that is *coherent* (defined shortly). The system $(\nu s)\mathcal{S}$ with a part becoming private in s is well-typed to $\Delta \setminus \Delta_s$, that is, Δ after removing Δ_s .

Definition 6 (A Session Environment Having s Only: Δ_s)

$$\Delta_s = \{s[p] : T \mid s[p] \in \text{dom}(\Delta)\} \cup \{s : h \mid s \in \text{dom}(\Delta)\}$$

Rule $[\text{T-sys}]$ says that a system $\Psi \blacklozenge N$ is well-typed if application N is well-typed and there exists a coordinator Ψ for handling this application. We say $\Gamma \vdash \Delta$ is coherent under Γ if the local types of all endpoints are dual to each other after their local types are updated because of messages or notifications in $s : h$.

Coherence. We say that a session environment is *coherent* if, at any time, given a session with its latest messages and notifications, every endpoint participating in it is able to find someone to interact with (i.e., its dual party exists) right now or afterwards.

Example 7. Continuing with Example 6 – the session environment $\Gamma \vdash \Delta$ is coherent even if w_2 will not receive any message from dfs at this point. The only possible action to take in Δ is that dfs sends out a message to w_1 . When this action fires, Δ is reduced to Δ' under a coordinator. (The reduction relation $\Gamma \vdash \Delta \rightarrow_T \Gamma' \vdash \Delta'$, where $\Gamma = \Gamma_0, \Psi$ and $\Gamma' = \Gamma_0, \Psi'$, is defined based on the rules of operational semantics of applications in Sect. 4, Figs. 6 and 7). In Δ' , which abstracts the environment when dfs sends a message to w_1 , w_2 will be able to receive this message.

$$\begin{aligned} \Delta &= s[dfs] : T_{dfs}, s[w_1] : T_{w_1}, s[w_2] : T_{w_2}, s : \emptyset \\ \Delta' &= s[dfs] : t(w_2!l_{d_2}(S).w_1?l_{r_1}(S').w_2?l_{r_2}(S').T)h(\mathcal{H})^{(1,\emptyset)}, \\ &\quad s[w_1] : T_{w_1}, s[w_2] : T_{w_2}, s : \langle dfs, w_1, l_{d_1}(S) \rangle \\ &\quad \text{where } T = \mu t. w_1!l_{d_1}(S).w_2!l_{d_2}(S).w_1?l_{r_1}(S').w_2?l_{r_2}(S').t \end{aligned}$$

We write $s[p] : T \bowtie s[q] : T'$ to state that actions of the two types are *dual*:

Definition 7 (Duality). We define $s[p] : T \bowtie s[q] : T'$ as follows:

$$\begin{aligned} s[p] : \text{end} \bowtie s[q] : \text{end} \quad s[p] : \underline{\text{end}} \bowtie s[q] : \underline{\text{end}} \quad s[p] : \text{end} \bowtie s[q] : \underline{\text{end}} \\ s[p] : \underline{\text{end}} \bowtie s[q] : \text{end} \quad s[p] : t \bowtie s[q] : t \quad \frac{s[p] : T \bowtie s[q] : T'}{s[p] : \mu t. T \bowtie s[q] : \mu t. T'} \\ \frac{\forall i \in I. s[p] : T_i \bowtie s[q] : T'_i}{s[p] : q! \{l_i(S_i).T_i\}_{i \in I} \bowtie s[q] : p? \{l_i(S_i).T'_i\}_{i \in I}} \\ \frac{s[p] : T_1 \bowtie s[q] : T_2 \quad s[p] : T'_1 \bowtie s[q] : T'_2 \quad \text{dom}(\mathcal{H}_1) = \text{dom}(\mathcal{H}_2) \quad \forall F \in \text{dom}(\mathcal{H}_1). s[p] : \mathcal{H}_1(F) \bowtie s[q] : \mathcal{H}_2(F)}{s[p] : t(T_1)h(\mathcal{H}_1)^\phi.T'_1 \bowtie s[q] : t(T_2)h(\mathcal{H}_2)^\phi.T'_2} \end{aligned}$$

Operation $T \downarrow p$ is to filter T to get the partial type which only contains actions of p . For example, $p_1!l'(S').p_2!l(S) \downarrow p_2 = p_2!l(S)$ and $p_1!\{T_1, T_2\} \downarrow p_2 = p_2?l(S)$ where $T_1 = l_1(S_1).p_2?l(S)$ and $T_2 = l_2(S_2).p_2?l(S)$. Next we define $(h)_{p \rightarrow q}$ to filter h to generate (1) the normal message types sent from p heading to q , and (2) the notifications heading to q . For example $(\langle p, q, l(S) \rangle \cdot \langle q, \text{crash } F \rangle \cdot \langle \psi, q \rangle^\phi \cdot \langle p, \text{crash } F \rangle)_{p \rightarrow q} = p?l(S) \cdot \langle F \rangle \cdot \langle \psi \rangle^\phi$. The message types are abbreviated to contain only necessary information.

We define $T - \mathbf{ht}$ to mean the effect of \mathbf{ht} on T . Its concept is similar to the *session remainder* defined in [35], which returns new local types of participants after participants consume messages from the global queue. Since failure notifications will not be consumed in our system, and we only have to observe the change of a participant's type after receiving or being triggered by some message types in \mathbf{ht} , we say that $T - \mathbf{ht}$ represents the effect of \mathbf{ht} on T . The behaviors follows our operational semantics of applications and systems defined in Figs. 6, 7, and 8.

For example $\mathbf{t}(q?\{l_i(S_i).T_i\}_{i \in I})\mathbf{h}(\mathcal{H})^\phi.T' - q?l_k(S_k).\mathbf{ht} = \mathbf{t}(T_k)\mathbf{h}(\mathcal{H})^\phi.T' - \mathbf{ht}$ where $k \in I$.

Now we define what it means for Δ to be coherent under Γ :

Definition 8 (Coherence). $\Gamma \vdash \Delta$ coherent if the following conditions hold:

1. If $\mathbf{s} : \mathbf{h} \in \Delta$, then $\exists G : (F, d) \in \Gamma$ and $\{p \mid \mathbf{s}[p] \in \text{dom}(\Delta)\} \subseteq \text{roles}(G)$ and G is well-formed and $\forall p \in \text{roles}(G), G \upharpoonright p$ is defined.
2. $\forall \mathbf{s}[p] : T, \mathbf{s}[q] : T' \in \Delta$ we have $\mathbf{s}[p] : T \downarrow q - (\mathbf{h})_{q \rightarrow p} \bowtie \mathbf{s}[q] : T' \downarrow p - (\mathbf{h})_{p \rightarrow q}$.

In condition 1, we require a coordinator for every session so that when a failure occurs, the coordinator can announce failure notifications to ask participants to handle the failure. Condition 2 requires that, for any two endpoints, say $\mathbf{s}[p]$ and $\mathbf{s}[q]$, in Δ , equation $\mathbf{s}[p] : T \downarrow q - (\mathbf{h})_{q \rightarrow p} \bowtie \mathbf{s}[q] : T' \downarrow p - (\mathbf{h})_{p \rightarrow q}$, must hold. This condition asserts that interactions of non-failed endpoints are dual to each other after the effect of \mathbf{h} ; while failed endpoints are removed from Δ , thus the condition is satisfied immediately.

7 Properties

We show that our type system ensures properties of subject congruence, subject reduction, and progress. All auxiliary definitions and proofs are in the long version [43].

The property of subject congruence states that if \mathcal{S} (a system containing an application and a coordinator) is well-typed by some session environment, then a \mathcal{S}' that is structurally congruent to it is also well-typed by the same session environment:

Theorem 1 (Subject Congruence). $\Gamma \vdash \mathcal{S} \triangleright \Delta$ and $\mathcal{S} \equiv \mathcal{S}'$ imply $\Gamma \vdash \mathcal{S}' \triangleright \Delta$.

Subject reduction states that a well-typed \mathcal{S} (coherent session environment respectively) is always well-typed (coherent respectively) after reduction:

Theorem 2 (Subject Reduction)

- $\Gamma \vdash \mathcal{S} \triangleright \Delta$ with $\Gamma \vdash \Delta$ coherent and $\mathcal{S} \rightarrow^* \mathcal{S}'$ imply that $\exists \Delta', \Gamma'$ such that $\Gamma' \vdash \mathcal{S}' \triangleright \Delta'$ and $\Gamma \vdash \Delta \rightarrow_T^* \Gamma' \vdash \Delta'$ or $\Delta \equiv \Delta'$ and $\Gamma' \vdash \Delta'$ coherent.
- $\Gamma \vdash \mathcal{S} \triangleright \emptyset$ and $\mathcal{S} \rightarrow^* \mathcal{S}'$ imply that $\Gamma' \vdash \mathcal{S}' \triangleright \emptyset$ for some Γ' .

We allow sessions to run in parallel at the top level, e.g., $\mathcal{S} = (\nu s_1)(\Psi_1 \blacklozenge N_1) \mid \dots \mid (\nu s_n)(\Psi_n \blacklozenge N_n)$. Assume we have \mathcal{S} with $a[p].P \in \mathcal{S}$. If we cannot apply rule **(Link)**, \mathcal{S} cannot reduce. To prevent this kind of situation, we require \mathcal{S} to be *initializable* such that, $\forall a[p].P \in \mathcal{S}$, **(Link)** is applicable.

The following property states that \mathcal{S} never gets stuck (property of progress):

Theorem 3 (Progress). If $\Gamma \vdash \mathcal{S} \triangleright \emptyset$ and \mathcal{S} is initializable, then either $\mathcal{S} \rightarrow^* \mathcal{S}'$ and \mathcal{S}' is initializable or $\mathcal{S}' = \Psi \blacklozenge \mathbf{s} : h \mid \dots \mid \Psi' \blacklozenge \mathbf{s}' : h'$ and h, \dots, h' only contain failure notifications sent by coordinators and messages heading to failed participants.

After all processes in \mathcal{S} terminate, failure notifications sent by coordinators are left; thus the final system can be of the form $\Psi \blacklozenge \mathbf{s} : h \mid \dots \mid \Psi' \blacklozenge \mathbf{s}' : h'$, where h, \dots, h' have failure notifications sent by coordinators and thus reduction rules (**CollectDone**), (**IssueDone**), and (**F**) will not be applied.

Minimality. The following proposition points out that, when all roles defined in a global type, say G , are robust, then the application obeying to G will never have interaction with a coordinator (i.e., interactions of the application are equivalent to those without a coordinator). This is an important property, as it states that our model does not incur coordination overhead when all participants are robust, or in failure-agnostic contexts as considered in previous MPST works.

Proposition 2. Assume $\forall p \in \text{roles}(G) = \{p_1, \dots, p_n\}$, p is robust and $P_i = \mathbf{s}[p_i] : \eta_i$ for $i \in \{1..n\}$ and $\mathcal{S} = (\nu \mathbf{s})(\Psi \blacklozenge P_1 \mid \dots \mid P_n \mid \mathbf{s} : h)$ where $P_i, i \in \{1..n\}$ contains no try-handle. Then we have $\Gamma \vdash \mathcal{S} \triangleright \emptyset$ and whenever $\mathcal{S} \rightarrow^* \mathcal{S}'$ we have $\Psi \in \mathcal{S}', \Psi = G : (\emptyset, \emptyset)$.

Proof. Immediately by typing rules $[\mathbf{T}\text{-ini}, \mathbf{T}\text{-s}, \mathbf{T}\text{-sys}]$, Definition 4 (Projection), and the operational semantics defined in Figs. 6, 7, and 8.

8 Related Work

Several session type works study exception handling [7, 9, 16, 30]. However, to the best of our knowledge this is the first theoretical work to develop a formalism and typing discipline for the coordinator-based model of *crash failure* handling in practical asynchronous distributed systems.

Structured interactional exceptions [7] study exception handling for binary sessions. The work extends session types with a *try-catch* construct and a *throw* instruction, allowing participants to raise runtime exceptions. Global escape [6] extends previous works on exception handling in binary session types to MPSTs. It supports nesting and sequencing of try-catch blocks with restrictions. Reduction rules for exception handling are of the form $\Sigma \vdash P \rightarrow \Sigma' \vdash P'$, where Σ is the *exception environment*. This central environment at the core of the semantics is updated synchronously and atomically. Furthermore, the reduction of a try-catch block to its continuation is done in a synchronous reduction step involving all participants in a block. Lastly this work can only handle exceptions, i.e., explicitly raised application-level failures. These do not affect communication channels [6], unlike participant crashes.

Similarly, our previous work [13] only deals with exceptions. An interaction $p \rightarrow q : S \vee F$ defines that p can send a message of type S to q . If F is not empty then instead of sending a message p can throw F . If a failure is thrown only participants that have casual dependencies to that failure are involved in the failure handling. No concurrent failures are allowed therefore all interactions which can raise failures are executed in a lock step fashion. As a consequence, the model can not be used to deal with crash failures.

Adameit et al. [1] propose session types for link failures, which extend session types with an optional block which surrounds a process and contains default values. The default values are used if a link failure occurs. In contrast to our work, the communication model is overall synchronous whereas our model is asynchronous; the optional block returns default values in case of a failure but it is still the task of the developer to do something useful with it.

Demangeon et al. study interrupts in MPSTs [16]. This work introduces an interruptible block $\{|G|\}^c \langle l \text{ by } \mathbf{r}; G' \rangle$ identified by c ; here the protocol G can be interrupted by a message l from \mathbf{r} and is continued by G' after either a normal or an interrupted completion of G . Interrupts are more a control flow instruction like exceptions than an actual failure handling construct, and the semantics can not model participant crashes.

Neykova and Yoshida [36] show that MPSTs can be used to calculate safe global states for a safe recovery in Erlang's *let it crash* model [2]. That work is well suited for recovery of lightweight processes in an actor setting. However, while it allows for elaborate failure handling by connecting (endpoint) processes with runtime monitors, the model does not address the fault tolerance of runtime monitors themselves. As monitors can be interacting in complex manners replication does not seem straightforwardly applicable, at least not without potentially hampering performance (just as with *straightforward* replication of entire applications).

Failure handling is studied in several process calculi and communication-centered programming languages without typing discipline. The conversation calculus [42] models exception behavior in abstract service-based systems with message-passing based communication. The work does not use channel types but studies the behavioral theory of bisimilarity. Error recovery is also studied in a concurrent object setting [45]; interacting objects are grouped into coordinated atomic actions (CAs) which enable safe error recovery. CAs can however not be nested. PSYNC [18] is a domain specific language based on the *heard-of* model of distributed computing [12]. Programs written in PSYNC are structured into rounds which are executed in a lock step manner. PSYNC comes with a state-based verification engine which enables checking of safety and liveness properties; for that programmers have to define non-trivial inductive invariants and ranking functions. In contrast to the coordinator model, the *heard-of* model is not widely deployed in practice. Verdi [44] is a framework for implementing and verifying distributed systems in Coq. It provides the possibility to verify the system against different network models. Verdi enables the verification of properties in an idealized fault model and then transfers the guarantees to more realistic fault models by applying transformation functions. Verdi supports safety properties but no liveness properties.

9 Final Remarks

Implementation. Based on our presented calculus we developed a domain-specific language and corresponding runtime system in Scala, using ZooKeeper as the

coordinator. Specifically our implementation provides mechanisms for (1) interacting with ZooKeeper as coordinator, (2) done and failure notification delivery and routing, (3) practical failure detection and dealing with false suspicions and (4) automatically inferring try-handle levels.

Conclusions. This work introduces a formal model of verified crash failure handling featuring a lightweight coordinator as common in many real-life systems. The model carefully exposes potential problems that may arise in distributed applications due to partial failures, such as inconsistent endpoint behaviors and orphan messages. Our typing discipline addresses these challenges by building on the mechanisms of MPSTs, e.g., global type well-formedness for sound failure handling specifications, modeling asynchronous permutations between regular messages and failure notifications in sessions, and the type-directed mechanisms for determining correct and orphaned messages in the event of failure. We adapt coherence of session typing environments (i.e., endpoint consistency) to consider failed roles and orphan messages, and show that our type system statically ensures subject reduction and progress in the presence of failures.

Future Work. We plan to expand our implementation and develop further applications. We believe dynamic role participation and role parameterization would be valuable for failure handling. Also, we are investigating options to enable addressing the coordinator as part of the protocol so that pertinent runtime information can be persisted by the coordinator. We plan to add support to our language and calculus for solving various explicit agreement tasks (e.g., consensus, atomic commit) via the coordinator.

References

1. Adameit, M., Peters, K., Nestmann, U.: Session types for link failures. In: Bouajjani, A., Silva, A. (eds.) FORTE 2017. LNCS, vol. 10321, pp. 1–16. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60225-7_1
2. Armstrong, J.: Making reliable distributed systems in the presence of software errors. Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden (2003)
3. Birman, K.P.: Byzantine Clients (2017). <https://goo.gl/1Qbc4r>
4. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In: OSDI 2006, pp. 335–350. USENIX Association (2006)
5. Caires, L., Pérez, J.A.: Multiparty session types within a canonical binary theory, and beyond. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 74–95. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_6
6. Capecchi, S., Giachino, E., Yoshida, N.: Global escape in multiparty sessions. *MSCS* **26**(2), 156–205 (2016)
7. Carbone, M., Honda, K., Yoshida, N.: Structured interactional exceptions in session types. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_32

8. Carbone, M., Lindley, S., Montesi, F., Schürmann, C., Wadler, P.: Coherence generalises duality: a logical explanation of multiparty session types. In: CONCUR 2016. LIPIcs, vol. 59, pp. 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
9. Carbone, M., Yoshida, N., Honda, K.: Asynchronous session types: exceptions and multiparty interactions. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 187–212. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01918-0_5
10. Chandra, T.D., Hadzilacos, V., Toueg, S., Charron-Bost, B.: On the impossibility of group membership. In: PODC 1996, pp. 322–330. ACM (1996)
11. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: a distributed storage system for structured data. In: OSDI 2006, pp. 205–218. USENIX Association (2006)
12. Charron-Bost, B., Schiper, A.: The Heard-Of model: computing in distributed systems with benign faults. *Distrib. Comput.* **22**(1), 49–71 (2009)
13. Chen, T.-C., Viering, M., Bejleri, A., Ziarek, L., Eugster, P.: A type theory for robust failure handling in distributed systems. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 96–113. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_7
14. Coppo, M., Dezani-Ciancaglini, M., Padovani, L., Yoshida, N.: A gentle introduction to multiparty asynchronous session types. In: Bernardo, M., Johnsen, E.B. (eds.) SFM 2015. LNCS, vol. 9104, pp. 146–178. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18941-3_4
15. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. *MSCS* **26**(2), 238–302 (2016)
16. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations. *Formal Methods Syst. Des.* **46**(3), 197–225 (2015)
17. Deniérou, P.M., Yoshida, N.: Dynamic multirole session types. In: POPL 2011, pp. 435–446. ACM (2011)
18. Dragoi, C., Henzinger, T., Zufferey, D.: PSync: a partially synchronous language for fault-tolerant distributed algorithms. In: POPL 2016, pp. 400–415. ACM (2016)
19. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. *J. ACM* **32**(2), 374–382 (1985)
20. Ghemawat, S., Gobiouff, H., Leung, S.T.: The Google file system. In: SOSP 2003, pp. 29–43. ACM (2003)
21. Gilbert, S., Lynch, N.: Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* **33**(2), 51–59 (2002)
22. Guerraoui, R., Schiper, A.: The generic consensus service. *IEEE Trans. Softw. Eng.* **27**(1), 29–41 (2001)
23. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0053567>
24. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. *J. ACM* **63**(1), 9:1–9:67 (2016)
25. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: Stevens, P., Wasowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 401–418. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_24
26. Hunt, P.: ZooKeeper: wait-free coordination for internet-scale systems. In: USENIX 2010. USENIX Association (2010)

27. Hüttel, H., et al.: Foundations of session types and behavioural contracts. *ACM Comput. Surv.* **49**(1), 3:1–3:36 (2016)
28. Imai, K., Yoshida, N., Yuen, S.: **Session-ocaml**: a session-based library with polarities and lenses. In: Jacquet, J.-M., Massink, M. (eds.) *COORDINATION 2017*. LNCS, vol. 10319, pp. 99–118. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59746-1_6
29. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.M.: Mace: language support for building distributed systems. In: *PLDI 2007*, vol. 42, pp. 179–188. ACM (2007)
30. Kouzapas, D., Yoshida, N.: Globally governed session semantics. *LMCS* **10**(4), 1–45 (2014)
31. Kreps, J., Narkhede, N., Rao, J.: Kafka: a distributed messaging system for log processing. In: *NetDB 2011* (2011)
32. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.* **4**(3), 382–401 (1982)
33. Leners, J.B., Wu, H., Hung, W.L., Aguilera, M.K., Walfish, M.: Detecting failures in distributed systems with the FALCON spy network. In: *SOSP 2011*, pp. 279–294. ACM (2011)
34. Lindley, S., Morris, J.G.: embedding session types in haskell. In: *Haskell 2016*, pp. 133–145. ACM (2016)
35. Mostrous, D., Yoshida, N.: Session typing and asynchronous subtyping for the higher-order π -calculus. *Inf. Comput.* **241**, 227–263 (2015)
36. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: *CC 2017*, pp. 98–108. ACM (2017)
37. Padovani, L.: A simple library implementation of binary sessions. *J. Funct. Program.* **27**, e4 (2017)
38. Pucella, R., Tov, J.A.: Haskell session types with (almost) no class. In: *Haskell 2008*, pp. 25–36. ACM (2008)
39. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: *ECOOP 2017*. *LIPICs*, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
40. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: *MSST 2010*, pp. 1–10. IEEE Computer Society (2010)
41. Sivaramakrishnan, K.C., Qudeisat, M., Ziarek, L., Nagaraj, K., Eugster, P.: Efficient sessions. *Sci. Comput. Program.* **78**(2), 147–167 (2013)
42. Vieira, H.T., Caires, L., Seco, J.C.: The conversation calculus: a model of service-oriented computation. In: Drossopoulou, S. (ed.) *ESOP 2008*. LNCS, vol. 4960, pp. 269–283. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78739-6_21
43. Viering, M., Chen, T.C., Eugster, P., Hu, R., Ziarek, L.: Technical appendix: a typing discipline for statically verified crash failure handling in distributed systems. http://distributed-systems-programming-group.github.io/paper/2018/esop_long.pdf
44. 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)
45. Xu, J., Randell, B., Romanovsky, A.B., Rubira, C.M.F., Stroud, R.J., Wu, Z.: Fault tolerance in concurrent object-oriented software through coordinated error recovery. In: *FTCS 1995*, pp. 499–508. IEEE Computer Society (1995)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





On Polymorphic Sessions and Functions

A Tale of Two (Fully Abstract) Encodings

Bernardo Toninho^{1,2}  and Nobuko Yoshida² 

¹ NOVA-LINCS, Departamento de Informática, FCT, Universidade Nova de Lisboa,
Lisbon, Portugal

`btoninho@fct.unl.pt`

² Department of Computing, Imperial College London, London, UK

Abstract. This work exploits the logical foundation of session types to determine what kind of type discipline for the π -calculus can exactly capture, and is captured by, λ -calculus behaviours. Leveraging the proof theoretic content of the soundness and completeness of sequent calculus and natural deduction presentations of linear logic, we develop the first *mutually inverse* and *fully abstract* processes-as-functions and functions-as-processes encodings between a polymorphic session π -calculus and a linear formulation of System F. We are then able to derive results of the session calculus from the theory of the λ -calculus: (1) we obtain a characterisation of inductive and coinductive session types via their algebraic representations in System F; and (2) we extend our results to account for *value* and *process* passing, entailing strong normalisation.

1 Introduction

Dating back to Milner’s seminal work [29], encodings of λ -calculus into π -calculus are seen as essential benchmarks to examine expressiveness of various extensions of the π -calculus. Milner’s original motivation was to demonstrate the power of link mobility by decomposing higher-order computations into pure name passing. Another goal was to analyse functional behaviours in a broad computational universe of concurrency and non-determinism. While *operationally* correct encodings of many higher-order constructs exist, it is challenging to obtain encodings that are precise wrt behavioural equivalence: the semantic distance between the λ -calculus and the π -calculus typically requires either restricting process behaviours [45] (e.g. via typed equivalences [5]) or enriching the λ -calculus with constants that allow for a suitable characterisation of the term equivalence induced by the behavioural equivalence on processes [43].

Encodings in π -calculi also gave rise to new typing disciplines: Session types [20, 22], a typing system that is able to ensure deadlock-freedom for communication protocols between two or more parties [23], were originally motivated “from process encodings of various data structures in an asynchronous version of the π -calculus” [21]. Recently, a propositions-as-types correspondence between linear logic and session types [8, 9, 54] has produced several new developments

and logically-motivated techniques [7, 26, 49, 54] to augment both the theory and practice of session-based message-passing concurrency. Notably, parametric session polymorphism [7] (in the sense of Reynolds [41]) has been proposed and a corresponding abstraction theorem has been shown.

Our work expands upon the proof theoretic consequences of this propositions-as-types correspondence to address the problem of how to *exactly* match the behaviours induced by session π -calculus encodings of the λ -calculus with those of the λ -calculus. We develop *mutually inverse* and *fully abstract* encodings (up to typed observational congruences) between a polymorphic session-typed π -calculus and the polymorphic λ -calculus. The encodings arise from the proof theoretic content of the equivalence between sequent calculus (i.e. the session calculus) and natural deduction (i.e. the λ -calculus) for *second-order* intuitionistic linear logic, greatly generalising [49]. While fully abstract encodings between λ -calculi and π -calculi have been proposed (e.g. [5, 43]), our work is the first to consider a two-way, *both* mutually inverse *and* fully abstract embedding between the two calculi by crucially exploiting the linear logic-based session discipline. This also sheds some definitive light on the nature of concurrency in the (logical) session calculi, which exhibit “don’t care” forms of non-determinism (e.g. processes may race on stateless replicated servers) rather than “don’t know” non-determinism (which requires less harmonious logical features [2]).

In the spirit of Gentzen [14], we use our encodings as a tool to study non-trivial properties of the session calculus, deriving them from results in the λ -calculus: We show the existence of inductive and coinductive sessions in the polymorphic session calculus by considering the representation of initial F -algebras and final F -coalgebras [28] in the polymorphic λ -calculus [1, 19] (in a linear setting [6]). By appealing to full abstraction, we are able to derive processes that satisfy the necessary algebraic properties and thus form adequate *uniform* representations of inductive and coinductive session types. The derived algebraic properties enable us to reason about standard data structure examples, providing a logical justification to typed variations of the representations in [30].

We systematically extend our results to a session calculus with λ -term and process passing (the latter being the core calculus of [50], inspired by Benton’s LNL [4]). By showing that our encodings naturally adapt to this setting, we prove that it is possible to encode higher-order process passing in the first-order session calculus fully abstractly, providing a typed and proof-theoretically justified re-envisioning of Sangiorgi’s encodings of higher-order π -calculus [46]. In addition, the encoding instantly provides a strong normalisation property of the higher-order session calculus.

Contributions and the outline of our paper are as follows:

§ 3.1 develops a functions-as-processes encoding of a linear formulation of System F, Linear-F, using a logically motivated polymorphic session π -calculus, $\text{Poly}\pi$, and shows that the encoding is operationally sound and complete.

§ 3.2 develops a processes-as-functions encoding of $\text{Poly}\pi$ into Linear-F, arising from the completeness of the sequent calculus wrt natural deduction, also operationally sound and complete.

§ 3.3 studies the relationship between the two encodings, establishing they are *mutually inverse* and *fully abstract* wrt typed congruence, the first two-way embedding satisfying *both* properties.

§ 4 develops a *faithful* representation of inductive and coinductive session types in $\text{Poly}\pi$ via the encoding of initial and final (co)algebras in the polymorphic λ -calculus. We demonstrate a use of these algebraic properties via examples.

§ 4.2 and 4.3 study term-passing and process-passing session calculi, extending our encodings to provide embeddings into the first-order session calculus. We show full abstraction and mutual inversion results, and derive strong normalisation of the higher-order session calculus from the encoding.

In order to introduce our encodings, we first overview $\text{Poly}\pi$, its typing system and behavioural equivalence (§ 2). We discuss related work and conclude with future work (§ 5). Detailed proofs can be found in [52].

2 Polymorphic Session π -Calculus

This section summarises the polymorphic session π -calculus [7], dubbed $\text{Poly}\pi$, arising as a process assignment to second-order linear logic [15], its typing system and behavioural equivalences.

2.1 Processes and Typing

Syntax. Given an infinite set Λ of names x, y, z, u, v , the grammar of processes P, Q, R and session types A, B, C is defined by:

$$\begin{aligned} P, Q, R &::= x\langle y \rangle.P \mid x(y).P \mid P \mid Q \mid (\nu y)P \mid [x \leftrightarrow y] \mid \mathbf{0} \\ &\quad \mid x\langle A \rangle.P \mid x(Y).P \mid x.\text{inl}; P \mid x.\text{inr}; P \mid x.\text{case}(P, Q) \mid !x(y).P \\ A, B &::= \mathbf{1} \mid A \multimap B \mid A \otimes B \mid A \& B \mid A \oplus B \mid !A \mid \forall X.A \mid \exists X.A \mid X \end{aligned}$$

$x\langle y \rangle.P$ denotes the output of channel y on x with continuation process P ; $x(y).P$ denotes an input along x , bound to y in P ; $P \mid Q$ denotes parallel composition; $(\nu y)P$ denotes the restriction of name y to the scope of P ; $\mathbf{0}$ denotes the inactive process; $[x \leftrightarrow y]$ denotes the linking of the two channels x and y (implemented as renaming); $x\langle A \rangle.P$ and $x(Y).P$ denote the sending and receiving of a *type* A along x bound to Y in P of the receiver process; $x.\text{inl}; P$ and $x.\text{inr}; P$ denote the emission of a selection between the left or right branch of a receiver $x.\text{case}(P, Q)$ process; $!x(y).P$ denotes an input-guarded replication, that spawns replicas upon receiving an input along x . We often abbreviate $(\nu y)x\langle y \rangle.P$ to $\bar{x}\langle y \rangle.P$ and omit trailing $\mathbf{0}$ processes. By convention, we range over linear channels with x, y, z and shared channels with u, v, w .

$$\begin{array}{c}
\begin{array}{c}
(\text{out}) \quad x\langle y \rangle.P \xrightarrow{\overline{x\langle y \rangle}} P \quad (\text{in}) \quad x(y).P \xrightarrow{x(z)} P\{z/y\} \quad (\text{outT}) \quad x\langle A \rangle.P \xrightarrow{\overline{x\langle A \rangle}} P \quad (\text{inT}) \quad x(Y).P \xrightarrow{x(B)} P\{B/Y\} \\
(\text{lout}) \quad x.\text{inl}; P \xrightarrow{\overline{x.\text{inl}}} P \quad (\text{id}) \quad (\nu x)([x \leftrightarrow y] \mid P) \xrightarrow{\tau} P\{y/x\} \quad (\text{open}) \quad \frac{P \xrightarrow{\overline{x\langle y \rangle}} Q}{(\nu y)P \xrightarrow{(\nu y)x\langle y \rangle} Q} \\
(\text{lin}) \quad x.\text{case}(P, Q) \xrightarrow{x.\text{inl}} P \quad (\text{rep}) \quad !x(y).P \xrightarrow{x(z)} P\{z/y\} \mid !x(y).P \quad (\text{close}) \quad \frac{P \xrightarrow{(\nu y)x\langle y \rangle} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q')} \\
(\text{par}) \quad \frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R} \quad (\text{com}) \quad \frac{P \xrightarrow{\bar{\alpha}} P' \quad Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \quad (\text{res}) \quad \frac{P \xrightarrow{\alpha} Q}{(\nu y)P \xrightarrow{\alpha} (\nu y)Q}
\end{array}
\end{array}$$

Fig. 1. Labelled transition system.

The syntax of session types is that of (intuitionistic) linear logic propositions which are assigned to channels according to their usages in processes: $\mathbf{1}$ denotes the type of a channel along which no further behaviour occurs; $A \multimap B$ denotes a session that waits to receive a channel of type A and will then proceed as a session of type B ; dually, $A \otimes B$ denotes a session that sends a channel of type A and continues as B ; $A \& B$ denotes a session that offers a choice between proceeding as behaviours A or B ; $A \oplus B$ denotes a session that internally chooses to continue as either A or B , signalling appropriately to the communicating partner; $!A$ denotes a session offering an unbounded (but finite) number of behaviours of type A ; $\forall X.A$ denotes a polymorphic session that receives a type B and behaves uniformly as $A\{B/X\}$; dually, $\exists X.A$ denotes an existentially typed session, which emits a type B and behaves as $A\{B/X\}$.

Operational Semantics. The operational semantics of our calculus is presented as a standard labelled transition system (Fig. 1) in the style of the *early* system for the π -calculus [46].

In the remainder of this work we write \equiv for a standard π -calculus structural congruence extended with the clause $[x \leftrightarrow y] \equiv [y \leftrightarrow x]$. In order to streamline the presentation of observational equivalence [7, 36], we write $\equiv_!$ for structural congruence extended with the so-called sharpened replication axioms [46], which capture basic equivalences of replicated processes (and are present in the proof dynamics of the exponential of linear logic). A transition $P \xrightarrow{\alpha} Q$ denotes that P may evolve to Q by performing the action represented by label α . An action α ($\bar{\alpha}$) requires a matching $\bar{\alpha}$ (α) in the environment to enable progress. Labels include: the silent internal action τ , output and bound output actions $\overline{x\langle y \rangle}$ and $\overline{(\nu z)x\langle z \rangle}$; input action $x(y)$; the binary choice actions $(x.\text{inl}, \overline{x.\text{inl}}, x.\text{inr}, \text{ and } \overline{x.\text{inr}})$; and output and input actions of types $\overline{x\langle A \rangle}$ and $x(A)$.

The labelled transition relation is defined by the rules in Fig. 1, subject to the side conditions: in rule (res), we require $y \notin \text{fn}(\alpha)$; in rule (par), we require $\text{bn}(\alpha) \cap \text{fn}(R) = \emptyset$; in rule (close), we require $y \notin \text{fn}(Q)$. We omit the symmetric versions of (par), (com), (lout), (lin), (close) and closure under α -conversion. We write $\rho_1 \rho_2$ for the composition of relations ρ_1, ρ_2 . We write \rightarrow to stand for $\xrightarrow{\tau} \equiv$.

$$\begin{array}{c}
(\neg\text{R}) \frac{\Omega; \Gamma; \Delta, x:A \vdash P :: z:B}{\Omega; \Gamma; \Delta \vdash z(x).P :: z:A \neg\text{O} B} \quad (\otimes\text{R}) \frac{\Omega; \Gamma; \Delta_1 \vdash P :: y:A \quad \Omega; \Gamma; \Delta_2 \vdash Q :: z:B}{\Omega; \Gamma; \Delta_1, \Delta_2 \vdash (\nu x)z\langle y \rangle.(P \mid Q) :: z:A \otimes B} \\
(\forall\text{R}) \frac{\Omega, X; \Gamma; \Delta \vdash P :: z:A}{\Omega; \Gamma; \Delta \vdash z(X).P :: z:\forall X.A} \quad (\forall\text{L}) \frac{\Omega \vdash B \text{ type} \quad \Omega; \Gamma; \Delta, x:A\{B/X\} \vdash P :: z:C}{\Omega; \Gamma; \Delta, x:\forall X.A \vdash x\langle B \rangle.P :: z:C} \\
(\exists\text{R}) \frac{\Omega \vdash B \text{ type} \quad \Omega; \Gamma; \Delta \vdash P :: z:A\{B/X\}}{\Omega; \Gamma; \Delta \vdash z\langle B \rangle.P :: z:\exists X.A} \quad (\exists\text{L}) \frac{\Omega, X; \Gamma; \Delta, x:A \vdash P :: z:C}{\Omega; \Gamma; \Delta, x:\exists X.A \vdash x(X).P :: z:C} \\
(\text{id}) \frac{}{\Omega; \Gamma; x:A \vdash [x \leftrightarrow z] :: z:A} \quad (\text{cut}) \frac{\Omega; \Gamma; \Delta_1 \vdash P :: x:A \quad \Omega; \Gamma; \Delta_2, x:A \vdash Q :: z:C}{\Omega; \Gamma; \Delta_1, \Delta_2 \vdash (\nu x)(P \mid Q) :: z:C}
\end{array}$$

Fig. 2. Typing rules (abridged – see [52] for all rules).

Weak transitions are defined as usual: we write \Rightarrow for the reflexive, transitive closure of $\xrightarrow{\tau}$ and \rightarrow^+ for the transitive closure of $\xrightarrow{\tau}$. Given $\alpha \neq \tau$, notation $\xRightarrow{\alpha}$ stands for $\Rightarrow \xrightarrow{\alpha} \Rightarrow$ and $\xrightarrow{\tau}$ stands for \Rightarrow .

Typing System. The typing rules of Poly π are given in Fig. 2, following [7]. The rules define the judgment $\Omega; \Gamma; \Delta \vdash P :: z:A$, denoting that process P offers a session of type A along channel z , using the *linear* sessions in Δ , (potentially) using the *unrestricted* or *shared* sessions in Γ , with polymorphic type variables maintained in Ω . We use a well-formedness judgment $\Omega \vdash A \text{ type}$ which states that A is well-formed wrt the type variable environment Ω (i.e. $fv(A) \subseteq \Omega$). We often write T for the right-hand side typing $z:A$, \cdot for the empty context and Δ, Δ' for the union of contexts Δ and Δ' , only defined when Δ and Δ' are disjoint. We write $\cdot \vdash P :: T$ for $\cdot; \cdot; \cdot \vdash P :: T$.

As in [8, 9, 36, 54], the typing discipline enforces that channel outputs always have as object a *fresh* name, in the style of the internal mobility π -calculus [44]. We clarify a few of the key rules: Rule $\forall\text{R}$ defines the meaning of (impredicative) universal quantification over session types, stating that a session of type $\forall X.A$ inputs a type and then behaves uniformly as A ; dually, to use such a session (rule $\forall\text{L}$), a process must output a type B which then warrants the use of the session as type $A\{B/X\}$. Rule $\neg\text{O}$ captures session input, where a session of type $A \neg\text{O} B$ expects to receive a session of type A which will then be used to produce a session of type B . Dually, session output (rule $\otimes\text{R}$) is achieved by producing a fresh session of type A (that uses a disjoint set of sessions to those of the continuation) and outputting the fresh session along z , which is then a session of type B . Linear composition is captured by rule cut which enables a process that offers a session $x:A$ (using linear sessions in Δ_1) to be composed with a process that *uses* that session (amongst others in Δ_2) to offer $z:C$. As shown in [7], typing entails Subject Reduction, Global Progress, and Termination.

Observational Equivalences. We briefly summarise the typed congruence and logical equivalence with polymorphism, giving rise to a suitable notion of relational parametricity in the sense of Reynolds [41], defined as a contextual logical

relation on typed processes [7]. The logical relation is reminiscent of a typed bisimulation. However, extra care is needed to ensure well-foundedness due to impredicative type instantiation. As a consequence, the logical relation allows us to reason about process equivalences where type variables are not instantiated with *the same*, but rather *related* types.

Typed Barbed Congruence (\cong). We use the typed contextual congruence from [7], which preserves *observable* actions, called barbs. Formally, *barbed congruence*, noted \cong , is the largest equivalence on well-typed processes that is τ -closed, barb preserving, and contextually closed under typed contexts; see [7, 52] for the full definition.

Logical Equivalence (\approx_L). The definition of logical equivalence is no more than a typed contextual bisimulation with the following intuitive reading: given two open processes P and Q (i.e. processes with non-empty left-hand side typings), we define their equivalence by inductively closing out the context, composing with equivalent processes offering appropriately typed sessions. When processes are closed, we have a single distinguished session channel along which we can perform observations, and proceed inductively on the structure of the offered session type. We can then show that such an equivalence satisfies the necessary fundamental properties (Theorem 2.3).

The logical relation is defined using the candidates technique of Girard [16]. In this setting, an *equivalence candidate* is a relation on typed processes satisfying basic closure conditions: an equivalence candidate must be compatible with barbed congruence and closed under forward and converse reduction.

Definition 2.1 (Equivalence Candidate). An *equivalence candidate* \mathcal{R} at $z:A$ and $z:B$, noted $\mathcal{R} :: z:A \Leftrightarrow B$, is a binary relation on processes such that, for every $(P, Q) \in \mathcal{R} :: z:A \Leftrightarrow B$ both $\cdot \vdash P :: z:A$ and $\cdot \vdash Q :: z:B$ hold, together with the following (we often write $(P, Q) \in \mathcal{R} :: z:A \Leftrightarrow B$ as $P \mathcal{R} Q :: z:A \Leftrightarrow B$):

1. If $(P, Q) \in \mathcal{R} :: z:A \Leftrightarrow B$, $\cdot \vdash P \cong P' :: z:A$, and $\cdot \vdash Q \cong Q' :: z:B$ then $(P', Q') \in \mathcal{R} :: z:A \Leftrightarrow B$.
2. If $(P, Q) \in \mathcal{R} :: z:A \Leftrightarrow B$ then, for all P_0 such that $\cdot \vdash P_0 :: z:A$ and $P_0 \Longrightarrow P$, we have $(P_0, Q) \in \mathcal{R} :: z:A \Leftrightarrow B$. Symmetrically for Q .

To define the logical relation we rely on some auxiliary notation, pertaining to the treatment of type variables arising due to impredicative polymorphism. We write $\omega : \Omega$ to denote a mapping ω that assigns a closed type to the type variables in Ω . We write $\omega(X)$ for the type mapped by ω to variable X . Given two mappings $\omega : \Omega$ and $\omega' : \Omega$, we define an equivalence candidate assignment η between ω and ω' as a mapping of equivalence candidate $\eta(X) :: -\omega(X) \Leftrightarrow \omega'(X)$ to the type variables in Ω , where the particular choice of a distinguished right-hand side channel is *delayed* (i.e. to be instantiated later on). We write $\eta(X)(z)$ for the instantiation of the (delayed) candidate with the name z . We write $\eta : \omega \Leftrightarrow \omega'$ to denote that η is a candidate assignment between ω and ω' ; and $\hat{\omega}(P)$ to denote the application of mapping ω to P .

We define a sequent-indexed family of process relations, that is, a set of pairs of processes (P, Q) , written $\Gamma; \Delta \vdash P \approx_L Q :: T[\eta : \omega \Leftrightarrow \omega']$, satisfying some conditions, typed under $\Omega; \Gamma; \Delta \vdash T$, with $\omega : \Omega$, $\omega' : \Omega$ and $\eta : \omega \Leftrightarrow \omega'$. Logical equivalence is defined inductively on the size of the typing contexts and then on the structure of the right-hand side type. We show only select cases (see [52] for the full definition).

Definition 2.2 (Logical Equivalence). (Base Case) Given a type A and mappings ω, ω', η , we define *logical equivalence*, noted $P \approx_L Q :: z:A[\eta : \omega \Leftrightarrow \omega']$, as the smallest symmetric binary relation containing all pairs of processes (P, Q) such that (i) $\cdot \vdash \hat{\omega}(P) :: z:\hat{\omega}(A)$; (ii) $\cdot \vdash \hat{\omega}'(Q) :: z:\hat{\omega}'(A)$; and (iii) satisfies the conditions given below:

- $P \approx_L Q :: z:X[\eta : \omega \Leftrightarrow \omega']$ iff $(P, Q) \in \eta(X)(z)$
- $P \approx_L Q :: z:A \multimap B[\eta : \omega \Leftrightarrow \omega']$ iff $\forall P', y. (P \xrightarrow{z(y)} P') \Rightarrow \exists Q'. Q \xRightarrow{z(y)} Q' \text{ s.t. } \forall R_1, R_2. R_1 \approx_L R_2 :: y:A[\eta : \omega \Leftrightarrow \omega'](\nu y)(P' | R_1) \approx_L (\nu y)(Q' | R_2) :: z:B[\eta : \omega \Leftrightarrow \omega']$
- $P \approx_L Q :: z:A \otimes B[\eta : \omega \Leftrightarrow \omega']$ iff $\forall P', y. (P \xrightarrow{(\nu y)z(y)} P') \Rightarrow \exists Q'. Q \xRightarrow{(\nu y)z(y)} Q' \text{ s.t. } \exists P_1, P_2, Q_1, Q_2. P' \equiv P_1 | P_2 \wedge Q' \equiv Q_1 | Q_2 \wedge P_1 \approx_L Q_1 :: y:A[\eta : \omega \Leftrightarrow \omega'] \wedge P_2 \approx_L Q_2 :: z:B[\eta : \omega \Leftrightarrow \omega']$
- $P \approx_L Q :: z:\forall X.A[\eta : \omega \Leftrightarrow \omega']$ iff $\forall B_1, B_2, P', \mathcal{R} :: -:B_1 \Leftrightarrow B_2. (P \xrightarrow{z(B_1)} P') \text{ implies } \exists Q'. Q \xRightarrow{z(B_2)} Q', P' \approx_L Q' :: z:A[\eta[X \mapsto \mathcal{R}] : \omega[X \mapsto B_1] \Leftrightarrow \omega'[X \mapsto B_2]]$

(Inductive Case). Let Γ, Δ be non empty. Given $\Omega; \Gamma; \Delta \vdash P :: T$ and $\Omega; \Gamma; \Delta \vdash Q :: T$, the binary relation on processes $\Gamma; \Delta \vdash P \approx_L Q :: T[\eta : \omega \Leftrightarrow \omega']$ (with $\omega, \omega' : \Omega$ and $\eta : \omega \Leftrightarrow \omega'$) is inductively defined as:

$$\begin{aligned}
 \Gamma; \Delta, y : A \vdash P \approx_L Q :: T[\eta : \omega \Leftrightarrow \omega'] &\text{ iff } \forall R_1, R_2. \text{ s.t. } R_1 \approx_L R_2 :: y:A[\eta : \omega \Leftrightarrow \omega'], \\
 &\quad \Gamma; \Delta \vdash (\nu y)(\hat{\omega}(P) | \hat{\omega}(R_1)) \approx_L (\nu y)(\hat{\omega}'(Q) | \hat{\omega}'(R_2)) :: T[\eta : \omega \Leftrightarrow \omega'] \\
 \Gamma, u:A; \Delta \vdash P \approx_L Q :: T[\eta : \omega \Leftrightarrow \omega'] &\text{ iff } \forall R_1, R_2. \text{ s.t. } R_1 \approx_L R_2 :: y:A[\eta : \omega \Leftrightarrow \omega'], \\
 \Gamma; \Delta \vdash (\nu u)(\hat{\omega}(P) !u(y).\hat{\omega}(R_1)) &\approx_L (\nu u)(\hat{\omega}'(Q) !u(y).\hat{\omega}'(R_2)) :: T[\eta : \omega \Leftrightarrow \omega']
 \end{aligned}$$

For the sake of readability we often omit the $\eta : \omega \Leftrightarrow \omega'$ portion of \approx_L , which is henceforth implicitly universally quantified. Thus, we write $\Omega; \Gamma; \Delta \vdash P \approx_L Q :: z:A$ (or $P \approx_L Q$) iff the two given processes are logically equivalent for all consistent instantiations of its type variables.

It is instructive to inspect the clause for type input $(\forall X.A)$: the two processes must be able to match inputs of any pair of *related* types (i.e. types related by a candidate), such that the continuations are related at the open type A with the appropriate type variable instantiations, following Girard [16]. The power of this style of logical relation arises from a combination of the extensional flavour of the equivalence and the fact that polymorphic equivalences do not require the same type to be instantiated in both processes, but rather that the types are *related* (via a suitable equivalence candidate relation).

Theorem 2.3 (Properties of Logical Equivalence [7])

Parametricity: *If $\Omega; \Gamma; \Delta \vdash P :: z:A$ then, for all $\omega, \omega' : \Omega$ and $\eta : \omega \Leftrightarrow \omega'$, we have $\Gamma; \Delta \vdash \hat{\omega}(P) \approx_L \hat{\omega}'(P) :: z:A[\eta : \omega \Leftrightarrow \omega']$.*

Soundness: *If $\Omega; \Gamma; \Delta \vdash P \approx_L Q :: z:A$ then $\mathcal{C}[P] \cong \mathcal{C}[Q] :: z:A$, for any closing $\mathcal{C}[-]$.*

Completeness: *If $\Omega; \Gamma; \Delta \vdash P \cong Q :: z:A$ then $\Omega; \Gamma; \Delta \vdash P \approx_L Q :: z:A$.*

3 To Linear-F and Back

We now develop our mutually inverse and fully abstract encodings between $\text{Poly}\pi$ and a linear polymorphic λ -calculus [55] that we dub Linear-F. We first introduce the syntax and typing of the linear λ -calculus and then proceed to detail our encodings and their properties (we omit typing ascriptions from the existential polymorphism constructs for readability).

Definition 3.1 (Linear-F). The syntax of terms M, N and types A, B of Linear-F is given below.

$$\begin{aligned}
M, N &::= \lambda x:A.M \mid M N \mid \langle M \otimes N \rangle \mid \text{let } x \otimes y = M \text{ in } N \mid !M \mid \text{let } !u = M \text{ in } N \mid \Lambda X.M \\
&\quad \mid M[A] \mid \text{pack } A \text{ with } M \mid \text{let } (X, y) = M \text{ in } N \mid \text{let } \mathbf{1} = M \text{ in } N \mid \langle \rangle \mid \mathbf{T} \mid \mathbf{F} \\
A, B &::= A \multimap B \mid A \otimes B \mid !A \mid \forall X.A \mid \exists X.A \mid X \mid \mathbf{1} \mid \mathbf{2}
\end{aligned}$$

The syntax of types is that of the multiplicative and exponential fragments of second-order intuitionistic linear logic: $\lambda x:A.M$ denotes linear λ -abstractions; $M N$ denotes the application; $\langle M \otimes N \rangle$ denotes the multiplicative pairing of M and N , as reflected in its elimination form $\text{let } x \otimes y = M \text{ in } N$ which simultaneously deconstructs the pair M , binding its first and second projection to x and y in N , respectively; $!M$ denotes a term M that does not use any linear variables and so may be used an arbitrary number of times; $\text{let } !u = M \text{ in } N$ binds the underlying exponential term of M as u in N ; $\Lambda X.M$ is the type abstraction former; $M[A]$ stands for type application; $\text{pack } A \text{ with } M$ is the existential type introduction form, where M is a term where the existentially typed variable is instantiated with A ; $\text{let } (X, y) = M \text{ in } N$ unpacks an existential package M , binding the representation type to X and the underlying term to y in N ; the multiplicative unit $\mathbf{1}$ has as introduction form the nullary pair $\langle \rangle$ and is eliminated by the construct $\text{let } \mathbf{1} = M \text{ in } N$, where M is a term of type $\mathbf{1}$. Booleans (type $\mathbf{2}$ with values \mathbf{T} and \mathbf{F}) are the basic observable.

The typing judgment in Linear-F is given as $\Omega; \Gamma; \Delta \vdash M : A$, following the DILL formulation of linear logic [3], stating that term M has type A in a linear context Δ (i.e. bindings for linear variables $x:B$), intuitionistic context Γ (i.e. binding for intuitionistic variables $u:B$) and type variable context Ω . The typing rules are standard [7]. The operational semantics of the calculus are the expected call-by-name semantics with commuting conversions [27]. We write \Downarrow for the evaluation relation. We write \cong for the largest typed congruence that is consistent with the observables of type $\mathbf{2}$ (i.e. a so-called Morris-style equivalence as in [5]).

3.1 Encoding Linear-F into Session π -Calculus

We define a translation from Linear-F to Poly π generalising the one from [49], accounting for polymorphism and multiplicative pairs. We translate typing derivations of λ -terms to those of π -calculus terms (we omit the full typing derivation for the sake of readability).

Proof theoretically, the λ -calculus corresponds to a proof term assignment for natural deduction presentations of logic, whereas the session π -calculus from § 2 corresponds to a proof term assignment for sequent calculus. Thus, we obtain a translation from λ -calculus to the session π -calculus by considering the proof theoretic content of the constructive proof of soundness of the sequent calculus wrt natural deduction. Following Gentzen [14], the translation from natural deduction to sequent calculus maps introduction rules to the corresponding right rules and elimination rules to a combination of the corresponding left rule, cut and/or identity.

Since typing in the session calculus identifies a distinguished channel along which a process offers a session, the translation of λ -terms is parameterised by a “result” channel along which the behaviour of the λ -term is implemented. Given a λ -term M , the process $\llbracket M \rrbracket_z$ encodes the behaviour of M along the session channel z . We enforce that the type **2** of booleans and its two constructors are consistently translated to their polymorphic Church encodings before applying the translation to Poly π . Thus, type **2** is first translated to $\forall X. !X \multimap !X \multimap X$, the value **T** to $\Lambda X. \lambda u. !X. \lambda v. !X. \text{let } !x = u \text{ in let } !y = v \text{ in } x$ and the value **F** to $\Lambda X. \lambda u. !X. \lambda v. !X. \text{let } !x = u \text{ in let } !y = v \text{ in } y$. Such representations of the booleans are adequate up to parametricity [6] and suitable for our purposes of relating the session calculus (which has no primitive notion of value or result type) with the λ -calculus precisely due to the tight correspondence between the two calculi.

Definition 3.2 (From Linear-F to Poly π). $\llbracket \Omega \rrbracket; \llbracket \Gamma \rrbracket; \llbracket \Delta \rrbracket \vdash \llbracket M \rrbracket_z :: z:A$ denotes the translation of contexts, types and terms from Linear-F to the polymorphic session calculus. The translations on contexts and types are the identity function. Booleans and their values are first translated to their Church encodings as specified above. The translation on λ -terms is given below:

$$\begin{array}{lll}
 \llbracket x \rrbracket_z & \triangleq [x \leftrightarrow z] & \llbracket M N \rrbracket_z \triangleq (\nu x)(\llbracket M \rrbracket_x \mid (\nu y)x\langle y \rangle.(\llbracket N \rrbracket_y \mid [x \leftrightarrow z])) \\
 \llbracket u \rrbracket_z & \triangleq (\nu x)u\langle x \rangle.[x \leftrightarrow z] & \llbracket \text{let } !u = M \text{ in } N \rrbracket_z \triangleq (\nu x)(\llbracket M \rrbracket_x \mid \llbracket N \rrbracket_{z\{x/u\}}) \\
 \llbracket \lambda x:A. M \rrbracket_z & \triangleq z(x). \llbracket M \rrbracket_z & \llbracket \langle M \otimes N \rangle \rrbracket_z \triangleq (\nu y)z\langle y \rangle.(\llbracket M \rrbracket_y \mid \llbracket N \rrbracket_z) \\
 \llbracket !M \rrbracket_z & \triangleq !z(x). \llbracket M \rrbracket_x & \llbracket \text{let } x \otimes y = M \text{ in } N \rrbracket_z \triangleq (\nu w)(\llbracket M \rrbracket_w \mid y\langle x \rangle. \llbracket N \rrbracket_z) \\
 \llbracket \Lambda X. M \rrbracket_z & \triangleq z(X). \llbracket M \rrbracket_z & \llbracket M[A] \rrbracket_z \triangleq (\nu x)(\llbracket M \rrbracket_x \mid x\langle A \rangle.[x \leftrightarrow z]) \\
 \llbracket \text{pack } A \text{ with } M \rrbracket_z & \triangleq z\langle A \rangle. \llbracket M \rrbracket_z & \llbracket \text{let } (X, y) = M \text{ in } N \rrbracket_z \triangleq (\nu x)(\llbracket M \rrbracket_y \mid y\langle X \rangle. \llbracket N \rrbracket_z) \\
 \llbracket \langle \rangle \rrbracket_z & \triangleq \mathbf{0} & \llbracket \text{let } \mathbf{1} = M \text{ in } N \rrbracket_z \triangleq (\nu x)(\llbracket M \rrbracket_x \mid \llbracket N \rrbracket_z)
 \end{array}$$

To translate a (linear) λ -abstraction $\lambda x:A. M$, which corresponds to the proof term for the introduction rule for \multimap , we map it to the corresponding $\multimap R$ rule, thus obtaining a process $z(x). \llbracket M \rrbracket_z$ that inputs along the result channel z a channel x which will be used in $\llbracket M \rrbracket_z$ to access the function argument. To encode the application $M N$, we compose (i.e. cut) $\llbracket M \rrbracket_x$, where x is a fresh name, with a process that provides the (encoded) function argument by outputting along x

a channel y which offers the behaviour of $\llbracket N \rrbracket_y$. After the output is performed, the type of x is now that of the function's codomain and thus we conclude by forwarding (i.e. the id rule) between x and the result channel z .

The encoding for polymorphism follows a similar pattern: To encode the abstraction $\lambda X.M$, we receive along the result channel a type that is bound to X and proceed inductively. To encode type application $M[A]$ we encode the abstraction M in parallel with a process that sends A to it, and forwards accordingly. Finally, the encoding of the existential package **pack** A with M maps to an output of the type A followed by the behaviour $\llbracket M \rrbracket_z$, with the encoding of the elimination form $\text{let } (X, y) = M \text{ in } N$ composing the translation of the term of existential type M with a process performing the appropriate type input and proceeding as $\llbracket N \rrbracket_z$.

Example 3.3 (Encoding of Linear-F). Consider the following λ -term corresponding to a polymorphic pairing function (recall that we write $\bar{z}\langle w \rangle.P$ for $(\nu w)z\langle w \rangle.P$):

$$M \triangleq \lambda X. \lambda Y. \lambda x. X. \lambda y. Y. \langle x \otimes y \rangle \text{ and } N \triangleq ((M[A][B] M_1) M_2)$$

Then we have, with $\tilde{x} = x_1 x_2 x_3 x_4$:

$$\begin{aligned} \llbracket N \rrbracket_z &\equiv (\nu \tilde{x}) (\llbracket M \rrbracket_{x_1} \mid x_1 \langle A \rangle. [x_1 \leftrightarrow x_2] \mid x_2 \langle B \rangle. [x_2 \leftrightarrow x_3] \mid \\ &\quad \bar{x}_3 \langle x \rangle. (\llbracket M_1 \rrbracket_x \mid [x_3 \leftrightarrow x_4]) \mid \bar{x}_4 \langle y \rangle. (\llbracket M_2 \rrbracket_y \mid [x_4 \leftrightarrow z])) \\ &\equiv (\nu \tilde{x}) (x_1(X). x_1(Y). x_1(x). x_1(y). \bar{x}_1 \langle w \rangle. ([x \leftrightarrow w] \mid [y \leftrightarrow x_1]) \mid x_1 \langle A \rangle. [x_1 \leftrightarrow x_2] \mid \\ &\quad x_2 \langle B \rangle. [x_2 \leftrightarrow x_3] \mid \bar{x}_3 \langle x \rangle. (\llbracket M_1 \rrbracket_x \mid [x_3 \leftrightarrow x_4]) \mid \bar{x}_4 \langle y \rangle. (\llbracket M_2 \rrbracket_y \mid [x_4 \leftrightarrow z])) \end{aligned}$$

We can observe that $N \rightarrow^+ (((\lambda x. A. \lambda y. B. \langle x \otimes y \rangle) M_1) M_2) \rightarrow^+ \langle M_1 \otimes M_2 \rangle$. At the process level, each reduction corresponding to the redex of type application is simulated by two reductions, obtaining:

$$\begin{aligned} \llbracket N \rrbracket_z &\rightarrow^+ (\nu x_3, x_4) (x_3(x). x_3(y). \bar{x}_3 \langle w \rangle. ([x \leftrightarrow w] \mid [y \leftrightarrow x_3]) \mid \\ &\quad \bar{x}_3 \langle x \rangle. (\llbracket M_1 \rrbracket_x \mid [x_3 \leftrightarrow x_4]) \mid \bar{x}_4 \langle y \rangle. (\llbracket M_2 \rrbracket_y \mid [x_4 \leftrightarrow z])) = P \end{aligned}$$

The reductions corresponding to the β -redexes clarify the way in which the encoding represents substitution of terms for variables via fine-grained name passing. Consider $\llbracket \langle M_1 \otimes M_2 \rangle \rrbracket_z \triangleq \bar{z} \langle w \rangle. (\llbracket M_1 \rrbracket_w \mid \llbracket M_2 \rrbracket_z)$ and

$$P \rightarrow^+ (\nu x, y) (\llbracket M_1 \rrbracket_x \mid \llbracket M_2 \rrbracket_y \mid \bar{z} \langle w \rangle. ([x \leftrightarrow w] \mid [y \leftrightarrow z]))$$

The encoding of the pairing of M_1 and M_2 outputs a fresh name w which will denote the behaviour of (the encoding of) M_1 , and then the behaviour of the encoding of M_2 is offered on z . The reduct of P outputs a fresh name w which is then identified with x and thus denotes the behaviour of $\llbracket M_1 \rrbracket_w$. The channel z is identified with y and thus denotes the behaviour of $\llbracket M_2 \rrbracket_z$, making the two processes listed above equivalent. This informal reasoning exposes the insights that justify the operational correspondence of the encoding. Proof-theoretically, these equivalences simply map to commuting conversions which push the processes $\llbracket M_1 \rrbracket_x$ and $\llbracket M_2 \rrbracket_z$ under the output on z .

Theorem 3.4 (Operational Correspondence)

- If $\Omega; \Gamma; \Delta \vdash M : A$ and $M \rightarrow N$ then $\llbracket M \rrbracket_z \Longrightarrow P$ such that $\llbracket N \rrbracket_z \approx_L P$
- If $\llbracket M \rrbracket_z \rightarrow P$ then $M \rightarrow^+ N$ and $\llbracket N \rrbracket_z \approx_L P$

3.2 Encoding Session π -calculus to Linear-F

Just as the proof theoretic content of the soundness of sequent calculus wrt natural deduction induces a translation from λ -terms to session-typed processes, the *completeness* of the sequent calculus wrt natural deduction induces a translation from the session calculus to the λ -calculus. This mapping identifies sequent calculus right rules with the introduction rules of natural deduction and left rules with elimination rules combined with (type-preserving) substitution. Crucially, the mapping is defined on *typing derivations*, enabling us to consistently identify when a process uses a session (i.e. left rules) or, dually, when a process offers a session (i.e. right rules).

$$\begin{array}{c}
 \left((-\circ R) \frac{\Delta, x:A \vdash P :: z:B}{\Delta \vdash z(x).P :: z:A \multimap B} \right) \triangleq (-\circ I) \frac{\Delta, x:A \vdash \langle P \rangle_{\Delta, x:A \vdash z:B} : B}{\Delta \vdash \lambda x:A. \langle P \rangle_{\Delta, x:A \vdash z:B} : A \multimap B} \\
 \\
 \left((-\circ L) \frac{\Delta_1 \vdash P :: y:A \quad \Delta_2, x:B \vdash Q :: z:C}{\Delta_1, \Delta_2, x:A \multimap B \vdash (\nu y)x\langle y \rangle.(P \mid Q) :: z:C} \right) \triangleq \\
 \text{(SUBST)} \\
 \frac{\Delta_2, x:B \vdash \langle Q \rangle_{\Delta_2, x:B \vdash z:C} : C \quad \frac{(-\circ E) \quad x:A \multimap B \vdash x:A \multimap B \quad \Delta_1 \vdash \langle P \rangle_{\Delta_1 \vdash y:A} : B}{\Delta_1, x:A \multimap B \vdash x \langle P \rangle_{\Delta_1 \vdash y:A} : B}}{\Delta_1, \Delta_2, x:A \multimap B \vdash \langle Q \rangle_{\Delta_2, x:B \vdash z:C} \{ (x \langle P \rangle_{\Delta_1 \vdash y:A}) / x \} : C}
 \end{array}$$

Fig. 3. Translation on typing derivations (excerpt – see [52])

Definition 3.5 (From Poly π to Linear-F). We write $\langle \Omega \rangle; \langle \Gamma \rangle; \langle \Delta \rangle \vdash \langle P \rangle : A$ for the translation from typing derivations in Poly π to derivations in Linear-F. The translations on types and contexts are the identity function. The translation on processes is given below, where the leftmost column indicates the typing rule at the root of the derivation (see Fig. 3 for an excerpt of the translation on typing derivations, where we write $\langle P \rangle_{\Omega; \Gamma; \Delta \vdash z:A}$ to denote the translation of $\Omega; \Gamma; \Delta \vdash P :: z:A$. We omit Ω and Γ when unchanged).

(1R) $\langle 0 \rangle$	$\triangleq \langle \rangle$	$(-\circ L) \langle (\nu y)x\langle y \rangle.(P \mid Q) \rangle$	$\triangleq \langle Q \rangle \{ (x \langle P \rangle) / x \}$
(id) $\langle [x \leftrightarrow y] \rangle$	$\triangleq x$	$(-\circ R) \langle z(x).P \rangle$	$\triangleq \lambda x:A. \langle P \rangle$
(1L) $\langle P \rangle$	$\triangleq \text{let } 1 = x \text{ in } \langle P \rangle$	$(\otimes R) \langle (\nu x)z\langle x \rangle.(P \mid Q) \rangle$	$\triangleq \langle P \rangle \otimes \langle Q \rangle$
(!R) $\langle !z(x).P \rangle$	$\triangleq !\langle P \rangle$	$(\otimes L) \langle x(y).P \rangle$	$\triangleq \text{let } x \otimes y = x \text{ in } \langle P \rangle$
(!L) $\langle P\{u/x\} \rangle$	$\triangleq \text{let } !u = x \text{ in } \langle P \rangle$	(copy) $\langle (\nu x)u\langle x \rangle.P \rangle$	$\triangleq \langle P \rangle \{ u/x \}$
($\forall R$) $\langle z(X).P \rangle$	$\triangleq \lambda X. \langle P \rangle$	($\forall L$) $\langle x(B).P \rangle$	$\triangleq \langle P \rangle \{ (x[B]) / x \}$
($\exists R$) $\langle z\langle B \rangle.P \rangle$	$\triangleq \text{pack } B \text{ with } \langle P \rangle$	($\exists L$) $\langle x(Y).P \rangle$	$\triangleq \text{let } (Y, x) = x \text{ in } \langle P \rangle$
(cut) $\langle (\nu x)(P \mid Q) \rangle$	$\triangleq \langle Q \rangle \{ \langle P \rangle / x \}$	(cut [!]) $\langle (\nu u)(!u(x).P \mid Q) \rangle$	$\triangleq \langle Q \rangle \{ \langle P \rangle / u \}$

For instance, the encoding of a process $z(x).P :: z:A \multimap B$, typed by rule $\multimap R$, results in the corresponding $\multimap I$ introduction rule in the λ -calculus and thus is $\lambda x:A. \langle P \rangle$. To encode the process $(\nu y)x\langle y \rangle.(P \mid Q)$, typed by rule $\multimap L$, we make use of substitution: Given that the sub-process Q is typed as $\Omega; \Gamma; \Delta', x:B \vdash Q :: z:C$, the encoding of the full process is given by $\langle Q \rangle \{ (x \langle P \rangle) / x \}$. The term $x \langle P \rangle$ consists of the application of x (of function type) to the argument $\langle P \rangle$, thus ensuring that the term resulting from the substitution is of the appropriate type. We note that, for instance, the encoding of rule $\otimes L$ does not need to appeal to substitution – the λ -calculus *let* style rules can be mapped directly. Similarly, rule $\forall R$ is mapped to type abstraction, whereas rule $\forall L$ which types a process of the form $x\langle B \rangle.P$ maps to a substitution of the type application $x[B]$ for x in $\langle P \rangle$. The encoding of existential polymorphism is simpler due to the *let*-style elimination. We also highlight the encoding of the cut rule which embodies parallel composition of two processes sharing a linear name, which clarifies the use/offer duality of the intuitionistic calculus – the process that offers P is encoded and substituted into the encoded user Q .

Theorem 3.6. *If $\Omega; \Gamma; \Delta \vdash P :: z:A$ then $\langle \Omega \rangle; \langle \Gamma \rangle; \langle \Delta \rangle \vdash \langle P \rangle : A$.*

Example 3.7 (Encoding of $\text{Poly}\pi$). Consider the following processes

$$P \triangleq z(X).z(Y).z(x).z(y).\bar{z}\langle w \rangle.([x \leftrightarrow w] \mid [y \leftrightarrow z]) \quad Q \triangleq z\langle 1 \rangle.z\langle 1 \rangle.\bar{z}\langle x \rangle.\bar{z}\langle y \rangle.z\langle w \rangle.[w \leftrightarrow r]$$

with $\vdash P :: z:\forall X.\forall Y.X \multimap Y \multimap X \otimes Y$ and $z:\forall X.\forall Y.X \multimap Y \multimap X \otimes Y \vdash Q :: r:1$.

Then: $\langle P \rangle = \lambda X.\lambda Y.\lambda x:X.\lambda y:Y.(x \otimes y) \quad \langle Q \rangle = \text{let } x \otimes y = z[1][1] \langle \rangle \text{ in let } 1 = y \text{ in } x$
 $\langle (\nu z)(P \mid Q) \rangle = \text{let } x \otimes y = (\lambda X.\lambda Y.\lambda x:X.\lambda y:Y.(x \otimes y))[1][1] \langle \rangle \text{ in let } 1 = y \text{ in } x$

By the behaviour of $(\nu z)(P \mid Q)$, which consists of a sequence of cuts, and its encoding, we have that $\langle (\nu z)(P \mid Q) \rangle \rightarrow^+ \langle \rangle$ and $(\nu z)(P \mid Q) \rightarrow^+ \mathbf{0} = \langle \langle \rangle \rangle$.

In general, the translation of Definition 3.5 can introduce some distance between the immediate operational behaviour of a process and its corresponding λ -term, insofar as the translations of cuts (and left rules to non *let*-form elimination rules) make use of substitutions that can take place deep within the resulting term. Consider the process at the root of the following typing judgment $\Delta_1, \Delta_2, \Delta_3 \vdash (\nu x)(x(y).P_1 \mid (\nu y)x\langle y \rangle.(P_2 \mid w(z).\mathbf{0})) :: w:1 \multimap 1$, derivable through a cut on session x between instances of $\multimap R$ and $\multimap L$, where the continuation process $w(z).\mathbf{0}$ offers a session $w:1 \multimap 1$ (and so must use rule $1L$ on x). We have that: $(\nu x)(x(y).P_1 \mid (\nu y)x\langle y \rangle.(P_2 \mid w(z).\mathbf{0})) \rightarrow (\nu x, y)(P_1 \mid P_2 \mid w(z).\mathbf{0})$. However, the translation of the process above results in the term $\lambda z:1.\text{let } 1 = ((\lambda y:A. \langle P_1 \rangle) \langle P_2 \rangle) \text{ in let } 1 = z \text{ in } \langle \rangle$, where the redex that corresponds to the process reduction is present but hidden under the binder for z (corresponding to the input along w). Thus, to establish operational completeness we consider full β -reduction, denoted by \rightarrow_β , i.e. enabling β -reductions under binders.

Theorem 3.8 (Operational Completeness). *Let $\Omega; \Gamma; \Delta \vdash P :: z:A$. If $P \rightarrow Q$ then $\langle P \rangle \rightarrow_\beta^* \langle Q \rangle$.*

In order to study the soundness direction it is instructive to consider typed process $x:1 \multimap 1 \vdash \bar{x}\langle y \rangle.(\nu z)(z(w).0 \mid \bar{z}\langle w \rangle.0) :: v:1$ and its translation:

$$\begin{aligned} \llbracket \bar{x}\langle y \rangle.(\nu z)(z(w).0 \mid \bar{z}\langle w \rangle.0) \rrbracket &= \llbracket (\nu z)(z(w).0 \mid \bar{z}\langle w \rangle.0) \rrbracket \{ (x \langle \rangle) / x \} \\ &= \text{let } 1 = (\lambda w:1. \text{let } 1 = w \text{ in } \langle \rangle) \langle \rangle \text{ in let } 1 = x \langle \rangle \text{ in } \langle \rangle \end{aligned}$$

The process above cannot reduce due to the output prefix on x , which cannot synchronise with a corresponding input action since there is no provider for x (i.e. the channel is in the left-hand side context). However, its encoding can exhibit the β -redex corresponding to the synchronisation along z , hidden by the prefix on x . The corresponding reductions hidden under prefixes in the encoding can be *soundly* exposed in the session calculus by appealing to the commuting conversions of linear logic (e.g. in the process above, the instance of rule $\multimap L$ corresponding to the output on x can be commuted with the cut on z).

As shown in [36], commuting conversions are sound wrt observational equivalence, and thus we formulate operational soundness through a notion of *extended* process reduction, which extends process reduction with the reductions that are induced by commuting conversions. Such a relation was also used for similar purposes in [5] and in [26], in a classical linear logic setting. For conciseness, we define extended reduction as a relation on *typed* processes modulo \equiv .

Definition 3.9 (Extended Reduction [5]). We define \mapsto as the type preserving relations on typed processes modulo \equiv generated by:

1. $\mathcal{C}[(\nu y)x\langle y \rangle.P \mid x(y).Q] \mapsto \mathcal{C}[(\nu y)(P \mid Q)]$;
2. $\mathcal{C}[(\nu y)x\langle y \rangle.P] \mid !x(y).Q \mapsto \mathcal{C}[(\nu y)(P \mid Q)] \mid !x(y).Q$; and
3. $(\nu x)(!x(y).Q) \mapsto 0$

where \mathcal{C} is a (typed) process context which does not capture the bound name y .

Theorem 3.10 (Operational Soundness). *Let $\Omega; \Gamma; \Delta \vdash P :: z:A$ and $\llbracket P \rrbracket \rightarrow M$, there exists Q such that $P \mapsto^* Q$ and $\llbracket Q \rrbracket =_\alpha M$.*

3.3 Inversion and Full Abstraction

Having established the operational preciseness of the encodings to-and-from Poly π and Linear-F, we establish our main results for the encodings. Specifically, we show that the encodings are mutually inverse up-to behavioural equivalence (with *fullness* as its corollary), which then enables us to establish *full abstraction* for *both* encodings.

Theorem 3.11 (Inverse). *If $\Omega; \Gamma; \Delta \vdash M : A$ then $\Omega; \Gamma; \Delta \vdash \llbracket [M]_z \rrbracket \cong M : A$. Also, if $\Omega; \Gamma; \Delta \vdash P :: z:A$ then $\Omega; \Gamma; \Delta \vdash \llbracket [P] \rrbracket_z \approx_L P :: z:A$.*

Corollary 3.12 (Fullness). *Let $\Omega; \Gamma; \Delta \vdash P :: z:A$. $\exists M$ s.t. $\Omega; \Gamma; \Delta \vdash M : A$ and $\Omega; \Gamma; \Delta \vdash \llbracket [M]_z \rrbracket \approx_L P :: z:A$. Also, let $\Omega; \Gamma; \Delta \vdash M : A$. $\exists P$ s.t. $\Omega; \Gamma; \Delta \vdash P :: z:A$ and $\Omega; \Gamma; \Delta \vdash \llbracket [P] \rrbracket \cong M : A$.*

We now state our full abstraction results. Given two Linear-F terms of the same type, equivalence in the image of the $\llbracket - \rrbracket_z$ translation can be used as a proof technique for contextual equivalence in Linear-F. This is called the *soundness* direction of full abstraction in the literature [18] and proved by showing the relation generated by $\llbracket M \rrbracket_z \approx_L \llbracket N \rrbracket_z$ forms \cong ; we then establish the *completeness* direction by contradiction, using fullness.

Theorem 3.13 (Full Abstraction). $\Omega; \Gamma; \Delta \vdash M \cong N : A$ iff $\Omega; \Gamma; \Delta \vdash \llbracket M \rrbracket_z \approx_L \llbracket N \rrbracket_z :: z:A$.

We can straightforwardly combine the above full abstraction with Theorem 3.11 to obtain full abstraction of the $\langle - \rangle$ translation.

Theorem 3.14 (Full Abstraction). $\Omega; \Gamma; \Delta \vdash P \approx_L Q :: z:A$ iff $\Omega; \Gamma; \Delta \vdash \langle P \rangle \cong \langle Q \rangle : A$.

4 Applications of the Encodings

In this section we develop applications of the encodings of the previous sections. Taking advantage of full abstraction and mutual inversion, we apply non-trivial properties from the theory of the λ -calculus to our session-typed process setting.

In § 4.1 we study inductive and coinductive sessions, arising through encodings of initial F -algebras and final F -coalgebras in the polymorphic λ -calculus.

In § 4.2 we study encodings for an extension of the core session calculus with term passing, where terms are derived from a simply-typed λ -calculus. Using the development of § 4.2 as a stepping stone, we generalise the encodings to a *higher-order* session calculus (§ 4.3), where processes can send, receive and execute other processes. We show full abstraction and mutual inversion theorems for the encodings from higher-order to first-order. As a consequence, we can straightforwardly derive a strong normalisation property for the higher-order process-passing calculus.

4.1 Inductive and Coinductive Session Types

The study of polymorphism in the λ -calculus [1, 6, 19, 40] has shown that parametric polymorphism is expressive enough to encode both inductive and coinductive types in a precise way, through a faithful representation of initial and final (co)algebras [28], without extending the language of terms nor the semantics of the calculus, giving a logical justification to the Church encodings of inductive datatypes such as lists and natural numbers. The polymorphic session calculus can express fairly intricate communication behaviours, including generic protocols through both existential and universal polymorphism (i.e. protocols that are parametric in their sub-protocols). Using our fully abstract encodings between the two calculi, we show that session polymorphism is expressive enough to encode inductive and coinductive sessions, “importing” the results for the λ -calculus, which may then be instantiated to provide a session-typed formulation of the encodings of data structures in the π -calculus of [30].

Inductive and Coinductive Types in System F. Exploring an algebraic interpretation of polymorphism where types are interpreted as functors, it can be shown that given a type F with a free variable X that occurs only positively (i.e. occurrences of X are on the left-hand side of an even number of function arrows), the polymorphic type $\forall X.((F(X) \rightarrow X) \rightarrow X)$ forms an initial F -algebra [1, 42] (we write $F(X)$ to denote that X occurs in F). This enables the representation of *inductively* defined structures using an algebraic or categorical justification. For instance, the natural numbers can be seen as the initial F -algebra of $F(X) = \mathbf{1} + X$ (where $\mathbf{1}$ is the unit type and $+$ is the coproduct), and are thus *already present* in System F, in a precise sense, as the type $\forall X.((\mathbf{1} + X) \rightarrow X) \rightarrow X$ (noting that both $\mathbf{1}$ and $+$ can also be encoded in System F). A similar story can be told for *coinductively* defined structures, which correspond to final F -coalgebras and are representable with the polymorphic type $\exists X.(X \rightarrow F(X)) \times X$, where \times is a product type. In the remainder of this section we assume the positivity requirement on F mentioned above.

While the complete formal development of the representation of inductive and coinductive types in System F would lead us to far astray, we summarise here the key concepts as they apply to the λ -calculus (the interested reader can refer to [19] for the full categorical details).

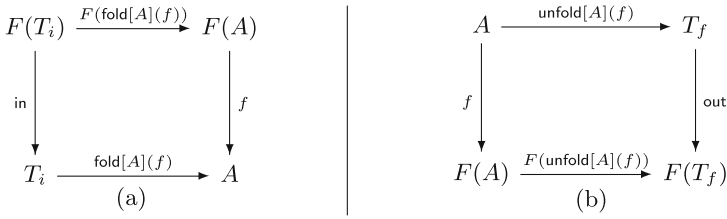


Fig. 4. Diagrams for initial F -algebras and final F -coalgebras

To show that the polymorphic type $T_i \triangleq \forall X.((F(X) \rightarrow X) \rightarrow X)$ is an initial F -algebra, one exhibits a pair of λ -terms, often dubbed *fold* and *in*, such that the diagram in Fig. 4(a) commutes (for any A , where $F(f)$, where f is a λ -term, denotes the functorial action of F applied to f), and, crucially, that *fold* is *unique*. When these conditions hold, we are justified in saying that T_i is a least fixed point of F . Through a fairly simple calculation, it is easy to see that:

$$\begin{aligned} \text{fold} &\triangleq \Lambda X. \lambda x. F(X) \rightarrow X. \lambda t. T_i. t[X](x) \\ \text{in} &\triangleq \lambda x. F(T_i). \Lambda X. \lambda y. F(X) \rightarrow X. y (F(\text{fold}[X])(x))(x) \end{aligned}$$

satisfy the necessary equalities. To show uniqueness one appeals to *parametricity*, which allows us to prove that any function of the appropriate type is equivalent to *fold*. This property is often dubbed *initiality* or *universality*.

The construction of final F -coalgebras and their justification as *greatest* fixed points is dual. Assuming products in the calculus and taking $T_f \triangleq \exists X.(X \rightarrow$

$F(X)) \times X$, we produce the λ -terms

$$\begin{aligned} \text{unfold} &\triangleq \Lambda X. \lambda f. X \rightarrow F(X). \lambda x. T_f. \text{pack } X \text{ with } (f, x) \\ \text{out} &\triangleq \lambda t : T_f. \text{let } (X, (f, x)) = t \text{ in } F(\text{unfold}[X](f))(f(x)) \end{aligned}$$

such that the diagram in Fig. 4(b) commutes and **unfold** is unique (again, up to parametricity). While the argument above applies to System F, a similar development can be made in Linear-F [6] by considering $T_i \triangleq \forall X. !(F(X) \multimap X) \multimap X$ and $T_f \triangleq \exists X. !(X \multimap F(X)) \otimes X$. Reusing the same names for the sake of conciseness, the associated *linear* λ -terms are:

$$\begin{aligned} \text{fold} &\triangleq \Lambda X. \lambda u. !(F(X) \multimap X). \lambda y. T_i. (y[X] u) : \forall X. !(F(X) \multimap X) \multimap T_i \multimap X \\ \text{in} &\triangleq \lambda x. F(T_i). \Lambda X. \lambda y. !(F(X) \multimap X). \text{let } !u = y \text{ in } k(F(\text{fold}[X](!u))(x)) : F(T_i) \multimap T_i \\ \text{unfold} &\triangleq \Lambda X. \lambda u. !(X \multimap F(X)). \lambda x. X. \text{pack } X \text{ with } (u \otimes x) : \forall X. !(X \multimap F(X)) \multimap X \multimap T_f \\ \text{out} &\triangleq \lambda t : T_f. \text{let } (X, (u, x)) = t \text{ in let } !f = u \text{ in } F(\text{unfold}[X](!f))(f(x)) : T_f \multimap F(T_f) \end{aligned}$$

Inductive and Coinductive Sessions for Free. As a consequence of full abstraction we may appeal to the $\llbracket - \rrbracket_z$ encoding to derive representations of **fold** and **unfold** that satisfy the necessary algebraic properties. The derived processes are (recall that we write $\bar{x}\langle y \rangle. P$ for $(\nu y)x\langle y \rangle. P$):

$$\begin{aligned} \llbracket \text{fold} \rrbracket_z &\triangleq z(X). z(u). z(y). (\nu v)((\nu x)([y \leftrightarrow x] \mid x\langle X \rangle. [x \leftrightarrow w]) \mid \bar{w}\langle v \rangle. ([u \leftrightarrow v] \mid [w \leftrightarrow z])) \\ \llbracket \text{unfold} \rrbracket_z &\triangleq z(X). z(u). z(x). z\langle X \rangle. \bar{z}\langle y \rangle. ([u \leftrightarrow y] \mid [x \leftrightarrow z]) \end{aligned}$$

We can then show universality of the two constructions. We write $P_{x,y}$ to single out that x and y are free in P and $P_{z,w}$ to denote the result of employing capture-avoiding substitution on P , substituting x and y by z and w . Let:

$$\begin{aligned} \text{foldP}(A)_{y_1, y_2} &\triangleq (\nu x)(\llbracket \text{fold} \rrbracket_x \mid x\langle A \rangle. \bar{x}\langle v \rangle. (\bar{u}\langle y \rangle. [y \leftrightarrow v] \mid \bar{x}\langle z \rangle. ([z \leftrightarrow y_1] \mid [x \leftrightarrow y_2]))) \\ \text{unfoldP}(A)_{y_1, y_2} &\triangleq (\nu x)(\llbracket \text{unfold} \rrbracket_x \mid x\langle A \rangle. \bar{x}\langle v \rangle. (\bar{u}\langle y \rangle. [y \leftrightarrow v] \mid \bar{x}\langle z \rangle. ([z \leftrightarrow y_1] \mid [x \leftrightarrow y_2]))) \end{aligned}$$

where $\text{foldP}(A)_{y_1, y_2}$ corresponds to the application of **fold** to an F -algebra A with the associated morphism $F(A) \multimap A$ available on the shared channel u , consuming an ambient session $y_1 : T_i$ and offering $y_2 : A$. Similarly, $\text{unfoldP}(A)_{y_1, y_2}$ corresponds to the application of **unfold** to an F -coalgebra A with the associated morphism $A \multimap F(A)$ available on the shared channel u , consuming an ambient session $y_1 : A$ and offering $y_2 : T_f$.

Theorem 4.1 (Universality of foldP). $\forall Q$ such that $X; u : F(X) \multimap X; y_1 : T_i \vdash Q :: y_2 : X$ we have $X; u : F(X) \multimap X; y_1 : T_i \vdash Q \approx_L \text{foldP}(X)_{y_1, y_2} :: y_2 : X$

Theorem 4.2 (Universality of unfoldP). $\forall Q$ and F -coalgebra A s.t. $\cdot; y_1 : A \vdash Q :: y_2 : T_f$ we have that $\cdot; u : F(A) \multimap A; y_1 : A \vdash Q \approx_L \text{unfoldP}(A)_{y_1, y_2} :: y_2 : T_f$.

Example 4.3 (Natural Numbers). We show how to represent the natural numbers as an inductive session type using $F(X) = \mathbf{1} \oplus X$, making use of **in**:

$$\text{zero}_x \triangleq (\nu z)(z. \text{in}!; \mathbf{0} \mid \llbracket \text{in}(z) \rrbracket_x) \quad \text{succ}_{y,x} \triangleq (\nu s)(s. \text{in}r; [y \leftrightarrow s] \mid \llbracket \text{in}(s) \rrbracket_x)$$

with $\text{Nat} \triangleq \forall X. !((\mathbf{1} \oplus X) \multimap X) \multimap X$ where $\vdash \text{zero}_x :: x:\text{Nat}$ and $y:\text{Nat} \vdash \text{succ}_{y,x} :: x:\text{Nat}$ encode the representation of 0 and successor, respectively. The natural 1 would thus be represented by $\text{one}_x \triangleq (\nu y)(\text{zero}_y \mid \text{succ}_{y,x})$. The behaviour of type Nat can be seen as a that of a sequence of internal choices of arbitrary (but finite) length. We can then observe that the foldP process acts as a recursor. For instance consider:

$$\text{stepDec}_d \triangleq d(n).n.\text{case}(\text{zero}_d, [n \leftrightarrow d]) \quad \text{dec}_{x,z} \triangleq (\nu u) (!u(d).\text{stepDec}_d \mid \text{foldP}(\text{Nat})_{x,z})$$

with $\text{stepDec}_d :: d:(\mathbf{1} \oplus \text{Nat}) \multimap \text{Nat}$ and $x:\text{Nat} \vdash \text{dec}_{x,z} :: z:\text{Nat}$, where dec decrements a given natural number session on channel x . We have that:

$$(\nu x)(\text{one}_x \mid \text{dec}_{x,z}) \equiv (\nu x, y, u)(\text{zero}_y \mid \text{succ}_{y,x} !u(d).\text{stepDec}_d \mid \text{foldP}(\text{Nat})_{x,z}) \approx_{\mathbb{L}} \text{zero}_z$$

We note that the resulting encoding is reminiscent of the encoding of lists of [30] (where zero is the empty list and succ the cons cell). The main differences in the encodings arise due to our primitive notions of labels and forwarding, as well as due to the generic nature of in and fold .

Example 4.4 (Streams). We build on Example 4.3 by representing *streams* of natural numbers as a coinductive session type. We encode infinite streams of naturals with $F(X) = \text{Nat} \otimes X$. Thus: $\text{NatStream} \triangleq \exists X. !(X \multimap (\text{Nat} \otimes X)) \otimes X$. The behaviour of a session of type NatStream amounts to an infinite sequence of outputs of channels of type Nat . Such an encoding enables us to construct the stream of all naturals nats (and the stream of all non-zero naturals oneNats):

$$\begin{aligned} \text{genHdNext}_z &\triangleq z(n).\bar{z}\langle y \rangle.(\bar{n}\langle n' \rangle.[n' \leftrightarrow y] \mid !z(w).\bar{n}\langle n' \rangle.\text{succ}_{n',w}) \\ \text{nats}_y &\triangleq (\nu x, u)(\text{zero}_x \mid !u(z).\text{genHdNext}_z \mid \text{unfoldP}(!\text{Nat})_{x,y}) \\ \text{oneNats}_y &\triangleq (\nu x, u)(\text{one}_x \mid !u(z).\text{genHdNext}_z \mid \text{unfoldP}(!\text{Nat})_{x,y}) \end{aligned}$$

with $\text{genHdNext}_z :: z:\text{Nat} \multimap \text{Nat} \otimes !\text{Nat}$ and both nats_y and $\text{oneNats} :: y:\text{NatStream}$. genHdNext_z consists of a helper that generates the current head of a stream and the next element. As expected, the following process implements a session that “unrolls” the stream once, providing the head of the stream and then behaving as the rest of the stream (recall that $\text{out} : T_f \multimap F(T_f)$).

$$(\nu x)(\text{nats}_x \mid \llbracket \text{out}(x) \rrbracket_y) :: y:\text{Nat} \otimes \text{NatStream}$$

We note a peculiarity of the interaction of linearity with the stream encoding: a process that begins to deconstruct a stream has no way of “bottoming out” and stopping. One cannot, for instance, extract the first element of a stream of naturals and stop unrolling the stream in a well-typed way. We can, however, easily encode a “terminating” stream of all natural numbers via $F(X) = (\text{Nat} \otimes !X)$ by replacing the genHdNext_z with the generator given as:

$$\text{genHdNextTer}_z \triangleq z(n).\bar{z}\langle y \rangle.(\bar{n}\langle n' \rangle.[n' \leftrightarrow y] \mid !z(w).!w(w').\bar{n}\langle n' \rangle.\text{succ}_{n',w'})$$

It is then easy to see that a usage of $\llbracket \text{out}(x) \rrbracket_y$ results in a session of type $\text{Nat} \otimes !\text{NatStream}$, enabling us to discard the stream as needed. One can replay

this argument with the operator $F(X) = (!\text{Nat} \otimes X)$ to enable discarding of stream elements. Assuming such modifications, we can then show:

$$(\nu y)((\nu x)(\text{nats}_x \mid \llbracket \text{out}(x) \rrbracket_y) \mid y(n).[y \leftrightarrow z]) \approx_L \text{oneNats}_z :: z:\text{NatStream}$$

4.2 Communicating Values – Sess $\pi\lambda$

We now consider a session calculus extended with a data layer obtained from a λ -calculus (whose terms are ranged over by M, N and types by τ, σ). We dub this calculus Sess $\pi\lambda$.

$$\begin{array}{ll} P, Q & :: = \cdots \mid x\langle M \rangle.P \mid x(y).P \\ M, N & :: = \lambda x:\tau.M \mid MN \mid x \end{array} \quad \begin{array}{ll} A, B & :: = \cdots \mid \tau \wedge A \mid \tau \supset A \\ \tau, \sigma & :: = \cdots \mid \tau \rightarrow \sigma \end{array}$$

Without loss of generality, we consider the data layer to be simply-typed, with a call-by-name semantics, satisfying the usual type safety properties. The typing judgment for this calculus is $\Psi \vdash M : \tau$. We omit session polymorphism for the sake of conciseness, restricting processes to communication of data and (session) channels. The typing judgment for processes is thus modified to $\Psi; \Gamma; \Delta \vdash P :: z:A$, where Ψ is an intuitionistic context that accounts for variables in the data layer. The rules for the relevant process constructs are (all other rules simply propagate the Ψ context from conclusion to premises):

$$\begin{array}{c} \frac{\Psi \vdash M : \tau \quad \Psi; \Gamma; \Delta \vdash P :: z:A}{\Psi; \Gamma; \Delta \vdash z\langle M \rangle.P :: z:\tau \wedge A} (\wedge R) \quad \frac{\Psi, y:\tau; \Gamma; \Delta, x:A \vdash Q :: z:C}{\Psi; \Gamma; \Delta, x:\tau \wedge A \vdash x(y).Q :: z:C} (\wedge L) \\[10pt] \frac{\Psi, x:\tau; \Gamma; \Delta \vdash P :: z:A}{\Psi; \Gamma; \Delta \vdash z(x).P :: z:\tau \supset A} (\supset R) \quad \frac{\Psi \vdash M : \tau \quad \Psi; \Gamma; \Delta, x:A \vdash Q :: z:C}{\Psi; \Gamma; \Delta, x:\tau \supset A \vdash x\langle M \rangle.Q :: z:C} (\supset L) \end{array}$$

With the reduction rule given by:¹ $x\langle M \rangle.P \mid x(y).Q \rightarrow P \mid Q\{M/y\}$. With a simple extension to our encodings we may eliminate the data layer by encoding the data objects as processes, showing that from an expressiveness point of view, data communication is orthogonal to the framework. We note that the data language we are considering is *not* linear, and the usage discipline of data in processes is itself also not linear.

To First-Order Processes. We now introduce our encoding for Sess $\pi\lambda$, defined inductively on session types, processes, types and λ -terms (we omit the purely inductive cases on session types and processes for conciseness). As before, the encoding on processes is defined on *typing derivations*, where we indicate the typing rule at the root of the typing derivation.

$$\begin{array}{lll} \llbracket \tau \wedge A \rrbracket \triangleq !\llbracket \tau \rrbracket \otimes \llbracket A \rrbracket & \llbracket \tau \supset A \rrbracket \triangleq !\llbracket \tau \rrbracket \multimap \llbracket A \rrbracket & \llbracket \tau \rightarrow \sigma \rrbracket \triangleq !\llbracket \tau \rrbracket \multimap \llbracket \sigma \rrbracket \\[10pt] (\wedge R) \llbracket z\langle M \rangle.P \rrbracket \triangleq \bar{z}\langle x \rangle.(!x(y).\llbracket M \rrbracket_y \mid \llbracket P \rrbracket) & (\wedge L) \llbracket x(y).P \rrbracket \triangleq x(y).\llbracket P \rrbracket & \\ (\supset R) \llbracket z(x).P \rrbracket \triangleq z(x).\llbracket P \rrbracket & (\supset L) \llbracket x\langle M \rangle.P \rrbracket \triangleq \bar{x}\langle y \rangle.(!y(w).\llbracket M \rrbracket_w \mid \llbracket P \rrbracket) & \end{array}$$

¹ For simplicity, in this section, we define the process semantics through a reduction relation.

$$\begin{aligned} \llbracket x \rrbracket_z &\triangleq \bar{x}\langle y \rangle. [y \leftrightarrow z] & \llbracket \lambda x:\tau. M \rrbracket_z &\triangleq z(x). \llbracket M \rrbracket_z \\ \llbracket M N \rrbracket_z &\triangleq (\nu y)(\llbracket M \rrbracket_y \mid \bar{y}\langle x \rangle. (!x(w). \llbracket N \rrbracket_w \mid [y \leftrightarrow z])) \end{aligned}$$

The encoding addresses the non-linear usage of data elements in processes by encoding the types $\tau \wedge A$ and $\tau \supset A$ as $!\llbracket \tau \rrbracket \otimes \llbracket A \rrbracket$ and $!\llbracket \tau \rrbracket \multimap \llbracket A \rrbracket$, respectively. Thus, sending and receiving of data is codified as the sending and receiving of channels of type $!$, which therefore can be used non-linearly. Moreover, since data terms are themselves non-linear, the $\tau \rightarrow \sigma$ type is encoded as $!\llbracket \tau \rrbracket \multimap \llbracket \sigma \rrbracket$, following Girard's embedding of intuitionistic logic in linear logic [15].

At the level of processes, offering a session of type $\tau \wedge A$ (i.e. a process of the form $z\langle M \rangle. P$) is encoded according to the translation of the type: we first send a *fresh* name x which will be used to access the encoding of the term M . Since M can be used an arbitrary number of times by the receiver, we guard the encoding of M with a replicated input, proceeding with the encoding of P accordingly. Using a session of type $\tau \supset A$ follows the same principle. The input cases (and the rest of the process constructs) are completely homomorphic.

The encoding of λ -terms follows Girard's decomposition of the intuitionistic function space [49]. The λ -abstraction is translated as input. Since variables in a λ -abstraction may be used non-linearly, the case for variables and application is slightly more intricate: to encode the application $M N$ we compose M in parallel with a process that will send the “reference” to the function argument N which will be encoded using replication, in order to handle the potential for 0 or more usages of variables in a function body. Respectively, a variable is encoded by performing an output to trigger the replication and forwarding accordingly. Without loss of generality, we assume variable names and their corresponding replicated counterparts match, which can be achieved through α -conversion before applying the translation. We exemplify our encoding as follows:

$$\begin{aligned} \llbracket z(x). z\langle x \rangle. z\langle (\lambda y:\sigma. x) \rangle. \mathbf{0} \rrbracket &= z(x). \bar{z}\langle w \rangle. (!w(u). \llbracket x \rrbracket_u \mid \bar{z}\langle v \rangle. (!v(i). \llbracket \lambda y:\sigma. x \rrbracket_i \mid \mathbf{0})) \\ &= z(x). \bar{z}\langle w \rangle. (!w(u). \bar{x}\langle y \rangle. [y \leftrightarrow u] \mid \bar{z}\langle v \rangle. (!v(i). i(y). \bar{x}\langle t \rangle. [t \leftrightarrow i] \mid \mathbf{0})) \end{aligned}$$

Properties of the Encoding. We discuss the correctness of our encoding. We can straightforwardly establish that the encoding preserves typing.

To show that our encoding is operationally sound and complete, we capture the interaction between substitution on λ -terms and the encoding into processes through logical equivalence. Consider the following reduction of a process:

$$\begin{aligned} (\nu z)(z(x). z\langle x \rangle. z\langle (\lambda y:\sigma. x) \rangle. \mathbf{0} \mid z\langle \lambda w:\tau_0. w \rangle. P) \\ \rightarrow (\nu z)(z\langle \lambda w:\tau_0. w \rangle. z\langle (\lambda y:\sigma. \lambda w:\tau_0. w) \rangle. \mathbf{0} \mid P) \end{aligned} \quad (1)$$

Given that substitution in the target session π -calculus amounts to renaming, whereas in the λ -calculus we replace a variable for a term, the relationship between the encoding of a substitution $M\{N/x\}$ and the encodings of M and N corresponds to the composition of the encoding of M with that of N , but where the encoding of N is guarded by a replication, codifying a form of explicit non-linear substitution.

Lemma 4.5 (Compositionality). *Let $\Psi, x:\tau \vdash M : \sigma$ and $\Psi \vdash N : \tau$. We have that $\llbracket M\{N/x\} \rrbracket_z \approx_L (\nu x)(\llbracket M \rrbracket_z \mid !x(y). \llbracket N \rrbracket_y)$*

Revisiting the process to the left of the arrow in Eq. 1 we have:

$$\begin{aligned} & \llbracket (\nu z)(z(x).z\langle x \rangle.z\langle (\lambda y:\sigma.x) \rangle.\mathbf{0} \mid z\langle \lambda w:\tau_0.w \rangle.P) \rrbracket \\ &= (\nu z)(\llbracket z(x).z\langle x \rangle.z\langle (\lambda y:\sigma.x) \rangle.\mathbf{0} \rrbracket_z \mid \bar{z}\langle x \rangle.(!x(b). \llbracket \lambda w:\tau_0.w \rrbracket_b \mid \llbracket P \rrbracket)) \\ &\rightarrow (\nu z, x)(\bar{z}\langle w \rangle.(!w(u).\bar{x}\langle y \rangle.[y \leftrightarrow u] \mid \bar{z}\langle v \rangle.(!v(i). \llbracket \lambda y:\sigma.x \rrbracket_i \mid \mathbf{0}) \mid !x(b). \llbracket \lambda w:\tau_0.w \rrbracket_b \mid \llbracket P \rrbracket)) \end{aligned}$$

whereas the process to the right of the arrow is encoded as:

$$\begin{aligned} & \llbracket (\nu z)(z\langle \lambda w:\tau_0.w \rangle.z\langle (\lambda y:\sigma.\lambda w:\tau_0.w) \rangle.\mathbf{0} \mid P) \rrbracket \\ &= (\nu z)(\bar{z}\langle w \rangle.(!w(u). \llbracket \lambda w:\tau_0.w \rrbracket_u \mid \bar{z}\langle v \rangle.(!v(i). \llbracket \lambda y:\sigma.\lambda w:\tau_0.w \rrbracket_i \mid \llbracket P \rrbracket))) \end{aligned}$$

While the reduction of the encoded process and the encoding of the reduct differ syntactically, they are observationally equivalent – the latter inlines the replicated process behaviour that is accessible in the former on x . Having characterised substitution, we establish operational correspondence for the encoding.

Theorem 4.6 (Operational Correspondence)

1. If $\Psi \vdash M : \tau$ and $\llbracket M \rrbracket_z \rightarrow Q$ then $M \rightarrow^+ N$ such that $\llbracket N \rrbracket_z \approx_L Q$
2. If $\Psi; \Gamma; \Delta \vdash P :: z:A$ and $\llbracket P \rrbracket \rightarrow Q$ then $P \rightarrow^+ P'$ such that $\llbracket P' \rrbracket \approx_L Q$
3. If $\Psi \vdash M : \tau$ and $M \rightarrow N$ then $\llbracket M \rrbracket_z \Longrightarrow P$ such that $P \approx_L \llbracket N \rrbracket_z$
4. If $\Psi; \Gamma; \Delta \vdash P :: z:A$ and $P \rightarrow Q$ then $\llbracket P \rrbracket \rightarrow^+ R$ with $R \approx_L \llbracket Q \rrbracket$

The process equivalence in Theorem 4.6 above need not be extended to account for data (although it would be relatively simple to do so), since the processes in the image of the encoding are fully erased of any data elements.

Back to λ -Terms. We extend our encoding of processes to λ -terms to $\text{Sess}\pi\lambda$. Our extended translation maps processes to linear λ -terms, with the session type $\tau \wedge A$ interpreted as a pair type where the first component is replicated. Dually, $\tau \supset A$ is interpreted as a function type where the domain type is replicated. The remaining session constructs are translated as in § 3.2.

$$\llbracket \tau \wedge A \rrbracket \triangleq !\langle \tau \rangle \otimes \langle A \rangle \quad \llbracket \tau \supset A \rrbracket \triangleq !\langle \tau \rangle \multimap \langle A \rangle \quad \llbracket \tau \rightarrow \sigma \rrbracket \triangleq !\langle \tau \rangle \multimap \langle \sigma \rangle$$

$$\begin{aligned} (\wedge L) \quad \llbracket x(y).P \rrbracket &\triangleq \text{let } y \otimes x = x \text{ in let } !y = y \text{ in } \langle P \rangle & (\wedge R) \quad \llbracket z\langle M \rangle.P \rrbracket &\triangleq \langle !\langle M \rangle \otimes \langle P \rangle \rangle \\ (\supset L) \quad \llbracket x(y).P \rrbracket &\triangleq \lambda x.!\langle \tau \rangle.\text{let } !x = x \text{ in } \langle P \rangle & (\supset R) \quad \llbracket x\langle M \rangle.P \rrbracket &\triangleq \langle P \rangle \{ (x \mid \langle M \rangle) / x \} \end{aligned}$$

$$\llbracket \lambda x:\tau.M \rrbracket \triangleq \lambda x.!\langle \tau \rangle.\text{let } !x = x \text{ in } \langle M \rangle \quad \llbracket M N \rrbracket \triangleq \langle M \rangle \mid \langle N \rangle \quad \llbracket x \rrbracket \triangleq x$$

The treatment of non-linear components of processes is identical to our previous encoding: non-linear functions $\tau \rightarrow \sigma$ are translated to linear functions of type $!\tau \multimap \sigma$; a process offering a session of type $\tau \wedge A$ (i.e. a process of the form $z\langle M \rangle.P$, typed by rule $\wedge R$) is translated to a pair where the first component is the encoding of M prefixed with $!$ so that it may be used non-linearly, and the second is the encoding of P . Non-linear variables are handled at the respective

binding sites: a process using a session of type $\tau \wedge A$ is encoded using the elimination form for the pair and the elimination form for the exponential; similarly, a process offering a session of type $\tau \supset A$ is encoded as a λ -abstraction where the bound variable is of type $!(\tau)$. Thus, we use the elimination form for the exponential, ensuring that the typing is correct. We illustrate our encoding:

$$\begin{aligned} \llbracket z(x).z\langle x \rangle.z\langle (\lambda y:\sigma.x) \rangle.0 \rrbracket &= \lambda x:!(\tau).\text{let } !x = x \text{ in } \langle !x \otimes \langle !(\lambda y:\sigma.x) \rangle \otimes \langle \rangle \rangle \\ &= \lambda x:!(\tau).\text{let } !x = x \text{ in } \langle !x \otimes \langle !(\lambda y:!(\sigma)).\text{let } !y = y \text{ in } x \rangle \otimes \langle \rangle \rangle \end{aligned}$$

Properties of the Encoding. Unsurprisingly due to the logical correspondence between natural deduction and sequent calculus presentations of logic, our encoding satisfies both type soundness and operational correspondence (c.f. Theorems 3.6, 3.8 and 3.10). The full development can be found in [52].

Relating the Two Encodings. We prove the two encodings are mutually inverse and preserve the full abstraction properties (we write $=_\beta$ and $=_{\beta\eta}$ for β - and $\beta\eta$ -equivalence, respectively).

Theorem 4.7 (Inverse). *If $\Psi; \Gamma; \Delta \vdash P :: z:A$ then $\llbracket \langle P \rangle \rrbracket_z \approx_L \llbracket P \rrbracket$. Also, if $\Psi \vdash M : \tau$ then $\llbracket \llbracket M \rrbracket_z \rrbracket =_\beta \llbracket M \rrbracket$.*

The equivalences above are formulated between the composition of the encodings applied to P (resp. M) and the process (resp. λ -term) *after* applying the translation embedding the non-linear components into their linear counterparts. This formulation matches more closely that of § 3.3, which applies to linear calculi for which the *target* languages of this section are a strict subset (and avoids the formalisation of process equivalence with terms). We also note that in this setting, observational equivalence and $\beta\eta$ -equivalence coincide [3, 31]. Moreover, the extensional flavour of \approx_L includes η -like principles at the process level.

Theorem 4.8. *Let $\cdot \vdash M : \tau$ and $\cdot \vdash N : \tau$. $\llbracket M \rrbracket =_{\beta\eta} \llbracket N \rrbracket$ iff $\llbracket M \rrbracket_z \approx_L \llbracket N \rrbracket_z$. Also, let $\cdot \vdash P :: z:A$ and $\cdot \vdash Q :: z:A$. We have that $\llbracket P \rrbracket \approx_L \llbracket Q \rrbracket$ iff $\langle P \rangle =_{\beta\eta} \langle Q \rangle$.*

We establish full abstraction for the encoding of λ -terms into processes (Theorem 4.8) in two steps: The completeness direction (i.e. from left-to-right) follows from operational completeness and strong normalisation of the λ -calculus. The soundness direction uses operational soundness. The proof of Theorem 4.8 uses the same strategy of Theorem 3.14, appealing to the inverse theorems.

4.3 Higher-Order Session Processes – $\text{Sess}\pi\lambda^+$

We extend the value-passing framework of the previous section, accounting for process-passing (i.e. the higher-order) in a session-typed setting. As shown in [50], we achieve this by adding to the data layer a *contextual monad* that encapsulates (open) session-typed processes as data values, with a corresponding elimination form in the process layer. We dub this calculus $\text{Sess}\pi\lambda^+$.

$$\begin{aligned} P, Q &:: \dots \mid x \leftarrow M \leftarrow \overline{y_i}; Q & M.N &:: \dots \mid \{x \leftarrow P \leftarrow \overline{y_i:A_i}\} \\ \tau, \sigma &:: \dots \mid \{x_j:A_j \vdash z:A\} \end{aligned}$$

The type $\{\overline{x_j:A_j} \vdash z:A\}$ is the type of a term which encapsulates an open process that uses the linear channels $\overline{x_j:A_j}$ and offers A along channel z . This formulation has the added benefit of formalising the integration of session-typed processes in a functional language and forms the basis for the concurrent programming language **SILL** [37, 50]. The typing rules for the new constructs are (for simplicity we assume no shared channels in process monads):

$$\frac{\Psi; \cdot; \overline{x_i:A_i} \vdash P :: z:A}{\Psi \vdash \{z \leftarrow P \leftarrow \overline{x_i:A_i}\} : \{\overline{x_i:A_i} \vdash z:A\}} \{\}I$$

$$\frac{\Psi \vdash M : \{\overline{x_i:A_i} \vdash x:A\} \quad \Delta_1 = \overline{y_i:A_i} \quad \Psi; \Gamma; \Delta_2, x:A \vdash Q :: z:C}{\Psi; \Gamma; \Delta_1, \Delta_2 \vdash x \leftarrow M \leftarrow \overline{y_i}; Q :: z:C} \{\}E$$

Rule $\{\}I$ embeds processes in the term language by essentially quoting an open process that is well-typed according to the type specification in the monadic type. Dually, rule $\{\}E$ allows for processes to use monadic values through composition that *consumes* some of the ambient channels in order to provide the monadic term with the necessary context (according to its type). These constructs are discussed in substantial detail in [50]. The reduction semantics of the process construct is given by (we tacitly assume that the names \overline{y} and c do not occur in P and omit the congruence case):

$$(c \leftarrow \{z \leftarrow P \leftarrow \overline{x_i:A_i}\} \leftarrow \overline{y_i}; Q) \rightarrow (\nu c)(P\{\overline{y}/\overline{x_i}\{c/z\}\} \mid Q)$$

The semantics allows for the underlying monadic term M to evaluate to a (quoted) process P . The process P is then executed in parallel with the continuation Q , sharing the linear channel c for subsequent interactions. We illustrate the higher-order extension with following typed process (we write $\{x \leftarrow P\}$ when P does not depend on any linear channels and assume $\vdash Q :: d:\text{Nat} \wedge 1$):

$$P \triangleq (\nu c)(c(\{d \leftarrow Q\}).c(x).\mathbf{0} \mid c(y).d \leftarrow y; d(n).c(n).\mathbf{0}) \quad (2)$$

Process P above gives an abstract view of a communication idiom where a process (the left-hand side of the parallel composition) sends another process Q which potentially encapsulates some complex computation. The receiver then *spawns* the execution of the received process and inputs from it a result value that is sent back to the original sender. An execution of P is given by:

$$\begin{aligned} P &\rightarrow (\nu c)(c(x).\mathbf{0} \mid d \leftarrow \{d \leftarrow Q\}; d(n).c(n).\mathbf{0}) \rightarrow (\nu c)(c(x).\mathbf{0} \mid (\nu d)(Q \mid d(n).c(n).\mathbf{0})) \\ &\rightarrow^+ (\nu c)(c(x).\mathbf{0} \mid c(42).\mathbf{0}) \rightarrow \mathbf{0} \end{aligned}$$

Given the seminal work of Sangiorgi [46], such a representation naturally begs the question of whether or not we can develop a *typed* encoding of higher-order processes into the first-order setting. Indeed, we can achieve such an encoding with a fairly simple extension of the encoding of § 4.2 to $\text{Sess}\pi\lambda^+$ by observing that monadic values are processes that need to be potentially provided with extra sessions in order to be executed correctly. For instance, a term of type $\{x:A \vdash y:B\}$ denotes a process that given a session x of type A will then offer $y:B$. Exploiting this observation we encode this type as the session $A \multimap B$, ensuring subsequent usages of such a term are consistent with this interpretation.

$$\begin{aligned}
\llbracket \{\overline{x_j:A_j} \vdash z:A\} \rrbracket &\triangleq \llbracket \overline{A_j} \rrbracket \multimap \llbracket A \rrbracket \\
\llbracket \{x \leftarrow P \rightarrow \overline{y_i}\}_z \rrbracket &\triangleq z(y_0) \dots z(y_n) \cdot \llbracket P\{z/x\} \rrbracket \quad (z \notin \text{fn}(P)) \\
\llbracket x \leftarrow M \leftarrow \overline{y_i}; Q \rrbracket &\triangleq (\nu x)(\llbracket M \rrbracket_x \cdot \overline{x}\langle a_0 \rangle \cdot ([a_0 \leftrightarrow y_0] \mid \dots \mid x\langle a_n \rangle \cdot ([a_n \leftrightarrow y_n] \mid \llbracket Q \rrbracket) \dots))
\end{aligned}$$

To encode the monadic type $\{\overline{x_j:A_j} \vdash z:A\}$, denoting the type of process P that is typed by $\overline{x_j:A_j} \vdash P :: z:A$, we require that the session in the image of the translation specifies a sequence of channel inputs with behaviours $\overline{A_j}$ that make up the linear context. After the contextual aspects of the type are encoded, the session will then offer the (encoded) behaviour of A . Thus, the encoding of the monadic type is $\llbracket A_0 \rrbracket \multimap \dots \multimap \llbracket A_n \rrbracket \multimap \llbracket A \rrbracket$, which we write as $\llbracket \overline{A_j} \rrbracket \multimap \llbracket A \rrbracket$. The encoding of monadic expressions adheres to this behaviour, first performing the necessary sequence of inputs and then proceeding inductively. Finally, the encoding of the elimination form for monadic expressions behaves dually, composing the encoding of the monadic expression with a sequence of outputs that instantiate the consumed names accordingly (via forwarding). The encoding of process P from Eq. 2 is thus:

$$\begin{aligned}
\llbracket P \rrbracket &= (\nu c)(\llbracket c\langle \{d \leftarrow Q\} \rangle.c(x).\mathbf{0} \rrbracket \mid \llbracket c(y).d \leftarrow y; d(n).c\langle n \rangle.\mathbf{0} \rrbracket) \\
&= (\nu c)(\overline{c}\langle w \rangle \cdot (!w(d).\llbracket Q \rrbracket \mid c(x).\mathbf{0})c(y).(\nu d)(\overline{y}\langle b \rangle \cdot [b \leftrightarrow d] \mid d(n).\overline{c}\langle m \rangle \cdot (\overline{n}\langle e \rangle \cdot [e \leftrightarrow m] \mid \mathbf{0})))
\end{aligned}$$

Properties of the Encoding. As in our previous development, we can show that our encoding for $\text{Sess}\pi\lambda^+$ is type sound and satisfies operational correspondence. The full development is omitted but can be found in [52].

We encode $\text{Sess}\pi\lambda^+$ into λ -terms, extending § 4.2 with:

$$\begin{aligned}
\llbracket \{\overline{x_i:A_i} \vdash z:A\} \rrbracket &\triangleq \llbracket \overline{A_i} \rrbracket \multimap \llbracket A \rrbracket \\
\llbracket x \leftarrow M \leftarrow \overline{y_i}; Q \rrbracket &\triangleq \llbracket Q \rrbracket \{ (\llbracket M \rrbracket \overline{y_i}) / x \} \quad \llbracket \{x \leftarrow P \leftarrow \overline{w_i}\} \rrbracket \triangleq \lambda w_0. \dots \lambda w_n. (\llbracket P \rrbracket)
\end{aligned}$$

The encoding translates the monadic type $\{\overline{x_i:A_i} \vdash z:A\}$ as a linear function $\llbracket \overline{A_i} \rrbracket \multimap \llbracket A \rrbracket$, which captures the fact that the underlying value must be provided with terms satisfying the requirements of the linear context. At the level of terms, the encoding for the monadic term constructor follows its type specification, generating a nesting of λ -abstractions that closes the term and proceed inductively. For the process encoding, we translate the monadic application construct analogously to the translation of a linear cut, but applying the appropriate variables to the translated monadic term (which is of function type). We remark the similarity between our encoding and that of the previous section, where monadic terms are translated to a sequence of inputs (here a nesting of λ -abstractions). Our encoding satisfies type soundness and operational correspondence, as usual. Further showcasing the applications of our development, we obtain a novel strong normalisation result for this higher-order session-calculus “for free”, through encoding to the λ -calculus.

Theorem 4.9 (Strong Normalisation). *Let $\Psi; \Gamma; \Delta \vdash P :: z:A$. There is no infinite reduction sequence starting from P .*

Theorem 4.10 (Inverse Encodings). *If $\Psi; \Gamma; \Delta \vdash P :: z:A$ then $\llbracket \llbracket P \rrbracket \rrbracket_z \approx_L \llbracket P \rrbracket$. Also, if $\Psi \vdash M : \tau$ then $\llbracket \llbracket M \rrbracket_z \rrbracket =_\beta \llbracket M \rrbracket$.*

Theorem 4.11. *Let $\vdash M : \tau, \vdash N : \tau, \vdash P :: z:A$ and $\vdash Q :: z:A$. $\langle M \rangle =_{\beta\eta} \langle N \rangle$ iff $\llbracket M \rrbracket_z \approx_L \llbracket N \rrbracket_z$ and $\llbracket P \rrbracket \approx_L \llbracket Q \rrbracket$ iff $\langle P \rangle =_{\beta\eta} \langle Q \rangle$.*

5 Related Work and Concluding Remarks

Process Encodings of Functions. Toninho et al. [49] study encodings of the simply-typed λ -calculus in a logically motivated session π -calculus, via encodings to the linear λ -calculus. Our work differs since they do not study polymorphism nor reverse encodings; and we provide deeper insights through applications of the encodings. Full abstraction or inverse properties are not studied.

Sangiorgi [43] uses a fully abstract compilation from the higher-order π -calculus ($\text{HO}\pi$) to the π -calculus to study full abstraction for Milner’s encodings of the λ -calculus. The work shows that Milner’s encoding of the lazy λ -calculus can be recovered by restricting the semantic domain of processes (the so-called *restrictive* approach) or by enriching the λ -calculus with suitable constants. This work was later refined in [45], which does not use $\text{HO}\pi$ and considers an operational equivalence on λ -terms called *open applicative bisimulation* which coincides with Lévy-Longo tree equality. The work [47] studies general conditions under which encodings of the λ -calculus in the π -calculus are fully abstract wrt Lévy-Longo and Böhm Trees, which are then applied to several encodings of (call-by-name) λ -calculus. The works above deal with *untyped calculi*, and so reverse encodings are unfeasible. In a broader sense, our approach takes the restrictive approach using linear logic-based session typing and the induced observational equivalence. We use a λ -calculus with booleans as observables and reason with a Morris-style equivalence instead of tree equalities. It would be an interesting future work to apply the conditions in [47] in our typed setting.

Wadler [54] shows a correspondence between a linear functional language with session types GV and a session-typed process calculus with polymorphism based on classical linear logic CP. Along the lines of this work, Lindley and Morris [26], in an exploration of inductive and coinductive session types through the addition of least and greatest fixed points to CP and GV, develop an encoding from a linear λ -calculus with session primitives (Concurrent μGV) to a pure linear λ -calculus (Functional μGV) via a CPS transformation. They also develop translations between μCP and Concurrent μGV , extending [25]. Mapping to the terminology used in our work [17], their encodings are shown to be operationally complete, but no results are shown for the operational soundness directions and neither full abstraction nor inverse properties are studied. In addition, their operational characterisations do not compose across encodings. For instance, while strong normalisation of Functional μGV implies the same property for Concurrent μGV through their operationally complete encoding, the encoding from μCP to μGV does not necessarily preserve this property.

Types for π -calculi delineate sequential behaviours by restricting composition and name usages, limiting the contexts in which processes can interact. Therefore typed equivalences offer a *coarser* semantics than untyped semantics. Berger et al. [5] study an encoding of System F in a polymorphic linear π -calculus, showing

it to be fully abstract based on game semantics techniques. Their typing system and proofs are more complex due to the fine-grained constraints from game semantics. Moreover, they do not study a reverse encoding. Orchard and Yoshida [33] develop embeddings to-and-from PCF with parallel effects and a session-typed π -calculus, but only develop operational correspondence and semantic soundness results, leaving the full abstraction problem open.

Polymorphism and Typed Behavioural Semantics. The work of [7] studies parametric session polymorphism for the intuitionistic setting, developing a behavioural equivalence that captures parametricity, which is used (denoted as \approx_L) in our paper. The work [39] introduces a typed bisimilarity for polymorphism in the π -calculus. Their bisimilarity is of an intensional flavour, whereas the one used in our work follows the extensional style of Reynolds [41]. Their typing discipline (originally from [53], which also develops type-preserving encodings of polymorphic λ -calculus into polymorphic π -calculus) differs significantly from the linear logic-based session typing of our work (e.g. theirs does not ensure deadlock-freedom). A key observation in their work is the coarser nature of typed equivalences with polymorphism (in analogy to those for IO-subtyping [38]) and their interaction with channel aliasing, suggesting a use of typed semantics and encodings of the π -calculus for fine-grained analyses of program behaviour.

F-Algebras and Linear-F. The use of initial and final (co)algebras to give a semantics to inductive and coinductive types dates back to Mendler [28], with their strong definability in System F appearing in [1, 19]. The definability of inductive and coinductive types using parametricity also appears in [40] in the context of a logic for parametric polymorphism and later in [6] in a linear variant of such a logic. The work of [55] studies parametricity for the polymorphic linear λ -calculus of this work, developing encodings of a few inductive types but not the initial (or final) algebraic encodings in their full generality. Inductive and coinductive session types in a logical process setting appear in [26, 51]. Both works consider a calculus with built-in recursion – the former in an intuitionistic setting where a process that offers a (co)inductive protocol is composed with another that consumes the (co)inductive protocol and the latter in a classical framework where composed recursive session types are dual each other.

Conclusion and Future Work. This work answers the question of what kind of type discipline of the π -calculus can exactly capture and is captured by λ -calculus behaviours. Our answer is given by showing the first mutually inverse and fully abstract encodings between two calculi with polymorphism, one being the Poly π session calculus based on intuitionistic linear logic, and the other (a linear) System F. This further demonstrates that the linear logic-based articulation of name-passing interactions originally proposed by [8] (and studied extensively thereafter e.g. [7, 9, 25, 36, 50, 51, 54]) provides a clear and applicable tool for message-passing concurrency. By exploiting the proof theoretic equivalences between natural deduction and sequent calculus we develop mutually inverse and fully abstract encodings, which naturally extend to more intricate settings such as process passing (in the sense of $\text{HO}\pi$). Our encodings also enable us to

derive properties of the π -calculi “for free”. Specifically, we show how to obtain adequate representations of least and greatest fixed points in $\text{Poly}\pi$ through the encoding of initial and final (co)algebras in the λ -calculus. We also straightforwardly derive a strong normalisation result for the higher-order session calculus, which otherwise involves non-trivial proof techniques [5, 7, 12, 13, 36]. Future work includes extensions to the classical linear logic-based framework, including multiparty session types [10, 11]. Encodings of session π -calculi to the λ -calculus have been used to implement session primitives in functional languages such as Haskell (see a recent survey [32]), OCaml [24, 34, 35] and Scala [48]. Following this line of work, we plan to develop encoding-based implementations of this work as embedded DSLs. This would potentially enable an exploration of algebraic constructs beyond initial and final co-algebras in a session programming setting. In particular, we wish to further study the meaning of functors, natural transformations and related constructions in a session-typed setting, both from a more fundamental viewpoint but also in terms of programming patterns.

Acknowledgements. The authors thank Viviana Bono, Dominic Orchard and the reviewers for their comments, suggestions and pointers to related works. This work is partially supported by EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1 and NOVA LINCS (UID/CEC/04516/2013).

References

1. Bainbridge, E.S., Freyd, P.J., Scedrov, A., Scott, P.J.: Functorial polymorphism. *Theor. Comput. Sci.* **70**(1), 35–64 (1990)
2. Balzer, S., Pfenning, F.: Manifest sharing with session types. In: ICFP (2017)
3. Barber, A.: Dual intuitionistic linear logic. Technical report ECS-LFCS-96-347. School of Informatics, University of Edinburgh (1996)
4. Benton, P.N.: A mixed linear and non-linear logic: proofs, terms and models. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933, pp. 121–135. Springer, Heidelberg (1995). <https://doi.org/10.1007/BFb0022251>
5. Berger, M., Honda, K., Yoshida, N.: Genericity and the π -calculus. *Acta Inf.* **42**(2–3), 83–141 (2005)
6. Birkedal, L., Møgelberg, R.E., Petersen, R.L.: Linear abadi and plotkin logic. *Log. Methods Comput. Sci.* **2**(5), 1–48 (2006)
7. Caires, L., Pérez, J.A., Pfenning, F., Toninho, B.: Behavioral polymorphism and parametricity in session-based communication. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 330–349. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_19
8. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 222–236. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_16
9. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. *Math. Struct. Comput. Sci.* **26**(3), 367–423 (2016)
10. Carbone, M., Lindley, S., Montesi, F., Schuermann, C., Wadler, P.: Coherence generalises duality: a logical explanation of multiparty session types. In: CONCUR 2016, vol. 59, pp. 33:1–33:15. Sch. Dag. (2016)

11. Carbone, M., Montesi, F., Schurmann, C., Yoshida, N.: Multiparty session types as coherence proofs. In: CONCUR 2015, vol. 42, pp. 412–426. Sch. Dag. (2015)
12. Demangeon, R., Hirschkoﬀ, D., Sangiorgi, D.: Mobile processes and termination. In: Palsberg, J. (ed.) *Semantics and Algebraic Specification*. LNCS, vol. 5700, pp. 250–273. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04164-8_13
13. Demangeon, R., Hirschkoﬀ, D., Sangiorgi, D.: Termination in higher-order concurrent calculi. *J. Log. Algebr. Program.* **79**(7), 550–577 (2010)
14. Gentzen, G.: Untersuchungen über das logische schließen. *Math. Z.* **39**, 176–210 (1935)
15. Girard, J.: Linear logic. *Theor. Comput. Sci.* **50**, 1–102 (1987)
16. Girard, J., Lafont, Y., Taylor, P.: *Proofs and Types*. CUP, Cambridge (1989)
17. Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.* **208**(9), 1031–1053 (2010)
18. Gorla, D., Nestmann, U.: Full abstraction for expressiveness: history, myths and facts. *Math. Struct. Comput. Sci.* **26**(4), 639–654 (2016)
19. Hasegawa, R.: Categorical data types in parametric polymorphism. *Math. Struct. Comput. Sci.* **4**(1), 71–109 (1994)
20. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_35
21. Honda, K.: Session types and distributed computing. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012. LNCS, vol. 7392, pp. 23–23. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31585-5_4
22. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0053567>
23. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL 2008, pp. 273–284 (2008)
24. Imai, K., Yoshida, N., Yuen, S.: *Session-ocaml*: a session-based library with polarities and lenses. In: Jacquet, J.-M., Massink, M. (eds.) COORDINATION 2017. LNCS, vol. 10319, pp. 99–118. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59746-1_6
25. Lindley, S., Morris, J.G.: A semantics for propositions as sessions. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 560–584. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_23
26. Lindley, S., Morris, J.G.: Talking bananas: structural recursion for session types. In: ICFP 2016, pp. 434–447 (2016)
27. Maraist, J., Odersky, M., Turner, D.N., Wadler, P.: Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *T. C. S.* **228**(1–2), 175–210 (1999)
28. Mendler, N.P.: Recursive types and type constraints in second-order lambda calculus. In: LICS 1987, pp. 30–36 (1987)
29. Milner, R.: Functions as processes. In: Paterson, M.S. (ed.) ICALP 1990. LNCS, vol. 443, pp. 167–180. Springer, Heidelberg (1990). <https://doi.org/10.1007/BFb0032030>
30. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes I and II. *Inf. Comput.* **100**(1), 1–77 (1992)
31. Ohta, Y., Hasegawa, M.: A terminating and confluent linear lambda calculus. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 166–180. Springer, Heidelberg (2006). https://doi.org/10.1007/11805618_13

32. Orchard, D., Yoshida, N.: Session types with linearity in Haskell. In: Gay, S., Ravara, A. (eds.) *Behavioural Types: From Theory to Tools*. River Publishers, Gistrup (2017)
33. Orchard, D.A., Yoshida, N.: Effects as sessions, sessions as effects. In: *POPL 2016*, pp. 568–581 (2016)
34. Padovani, L.: A Simple Library Implementation of Binary Sessions. *JFP* **27** (2016)
35. Padovani, L.: Context-free session type inference. In: Yang, H. (ed.) *ESOP 2017*. LNCS, vol. 10201, pp. 804–830. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_30
36. Pérez, J.A., Caires, L., Pfenning, F., Toninho, B.: Linear logical relations for session-based concurrency. In: Seidl, H. (ed.) *ESOP 2012*. LNCS, vol. 7211, pp. 539–558. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_27
37. Pfenning, F., Griffith, D.: Polarized substructural session types. In: Pitts, A. (ed.) *FoSSaCS 2015*. LNCS, vol. 9034, pp. 3–22. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46678-0_1
38. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. *Math. Struct. Comput. Sci.* **6**(5), 409–453 (1996)
39. Pierce, B.C., Sangiorgi, D.: Behavioral equivalence in the polymorphic pi-calculus. *J. ACM* **47**(3), 531–584 (2000)
40. Plotkin, G., Abadi, M.: A logic for parametric polymorphism. In: Bezem, M., Groote, J.F. (eds.) *TLCA 1993*. LNCS, vol. 664, pp. 361–375. Springer, Heidelberg (1993). <https://doi.org/10.1007/BFb0037118>
41. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: *IFIP Congress*, pp. 513–523 (1983)
42. Reynolds, J.C., Plotkin, G.D.: On functors expressible in the polymorphic typed lambda calculus. *Inf. Comput.* **105**(1), 1–29 (1993)
43. Sangiorgi, D.: An investigation into functions as processes. In: Brookes, S., Main, M., Melton, A., Mislove, M., Schmidt, D. (eds.) *MFPS 1993*. LNCS, vol. 802, pp. 143–159. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58027-1_7
44. Sangiorgi, D.: Π -calculus, internal mobility, and agent-passing calculi. *Theor. Comput. Sci.* **167**(1&2), 235–274 (1996)
45. Sangiorgi, D.: Lazy functions and mobile processes. In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pp. 691–720 (2000)
46. Sangiorgi, D., Walker, D.: *The π -Calculus: A Theory of Mobile Processes*. Cambridge University Press, Cambridge (2001)
47. Sangiorgi, D., Xu, X.: Trees from functions as processes. In: Baldan, P., Gorla, D. (eds.) *CONCUR 2014*. LNCS, vol. 8704, pp. 78–92. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44584-6_7
48. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: *ECOOP 2017* (2017)
49. Toninho, B., Caires, L., Pfenning, F.: Functions as session-typed processes. In: Birkedal, L. (ed.) *FoSSaCS 2012*. LNCS, vol. 7213, pp. 346–360. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28729-9_23
50. Toninho, B., Caires, L., Pfenning, F.: Higher-order processes, functions, and sessions: a monadic integration. In: Felleisen, M., Gardner, P. (eds.) *ESOP 2013*. LNCS, vol. 7792, pp. 350–369. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_20

51. Toninho, B., Caires, L., Pfenning, F.: Corecursion and non-divergence in session-typed processes. In: Maffei, M., Tuosto, E. (eds.) TGC 2014. LNCS, vol. 8902, pp. 159–175. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45917-1_11
52. Toninho, B., Yoshida, N.: On polymorphic sessions and functions: a tale of two (fully abstract) encodings (long version). CoRR abs/1711.00878 (2017)
53. Turner, D.: The polymorphic pi-calculus: Theory and implementation. Technical report ECS-LFCS-96-345. School of Informatics, University of Edinburgh (1996)
54. Wadler, P.: Propositions as sessions. *J. Funct. Program.* **24**(2–3), 384–418 (2014)
55. Zhao, J., Zhang, Q., Zdancewic, S.: Relational parametricity for a polymorphic linear lambda calculus. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 344–359. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17164-2_24

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Concurrent Kleene Algebra: Free Model and Completeness

Tobias Kappé^(✉), Paul Brunet, Alexandra Silva, and Fabio Zanasi

University College London, London, UK
tkappe@cs.ucl.ac.uk

Abstract. Concurrent Kleene Algebra (CKA) was introduced by Hoare, Moeller, Struth and Wehrman in 2009 as a framework to reason about concurrent programs. We prove that the axioms for CKA with bounded parallelism are complete for the semantics proposed in the original paper; consequently, these semantics are the free model for this fragment. This result settles a conjecture of Hoare and collaborators. Moreover, the technique developed to this end allows us to establish a Kleene Theorem for CKA, extending an earlier Kleene Theorem for a fragment of CKA.

1 Introduction

Concurrent Kleene Algebra (CKA) [8] is a mathematical formalism which extends Kleene Algebra (KA) with a parallel composition operator, in order to express concurrent program behaviour.¹ In spite of such a seemingly simple addition, extending the existing KA toolkit (notably, completeness) to the setting of CKA turned out to be a challenging task. A lot of research happened since the original paper, both foundational [13, 20] and on how CKA could be used to reason about important verification tasks in concurrent systems [9, 11]. However, and despite several conjectures [9, 13], the question of the characterisation of the free CKA and the completeness of the axioms remained open, making it impractical to use CKA in verification tasks. This paper settles these two open questions. We answer positively the conjecture that the free model of CKA is formed by series parallel pomset languages, downward-closed under Gischer’s subsumption order [6]—a generalisation of regular languages to sets of partially ordered words. To this end, we prove that the original axioms proposed in [8] are indeed complete.

Our proof of completeness is based on extending an existing completeness result that establishes series-parallel rational pomset languages as the free Bi-Kleene Algebra (BKA) [20]. The extension to the existing result for BKA provides a clear understanding of the difficulties introduced by the presence of the exchange axiom and shows how to separate concerns between CKA and BKA, a technique which is also useful elsewhere. For one, our construction also provides

¹ In its original formulation, CKA also features an operator (*parallel star*) for unbounded parallelism: in harmony with several recent works [13, 14], we study the variant of CKA without parallel star, sometimes called “weak” CKA.

an extension of (half of) Kleene’s theorem for BKA [14] to CKA, establishing pomset automata as an operational model for CKA and opening the door to decidability procedures similar to those previously studied for KA. Furthermore, it reduces deciding the equational theory of CKA to deciding the equational theory of BKA.

BKA is defined as CKA with the only (but significant) omission of the *exchange law*, $(e \parallel f) \cdot (g \parallel h) \leq_{\text{CKA}} (e \cdot g) \parallel (f \cdot h)$. The exchange law is the core element of CKA as it softens true concurrency: it states that when two sequentially composed programs (i.e., $e \cdot g$ and $f \cdot h$) are composed in parallel, they can be implemented by running their heads in parallel, followed by running their tails in parallel (i.e., $e \parallel f$, then $g \parallel h$). The exchange law allows the implementer of a CKA expression to interleave threads at will, without violating the specification.

To illustrate the use of the exchange law, consider a protocol with three actions: query a channel c , collect an answer from the same channel, and print an unrelated message m on screen. The specification for this protocol requires the query to happen before reception of the message, but the printing action being independent, it may be executed concurrently. We will write this specification as $(q(c) \cdot r(c)) \parallel p(m)$, with the operator \cdot denoting sequential composition. However, if one wants to implement this protocol in a sequential programming language, a total ordering of these events has to be introduced. Suppose we choose to implement this protocol by printing m while we wait to receive an answer. This implementation can be written $q(c) \cdot p(m) \cdot r(c)$. Using the laws of CKA, we can prove that $q(c) \cdot p(m) \cdot r(c) \leq_{\text{CKA}} (q(c) \cdot r(c)) \parallel p(m)$, which we interpret as the fact that this implementation respects the specification. Intuitively, this means that the specification lists the necessary dependencies, but the implementation can introduce more.

Having a complete axiomatisation of CKA has two main benefits. First, it allows one to get certificates of correctness. Indeed, if one wants to use CKA for program verification, the decision procedure presented in [3] may be used to test program equivalence. If the test gives a negative answer, this algorithm provides a counter-example. However if the answer is positive, no meaningful witness is produced. With the completeness result presented here, that is constructive in nature, one could generate an axiomatic proof of equivalence in these cases. Second, it gives one a simple way of checking when the aforementioned procedure applies. By construction, we know that two terms are semantically equivalent whenever they are equal in every concurrent Kleene algebra, that is any model of the axioms of CKA. This means that if we consider a specific semantic domain, one simply needs to check that the axioms of CKA hold in there to know that the decision procedure of [3] is sound in this model.

While this paper was in writing, a manuscript with the same result appeared [19]. Among other things, the proof presented here is different in that it explicitly shows how to syntactically construct terms that express certain pomset languages, as opposed to showing that such terms must exist by reasoning on a semantic level. We refer to Sect. 5 for a more extensive comparison.

The remainder of this paper is organised as follows. In Sect. 2, we give an informal overview of the completeness proof. In Sect. 3, we introduce the necessary concepts, notation and lemmas. In Sect. 4, we work out the proof. We discuss the result in a broader perspective and outline further work in Sect. 5.

2 Overview of the Completeness Proof

We start with an overview of the steps necessary to arrive at the main result. As mentioned, our strategy in tackling CKA-completeness is to build on the existing BKA-completeness result. Following an observation by Laurence and Struth, we identify *downward-closure* (under Gischer’s subsumption order [6]) as the feature that distinguishes the pomsets giving semantics to BKA-expressions from those associated with CKA-expressions. In a slogan,

$$\text{CKA-semantics} = \text{BKA-semantics} + \text{downward-closure}.$$

This situation is depicted in the upper part of the commuting diagram in Fig. 1. Intuitively, downward-closure can be thought of as the semantic outcome of adding the exchange axiom, which distinguishes CKA from BKA. Thus, if a and b are events that can happen in parallel according to the BKA-semantics of a term, then a and b may also be ordered in the CKA-semantics of that same term.

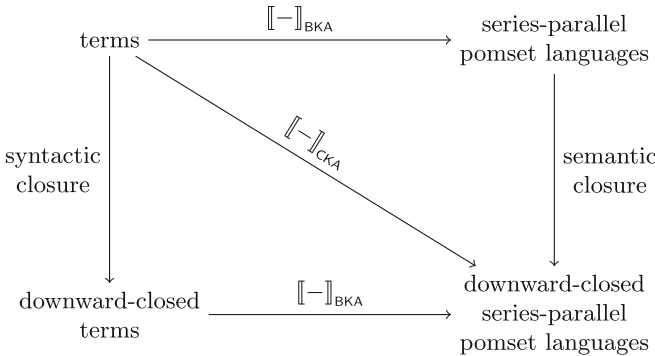


Fig. 1. The connection between BKA and CKA semantics mediated by closure.

The core of our CKA-completeness proof will be to construct a syntactic counterpart to the semantic closure. Concretely, we shall build a function that maps a CKA term e to an equivalent term $e\downarrow$, called the (syntactic) *closure* of e . The lower part of the commuting diagram in Fig. 1 shows the property that $e\downarrow$ must satisfy in order to deserve the name of closure: its BKA semantics has to be the same as the CKA semantics of e .

Example 2.1. Consider $e = a \parallel b$, whose CKA-semantics prescribe that a and b are events that may happen in parallel. One closure of this term would be $e\downarrow = a \parallel b + a \cdot b + b \cdot a$, whose BKA-semantics stipulate that either a and b execute purely in parallel, or a precedes b , or b precedes a —thus matching the optional parallelism of a and b . For a more non-trivial example, take $e = a^* \parallel b^*$, which represents that finitely many repetitions of a and b occur, possibly in parallel. A closure of this term would be $e\downarrow = (a^* \parallel b^*)^*$: finitely many repetitions of a and b occur truly in parallel, which is repeated indefinitely.

In order to find $e\downarrow$ systematically, we are going to construct it in stages, through a completely syntactic procedure where each transformation has to be valid according to the axioms. There are three main stages.

- (i) We note that, not unexpectedly, the hardest case for computing the closure of a term is when e is a parallel composition, i.e., when $e = e_0 \parallel e_1$ for some CKA terms e_0 and e_1 . For the other operators, the closure of the result can be obtained by applying the same operator to the closures of its arguments. For instance, $(e + f)\downarrow = e\downarrow + f\downarrow$. This means that we can focus on calculating the closure for the particular case of parallel composition.
- (ii) We construct a *preclosure* of such terms e , whose BKA semantics contains all but possibly the sequentially composed pomsets of the CKA semantics of e . Since every sequentially composed pomset decomposes (uniquely) into non-sequential pomsets, we can use the preclosure as a basis for induction.
- (iii) We extend this preclosure of e to a proper closure, by leveraging the fixpoint axioms of KA to solve a system of linear inequations. This system encodes “stringing together” non-sequential pomsets to build all pomsets in e .

As a straightforward consequence of the closure construction, we obtain a completeness theorem for CKA, which establishes the set of closed series-rational pomset languages as the free CKA.

3 Preliminaries

We fix a finite set of symbols Σ , the *alphabet*. We use the symbols a , b and c to denote elements of Σ . The two-element set $\{0, 1\}$ is denoted by 2 . Given a set S , the set of subsets (*powerset*) of S is denoted by 2^S .

In the interest of readability, the proofs for technical lemmas in this section can be found in the full version [15].

3.1 Pomsets

A trace of a sequential program can be modelled as a word, where each letter represents an atomic event, and the order of the letters in the word represents the order in which the events took place. Analogously, a trace of a concurrent program can be thought of as word where letters are partially ordered, i.e., there need not be a causal link between events. In literature, such a partially ordered

word is commonly called a *partial word* [7], or *partially ordered multiset* (*pomset*, for short) [6]; we use the latter term.

A formal definition of pomsets requires some work, because the partial order should order *occurrences* of events rather than the events themselves. For this reason, we first define a labelled poset.

Definition 3.1. A labelled poset is a tuple $\langle S, \leq, \lambda \rangle$, where $\langle S, \leq \rangle$ is a partially ordered set (i.e., S is a set and \leq is a partial order on S), in which S is called the carrier and \leq is the order; $\lambda : S \rightarrow \Sigma$ is a function called the labelling.

We denote labelled posets with lower-case bold symbols \mathbf{u}, \mathbf{v} , et cetera. Given a labelled poset \mathbf{u} , we write $S_{\mathbf{u}}$ for its carrier, $\leq_{\mathbf{u}}$ for its order and $\lambda_{\mathbf{u}}$ for its labelling. We write $\mathbf{1}$ for the empty labelled poset. We say that two labelled posets are *disjoint* if their carriers are disjoint.

Disjoint labelled posets can be composed parallelly and sequentially; parallel composition simply juxtaposes the events, while sequential composition imposes an ordering between occurrences of events originating from the left operand and those originating from the right operand.

Definition 3.2. Let \mathbf{u} and \mathbf{v} be disjoint. We write $\mathbf{u} \parallel \mathbf{v}$ for the parallel composition of \mathbf{u} and \mathbf{v} , which is the labelled poset with the carrier $S_{\mathbf{u} \parallel \mathbf{v}} = S_{\mathbf{u}} \cup S_{\mathbf{v}}$, the order $\leq_{\mathbf{u} \parallel \mathbf{v}} = \leq_{\mathbf{u}} \cup \leq_{\mathbf{v}}$ and the labeling $\lambda_{\mathbf{u} \parallel \mathbf{v}}$ defined by

$$\lambda_{\mathbf{u} \parallel \mathbf{v}}(x) = \begin{cases} \lambda_{\mathbf{u}}(x) & x \in S_{\mathbf{u}}; \\ \lambda_{\mathbf{v}}(x) & x \in S_{\mathbf{v}}. \end{cases}$$

Similarly, we write $\mathbf{u} \cdot \mathbf{v}$ for the sequential composition of \mathbf{u} and \mathbf{v} , that is, labelled poset with the carrier $S_{\mathbf{u} \cdot \mathbf{v}}$ and the partial order

$$\leq_{\mathbf{u} \cdot \mathbf{v}} = \leq_{\mathbf{u}} \cup \leq_{\mathbf{v}} \cup (S_{\mathbf{u}} \times S_{\mathbf{v}}),$$

as well as the labelling $\lambda_{\mathbf{u} \cdot \mathbf{v}} = \lambda_{\mathbf{u} \parallel \mathbf{v}}$.

Note that $\mathbf{1}$ is neutral for sequential and parallel composition, in the sense that we have $\mathbf{1} \parallel \mathbf{u} = \mathbf{1} \cdot \mathbf{u} = \mathbf{u} = \mathbf{u} \cdot \mathbf{1} = \mathbf{u} \parallel \mathbf{1}$.

There is a natural ordering between labelled posets with regard to concurrency.

Definition 3.3. Let \mathbf{u}, \mathbf{v} be labelled posets. A subsumption from \mathbf{u} to \mathbf{v} is a bijection $h : S_{\mathbf{u}} \rightarrow S_{\mathbf{v}}$ that preserves order and labels, i.e., $u \leq_{\mathbf{u}} u'$ implies that $h(u) \leq_{\mathbf{v}} h(u')$, and $\lambda_{\mathbf{v}} \circ h = \lambda_{\mathbf{u}}$. We simplify and write $h : \mathbf{u} \rightarrow \mathbf{v}$ for a subsumption from \mathbf{u} to \mathbf{v} . If such a subsumption exists, we write $\mathbf{v} \sqsubseteq \mathbf{u}$. Furthermore, h is an isomorphism if both h and its inverse h^{-1} are subsumptions. If there exists an isomorphism from \mathbf{u} to \mathbf{v} we write $\mathbf{u} \cong \mathbf{v}$.

Intuitively, if $\mathbf{u} \sqsubseteq \mathbf{v}$, then \mathbf{u} and \mathbf{v} both order the same set of (occurrences of) events, but \mathbf{u} has more causal links, or “is more sequential” than \mathbf{v} . One easily sees that \sqsubseteq is a preorder on labelled posets of finite carrier.

Since the actual contents of the carrier of a labelled poset do not matter, we can abstract from them using isomorphism. This gives rise to pomsets.

Definition 3.4. A pomset is an isomorphism class of labelled posets, i.e., the class $[\mathbf{v}] \triangleq \{\mathbf{u} : \mathbf{u} \cong \mathbf{v}\}$ for some labelled poset \mathbf{v} . Composition lifts to pomsets: we write $[\mathbf{u}] \parallel [\mathbf{v}]$ for $[\mathbf{u} \parallel \mathbf{v}]$ and $[\mathbf{u}] \cdot [\mathbf{v}]$ for $[\mathbf{u} \cdot \mathbf{v}]$. Similarly, subsumption also lifts to pomsets: we write $[\mathbf{u}] \sqsubseteq [\mathbf{v}]$, precisely when $\mathbf{u} \sqsubseteq \mathbf{v}$.

We denote pomsets with upper-case symbols U, V , et cetera. The *empty pomset*, i.e., $[\mathbf{1}] = \{\mathbf{1}\}$, is denoted by 1 ; this pomset is neutral for sequential and parallel composition. To ensure that $[\mathbf{v}]$ is a set, we limit the discussion to labelled posets whose carrier is a subset of some set \mathbb{S} . The labelled posets in this paper have finite carrier; it thus suffices to choose $\mathbb{S} = \mathbb{N}$ to represent all pomsets with finite (or even countably infinite) carrier.

Composition of pomsets is well-defined: if \mathbf{u} and \mathbf{v} are not disjoint, we can find \mathbf{u}', \mathbf{v}' disjoint from \mathbf{u}, \mathbf{v} respectively such that $\mathbf{u} \cong \mathbf{u}'$ and $\mathbf{v} \cong \mathbf{v}'$. The choice of representative does not matter, for if $\mathbf{u} \cong \mathbf{u}'$ and $\mathbf{v} \cong \mathbf{v}'$, then $\mathbf{u} \cdot \mathbf{v} \cong \mathbf{u}' \cdot \mathbf{v}'$. Subsumption of pomsets is also well-defined: if $\mathbf{u}' \cong \mathbf{u} \sqsubseteq \mathbf{v} \cong \mathbf{v}'$, then $\mathbf{u}' \sqsubseteq \mathbf{v}'$. One easily sees that \sqsubseteq is a partial order on finite pomsets, and that sequential and parallel composition are monotone with respect to \sqsubseteq , i.e., if $U \sqsubseteq W$ and $V \sqsubseteq X$, then $U \cdot V \sqsubseteq W \cdot X$ and $U \parallel V \sqsubseteq W \parallel X$. Lastly, we note that both types of composition are associative, both on the level of pomsets and labelled posets; we therefore omit parentheses when no ambiguity is likely.

Series-Parallel Pomsets. If $a \in \Sigma$, we can construct a labelled poset with a single element labelled by a ; indeed, since any labelled poset thus constructed is isomorphic, we also use a to denote this isomorphism class; such a pomset is called a *primitive pomset*. A pomset built from primitive pomsets and sequential and parallel composition is called *series-parallel*; more formally:

Definition 3.5. The set of series-parallel pomsets, denoted $\text{SP}(\Sigma)$, is the smallest set such that $1 \in \text{SP}(\Sigma)$ as well as $a \in \text{SP}(\Sigma)$ for every $a \in \Sigma$, and is closed under parallel and sequential composition.

We elide the sequential composition operator when we explicitly construct a pomset from primitive pomsets, i.e., we write ab instead of $a \cdot b$ for the pomset obtained by sequentially composing the (primitive) pomsets a and b . In this notation, sequential composition takes precedence over parallel composition.

All pomsets encountered in this paper are series-parallel. A useful feature of series-parallel pomsets is that we can deconstruct them in a standard fashion [6].

Lemma 3.1. Let $U \in \text{SP}(\Sigma)$. Then exactly one of the following is true: either (i) $U = 1$, or (ii) $U = a$ for some $a \in \Sigma$, or (iii) $U = U_0 \cdot U_1$ for $U_0, U_1 \in \text{SP}(\Sigma) \setminus \{1\}$, or (iv) $U = U_0 \parallel U_1$ for $U_0, U_1 \in \text{SP}(\Sigma) \setminus \{1\}$.

In the sequel, it will be useful to refer to pomsets that are *not* of the third kind above, i.e., cannot be written as $U_0 \cdot U_1$ for $U_0, U_1 \in \text{SP}(\Sigma) \setminus \{1\}$, as *non-sequential* pomsets. Lemma 3.1 gives a normal form for series-parallel pomsets, as follows.

Corollary 3.1. A pomset $U \in \text{SP}(\Sigma)$ can be uniquely decomposed as $U = U_0 \cdot U_1 \cdots U_{n-1}$, where for all $0 \leq i < n$, U_i is series parallel and non-sequential.

Factorisation. We now go over some lemmas on pomsets that will allow us to factorise pomsets later on. First of all, one easily shows that subsumption is irrelevant on empty and primitive pomsets, as witnessed by the following lemma.

Lemma 3.2. *Let U and V be pomsets such that $U \sqsubseteq V$ or $V \sqsubseteq U$. If U is empty or primitive, then $U = V$.*

We can also consider how pomset composition and subsumption relate. It is not hard to see that if a pomset is subsumed by a sequentially composed pomset, then this sequential composition also appears in the subsumed pomset. A similar statement holds for pomsets that subsume a parallel composition.

Lemma 3.3 (Factorisation). *Let U , V_0 , and V_1 be pomsets such that U is subsumed by $V_0 \cdot V_1$. Then there exist pomsets U_0 and U_1 such that:*

$$U = U_0 \cdot U_1, U_0 \sqsubseteq V_0, \text{ and } U_1 \sqsubseteq V_1.$$

Also, if U_0 , U_1 and V are pomsets such that $U_0 \parallel U_1 \sqsubseteq V$, then there exist pomsets V_0 and V_1 such that:

$$V = V_0 \parallel V_1, U_0 \sqsubseteq V_0, \text{ and } U_1 \sqsubseteq V_1.$$

The next lemma can be thought of as a generalisation of Levi's lemma [21], a well-known statement about words, to pomsets. It says that if a sequential composition is subsumed by another (possibly longer) sequential composition, then there must be a pomset “in the middle”, describing the overlap between the two; this pomset gives rise to a factorisation.

Lemma 3.4. *Let U and V be pomsets, and let W_0, W_1, \dots, W_{n-1} with $n > 0$ be non-empty pomsets such that $U \cdot V \sqsubseteq W_0 \cdot W_1 \cdots W_{n-1}$. There exists an $m < n$ and pomsets Y, Z such that:*

$$Y \cdot Z \sqsubseteq W_m, U \sqsubseteq W_0 \cdot W_1 \cdots W_{m-1} \cdot Y, \text{ and } V \sqsubseteq Z \cdot W_{m+1} \cdot W_{m+2} \cdots W_n.$$

Moreover, if U and V are series-parallel, then so are Y and Z .

Levi's lemma also has an analogue for parallel composition.

Lemma 3.5. *Let U, V, W, X be pomsets such that $U \parallel V = W \parallel X$. There exist pomsets Y_0, Y_1, Z_0, Z_1 such that*

$$U = Y_0 \parallel Y_1, V = Z_0 \parallel Z_1, W = Y_0 \parallel Z_0, \text{ and } X = Y_1 \parallel Z_1.$$

The final lemma is useful when we have a sequentially composed pomset subsumed by a parallelly composed pomset. It tells us that we can factor the involved pomsets to find subsumptions between smaller pomsets. This lemma first appeared in [6], where it is called the interpolation lemma.

Lemma 3.6 (Interpolation). *Let U, V, W, X be pomsets such that $U \cdot V$ is subsumed by $W \parallel X$. Then there exist pomsets W_0, W_1, X_0, X_1 such that*

$$W_0 \cdot W_1 \sqsubseteq W, X_0 \cdot X_1 \sqsubseteq X, U \sqsubseteq W_0 \parallel X_0, \text{ and } V \sqsubseteq W_1 \parallel X_1.$$

Moreover, if W and X are series-parallel, then so are W_0, W_1, X_0 and X_1 .

On a semi-formal level, the interpolation lemma can be understood as follows. If $U \cdot V \sqsubseteq W \parallel X$, then the events in W are partitioned between those that end up in U , and those that end up in V ; these give rise to the “sub-pomsets” W_0 and W_1 of W , respectively. Similarly, X partitions into “sub-pomsets” X_0 and X_1 . We refer to Fig. 2 for a graphical depiction of this situation.

Now, if y precedes z in $W_0 \parallel X_0$, then y must precede z in $W \parallel X$, and therefore also in $U \cdot V$. Since y and z are both events in U , it then follows that y precedes z in U , establishing that $U \sqsubseteq W_0 \parallel X_0$. Furthermore, if y precedes z in W , then we can exclude the case where y is in W_1 and z in W_0 , for then z precedes y in $U \cdot V$, contradicting that y precedes z in $U \cdot V$. Accordingly, either y and z both belong to W_0 or W_1 , or y is in W_0 while z is in W_1 ; in all of these cases, y must precede z in $W_0 \cdot W_1$. The other subsumptions hold analogously.

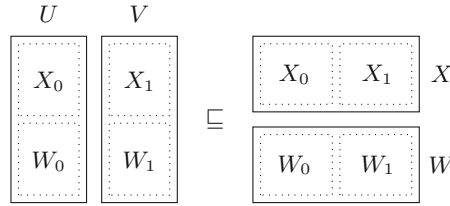


Fig. 2. Splitting pomsets in the interpolation lemma

Pomset Languages. The semantics of BKA and CKA are given in terms of sets of series-parallel pomsets.

Definition 3.6. *A subset of $\text{SP}(\Sigma)$ is referred to as a pomset language.*

As a convention, we denote pomset languages by the symbols \mathcal{U}, \mathcal{V} , et cetera. Sequential and parallel composition of pomsets extends to pomset languages in a pointwise manner, i.e.,

$$\mathcal{U} \cdot \mathcal{V} \triangleq \{U \cdot V : U \in \mathcal{U}, V \in \mathcal{V}\}$$

and similarly for parallel composition. Like languages of words, pomset languages have a Kleene star operator, which is similarly defined, i.e., $\mathcal{U}^* \triangleq \bigcup_{n \in \mathbb{N}} \mathcal{U}^n$, where the n^{th} power of \mathcal{U} is inductively defined as $\mathcal{U}^0 \triangleq \{1\}$ and $\mathcal{U}^{n+1} \triangleq \mathcal{U}^n \cdot \mathcal{U}$.

A pomset language \mathcal{U} is *closed under subsumption* (or simply *closed*) if whenever $U \in \mathcal{U}$ with $U' \sqsubseteq U$ and $U' \in \text{SP}(\Sigma)$, it holds that $U' \in \mathcal{U}$. The *closure under subsumption* (or simply *closure*) of a pomset language \mathcal{U} , denoted $\mathcal{U} \downarrow$, is defined as the smallest pomset language that contains \mathcal{U} and is closed, i.e.,

$$\mathcal{U} \downarrow \triangleq \{U' \in \text{SP}(\Sigma) : \exists U \in \mathcal{U}. U' \sqsubseteq U\}$$

Closure relates to union, sequential composition and iteration as follows.

Lemma 3.7. *Let \mathcal{U}, \mathcal{V} be pomset languages; then:*

$$(\mathcal{U} \cup \mathcal{V}) \downarrow = \mathcal{U} \downarrow \cup \mathcal{V} \downarrow, (\mathcal{U} \cdot \mathcal{V}) \downarrow = \mathcal{U} \downarrow \cdot \mathcal{V} \downarrow, \text{ and } \mathcal{U}^* \downarrow = \mathcal{U} \downarrow^*.$$

Proof. The first claim holds for infinite unions, too, and follows immediately from the definition of closure.

For the second claim, suppose that $U \in \mathcal{U}$ and $V \in \mathcal{V}$, and that $W \sqsubseteq U \cdot V$. By Lemma 3.3, we find pomsets W_0 and W_1 such that $W = W_0 \cdot W_1$, with $W_0 \sqsubseteq U$ and $W_1 \sqsubseteq V$. It then holds that $W_0 \in \mathcal{U} \downarrow$ and $W_1 \in \mathcal{V} \downarrow$, meaning that $W = W_0 \cdot W_1 \in \mathcal{U} \downarrow \cdot \mathcal{V} \downarrow$. This shows that $(\mathcal{U} \cdot \mathcal{V}) \downarrow \sqsubseteq \mathcal{U} \downarrow \cdot \mathcal{V} \downarrow$. Proving the reverse inclusion is a simple matter of unfolding the definitions.

For the third claim, we can calculate directly using the first and second parts of this lemma:

$$\mathcal{U}^* \downarrow = \left(\bigcup_{n \in \mathbb{N}} \underbrace{\mathcal{U} \cdot \mathcal{U} \cdots \mathcal{U}}_{n \text{ times}} \right) \downarrow = \bigcup_{n \in \mathbb{N}} \left(\underbrace{\mathcal{U} \cdot \mathcal{U} \cdots \mathcal{U}}_{n \text{ times}} \right) \downarrow = \bigcup_{n \in \mathbb{N}} \underbrace{\mathcal{U} \downarrow \cdot \mathcal{U} \downarrow \cdots \mathcal{U} \downarrow}_{n \text{ times}} = \mathcal{U} \downarrow^* \quad \square$$

3.2 Concurrent Kleene Algebra

We now consider two extensions of Kleene Algebra (KA), known as *Bi-Kleene Algebra* (BKA) and *Concurrent Kleene Algebra* (CKA). Both extend KA with an operator for parallel composition and thus share a common syntax.

Definition 3.7. *The set \mathcal{T} is the smallest set generated by the grammar*

$$e, f ::= 0 \mid 1 \mid a \in \Sigma \mid e + f \mid e \cdot f \mid e \parallel f \mid e^*$$

The BKA-semantics of a term is a straightforward inductive application of the operators on the level of pomset languages. The CKA-semantics of a term is the BKA-semantics, downward-closed under the subsumption order; the CKA-semantics thus includes all possible sequentialisations.

Definition 3.8. *The function $\llbracket - \rrbracket_{\text{BKA}} : \mathcal{T} \rightarrow 2^{\text{SP}(\Sigma)}$ is defined as follows:*

$$\begin{aligned} \llbracket 0 \rrbracket_{\text{BKA}} &\triangleq \emptyset & \llbracket e + f \rrbracket_{\text{BKA}} &\triangleq \llbracket e \rrbracket_{\text{BKA}} \cup \llbracket f \rrbracket_{\text{BKA}} & \llbracket e^* \rrbracket_{\text{BKA}} &\triangleq \llbracket e \rrbracket_{\text{BKA}}^* \\ \llbracket 1 \rrbracket_{\text{BKA}} &\triangleq \{1\} & \llbracket e \cdot f \rrbracket_{\text{BKA}} &\triangleq \llbracket e \rrbracket_{\text{BKA}} \cdot \llbracket f \rrbracket_{\text{BKA}} \\ \llbracket a \rrbracket_{\text{BKA}} &\triangleq \{a\} & \llbracket e \parallel f \rrbracket_{\text{BKA}} &\triangleq \llbracket e \rrbracket_{\text{BKA}} \parallel \llbracket f \rrbracket_{\text{BKA}} \end{aligned}$$

Finally, $\llbracket - \rrbracket_{\text{CKA}} : \mathcal{T} \rightarrow 2^{\text{SP}(\Sigma)}$ is defined as $\llbracket e \rrbracket_{\text{CKA}} \triangleq \llbracket e \rrbracket_{\text{BKA}} \downarrow$.

Following Lodaya and Weil [22], if \mathcal{U} is a pomset language such that $\mathcal{U} = \llbracket e \rrbracket_{\text{BKA}}$ for some $e \in \mathcal{T}$, we say that the language \mathcal{U} is *series-rational*. Note that if \mathcal{U} is such that $\mathcal{U} = \llbracket e \rrbracket_{\text{CKA}}$ for some term $e \in \mathcal{T}$, then \mathcal{U} is closed by definition.

To axiomatise semantic equivalence between terms, we build the following relations, which match the axioms proposed in [20]. The axioms of CKA as defined in [8] come from a double quantale structure mediated by the exchange law; these imply the ones given here. The converse implication does not hold; in particular, our syntax does not include an infinitary greatest lower bound operator. However, BKA (as defined in this paper) does have a *finitary* greatest lower bound [20], and by the existence of closure, so does CKA.

Definition 3.9. *The relation \equiv_{BKA} is the smallest congruence on \mathcal{T} (with respect to all operators) such that for all $e, f, g \in \mathcal{T}$:*

$$\begin{aligned}
 e + 0 &\equiv_{\text{BKA}} e & e + e &\equiv_{\text{BKA}} e & e + f &\equiv_{\text{BKA}} f + e & e + (f + g) &\equiv_{\text{BKA}} (f + g) + e \\
 e \cdot 1 &\equiv_{\text{BKA}} e & 1 \cdot e &\equiv_{\text{BKA}} e & e \cdot (f \cdot g) &\equiv_{\text{BKA}} (e \cdot f) \cdot g \\
 e \cdot 0 &\equiv_{\text{BKA}} 0 \equiv_{\text{BKA}} 0 \cdot e & e \cdot (f + g) &\equiv_{\text{BKA}} e \cdot f + e \cdot g & (e + f) \cdot g &\equiv_{\text{BKA}} e \cdot g + f \cdot g \\
 e \parallel f &\equiv_{\text{BKA}} f \parallel e & e \parallel 1 &\equiv_{\text{BKA}} e & e \parallel (f \parallel g) &\equiv_{\text{BKA}} (e \parallel f) \parallel g \\
 e \parallel 0 &\equiv_{\text{BKA}} 0 & e \parallel (f + g) &\equiv_{\text{BKA}} e \parallel f + e \parallel g & 1 + e \cdot e^* &\equiv_{\text{BKA}} e^* \\
 e + f \cdot g &\leq_{\text{BKA}} g \implies f^* \cdot e \leq_{\text{BKA}} g
 \end{aligned}$$

in which we use $e \leq_{\text{BKA}} f$ as a shorthand for $e + f \equiv_{\text{BKA}} f$. The final (conditional) axiom is referred to as the least fixpoint axiom.

The relation \equiv_{CKA} is the smallest congruence on \mathcal{T} that satisfies the rules of \equiv_{BKA} , and furthermore satisfies the exchange law for all $e, f, g, h \in \mathcal{T}$:

$$(e \parallel f) \cdot (g \parallel h) \leq_{\text{CKA}} (e \cdot g) \parallel (f \cdot h)$$

where we similarly use $e \leq_{\text{CKA}} f$ as a shorthand for $e + f \equiv_{\text{CKA}} f$.

We can see that \equiv_{BKA} includes the familiar axioms of KA, and stipulates that \parallel is commutative and associative with unit 1 and annihilator 0, as well as distributive over $+$. When using CKA to model concurrent program flow, the exchange law models sequentialisation: if we have two programs, the first of which executes e followed by g , and the second of which executes f followed by h , then we can sequentialise this by executing e and f in parallel, followed by executing g and h in parallel.

We use the symbol \top in statements that are true for $\top \in \{\text{BKA}, \text{CKA}\}$. The relation \equiv_{\top} is sound for equivalence of terms under \top [13].

Lemma 3.8. *Let $e, f \in \mathcal{T}$. If $e \equiv_{\top} f$, then $\llbracket e \rrbracket_{\top} = \llbracket f \rrbracket_{\top}$.*

Since all binary operators are associative (up to \equiv_{\top}), we drop parentheses when writing terms like $e + f + g$ —this does not incur ambiguity with regard to $\llbracket - \rrbracket_{\top}$. We furthermore consider \cdot to have precedence over \parallel , which has precedence over $+$; as usual, the Kleene star has the highest precedence of all operators. For instance, when we write $e + f \cdot g^* \parallel h$, this should be read as $e + ((f \cdot (g^*)) \parallel h)$.

In case of BKA, the implication in Lemma 3.8 is an equivalence [20], and thus gives a complete axiomatisation of semantic BKA-equivalence of terms.²

Theorem 3.1. *Let $e, f \in \mathcal{T}$. Then $e \equiv_{\text{BKA}} f$ if and only if $\llbracket e \rrbracket_{\text{BKA}} = \llbracket f \rrbracket_{\text{BKA}}$.*

Given a term $e \in \mathcal{T}$, we can determine syntactically whether its (BKA or CKA) semantics contains the empty pomset, using the function defined below.

² Strictly speaking, the proof in [20] includes the parallel star operator in BKA. Since this is a conservative extension of BKA, this proof applies to BKA as well.

Definition 3.10. The nullability function $\epsilon : \mathcal{T} \rightarrow 2$ is defined as follows:

$$\begin{aligned} \epsilon(0) &\triangleq 0 & \epsilon(e + f) &\triangleq \epsilon(e) \vee \epsilon(f) & \epsilon(e^*) &\triangleq 1 \\ \epsilon(1) &\triangleq 1 & \epsilon(e \cdot f) &\triangleq \epsilon(e) \wedge \epsilon(f) \\ \epsilon(a) &\triangleq 0 & \epsilon(e \parallel f) &\triangleq \epsilon(e) \wedge \epsilon(f) \end{aligned}$$

in which \vee and \wedge are understood as the usual lattice operations on 2.

That ϵ encodes the presence of 1 in the semantics is witnessed by the following.

Lemma 3.9. Let $e \in \mathcal{T}$. Then $\epsilon(e) \leq_{\tau} e$ and $1 \in \llbracket e \rrbracket_{\tau}$ if and only if $\epsilon(e) = 1$.

In the sequel, we need the (parallel) width of a term. This is defined as follows.

Definition 3.11. Let $e \in \mathcal{T}$. The (parallel) width of e , denoted by $|e|$, is defined as 0 when $e \equiv_{\text{BKA}} 0$; for all other cases, it is defined inductively, as follows:

$$\begin{aligned} |1| &\triangleq 0 & |e + f| &\triangleq \max(|e|, |f|) & |e \parallel f| &\triangleq |e| + |f| \\ |a| &\triangleq 1 & |e \cdot f| &\triangleq \max(|e|, |f|) & |e^*| &\triangleq |e| \end{aligned}$$

The width of a term is invariant with respect to equivalence of terms.

Lemma 3.10. Let $e, f \in \mathcal{T}$. If $e \equiv_{\text{BKA}} f$, then $|e| = |f|$.

The width of a term is related to its semantics as demonstrated below.

Lemma 3.11. Let $e \in \mathcal{T}$, and let $U \in \llbracket e \rrbracket_{\text{BKA}}$ be such that $U \neq 1$. Then $|e| > 0$.

3.3 Linear Systems

KA is equipped to find the least solutions to linear inequations. For instance, if we want to find X such that $e \cdot X + f \leq_{\text{KA}} X$, it is not hard to show that $e^* \cdot f$ is the *least solution* for X , in the sense that this choice of X satisfies the inequation, and for any choice of X that also satisfies this inequation it holds that $e^* \cdot f \leq_{\text{KA}} X$. Since KA is contained in BKA and CKA, the same constructions also apply there. These axioms generalise to systems of linear inequations in a straightforward manner; indeed, Kozen [18] exploited this generalisation to axiomatise KA. In this paper, we use systems of linear inequations to construct particular expressions. To do this, we introduce vectors and matrices of terms.

For the remainder of this section, we fix I as a finite set.

Definition 3.12. An I -vector is a function from I to \mathcal{T} . Addition of I -vectors is defined pointwise, i.e., if p and q are I -vectors, then $p + q$ is the I -vector defined for $i \in I$ by $(p + q)(i) \triangleq p(i) + q(i)$.

An I -matrix is a function from I^2 to \mathcal{T} . Left-multiplication of an I -vector by an I -matrix is defined in the usual fashion, i.e., if M is an I -matrix and p is an I -vector, then $M \cdot p$ is the I -vector defined for $i \in I$ by

$$(M \cdot p)(i) \triangleq \sum_{j \in I} M(i, j) \cdot p(j)$$

Equivalence between terms extends pointwise to I -vectors. More precisely, we write $p \equiv_{\tau} q$ for I -vectors p and q when $p(i) \equiv_{\tau} q(i)$ for all $i \in I$, and $p \leq_{\tau} q$ when $p + q \equiv_{\tau} q$.

Definition 3.13. An I -linear system \mathcal{L} is a pair $\langle M, p \rangle$ where M is an I -matrix and p is an I -vector. A solution to \mathcal{L} in T is an I -vector s such that $M \cdot s + p \leq_{\tau} s$. A least solution to \mathcal{L} in T is a solution s in T such that for any solution t in T it holds that $s \leq_{\tau} t$.

It is not very hard to show that least solutions of a linear system are unique, up to \equiv_{τ} ; we therefore speak of *the* least solution of a linear system.

Interestingly, *any* I -linear system has a least solution, and one can construct this solution using only the operators of KA. The construction proceeds by induction on $|I|$. In the base, where I is empty, the solution is trivial; for the inductive step it suffices to reduce the problem to finding the least solution of a strictly smaller linear system. This construction is not unlike Kleene's procedure to obtain a regular expression from a finite automaton [17]. Alternatively, we can regard the existence of least solutions as a special case of Kozen's proof of the fixpoint for matrices over a KA, as seen in [18, Lemma 9].

As a matter of fact, because this construction uses the axioms of KA exclusively, the least solution that is constructed is the same for both BKA and CKA.

Lemma 3.12. Let \mathcal{L} be an I -linear system. One can construct a single I -vector x that is the least solution to \mathcal{L} in both BKA and CKA.

We include a full proof of the lemma above using the notation of this paper in the full version of this paper [15].

4 Completeness of CKA

We now turn our attention to proving that \equiv_{CKA} is complete for CKA-semantic equivalence of terms, i.e., that if $e, f \in \mathcal{T}$ are such that $\llbracket e \rrbracket_{\text{CKA}} = \llbracket f \rrbracket_{\text{CKA}}$, then $e \equiv_{\text{CKA}} f$. In the interest of readability, proofs of technical lemmas in this section can be found in the full version of this paper [15].

As mentioned before, our proof of completeness is based on the completeness result for BKA reproduced in Theorem 3.1. Recall that $\llbracket e \rrbracket_{\text{CKA}} = \llbracket e \rrbracket_{\text{BKA}} \downarrow$. To reuse completeness of BKA, we construct a syntactic variant of the closure operator, which is formalised below.

Definition 4.1. Let $e \in \mathcal{T}$. We say that $e \downarrow$ is a closure of e if both $e \equiv_{\text{CKA}} e \downarrow$ and $\llbracket e \downarrow \rrbracket_{\text{BKA}} = \llbracket e \rrbracket_{\text{BKA}} \downarrow$ hold.

Example 4.1. Let $e = a \parallel b$; as proposed in Sect. 2, we claim that $e \downarrow = a \parallel b + b \cdot a + a \cdot b$ is a closure of e . To see why, first note that $e \leq_{\text{CKA}} e \downarrow$ by construction. Furthermore,

$$ab \equiv_{\text{CKA}} (a \parallel 1) \cdot (1 \parallel b) \leq_{\text{CKA}} (a \cdot 1) \parallel (1 \cdot b) \equiv_{\text{CKA}} a \parallel b$$

and similarly $ba \leq_{\text{CKA}} e$; thus, $e \equiv_{\text{CKA}} e \downarrow$. Lastly, the pomsets in $\llbracket e \rrbracket_{\text{BKA}} \downarrow$ and $\llbracket e \downarrow \rrbracket_{\text{BKA}}$ are simply $a \parallel b$, ab and ba , and therefore $\llbracket e \downarrow \rrbracket_{\text{BKA}} = \llbracket e \rrbracket_{\text{BKA}} \downarrow$.

Laurence and Struth observed that the existence of a closure for every term implies a completeness theorem for CKA, as follows.

Lemma 4.1. *Suppose that we can construct a closure for every element of \mathcal{T} . If $e, f \in \mathcal{T}$ such that $\llbracket e \rrbracket_{\text{CKA}} = \llbracket f \rrbracket_{\text{CKA}}$, then $e \equiv_{\text{CKA}} f$.*

Proof. Since $\llbracket e \rrbracket_{\text{CKA}} = \llbracket e \rrbracket_{\text{BKA}} \downarrow = \llbracket e \downarrow \rrbracket_{\text{BKA}}$ and similarly $\llbracket f \rrbracket_{\text{CKA}} = \llbracket f \downarrow \rrbracket_{\text{BKA}}$, we have $\llbracket e \downarrow \rrbracket_{\text{BKA}} = \llbracket f \downarrow \rrbracket_{\text{BKA}}$. By Theorem 3.1, we get $e \downarrow \equiv_{\text{BKA}} f \downarrow$, and thus $e \downarrow \equiv_{\text{CKA}} f \downarrow$, since all axioms of BKA are also axioms of CKA. By $e \equiv_{\text{CKA}} e \downarrow$ and $f \downarrow \equiv_{\text{CKA}} f$, we can then conclude that $e \equiv_{\text{CKA}} f$. \square

The remainder of this section is dedicated to showing that the premise of Lemma 4.1 holds. We do this by explicitly constructing a closure $e \downarrow$ for every $e \in \mathcal{T}$. First, we note that closure can be constructed for the base terms.

Lemma 4.2. *Let $e \in 2$ or $e = a$ for some $a \in \Sigma$. Then e is a closure of itself.*

Furthermore, closure can be constructed compositionally for all operators except parallel composition, in the following sense.

Lemma 4.3. *Suppose that $e_0, e_1 \in \mathcal{T}$, and that e_0 and e_1 have closures $e_0 \downarrow$ and $e_1 \downarrow$. Then (i) $e_0 \downarrow + e_1 \downarrow$ is a closure of $e_0 + e_1$, (ii) $e_0 \downarrow \cdot e_1 \downarrow$ is a closure of $e_0 \cdot e_1$, and (iii) $(e_0 \downarrow)^*$ is a closure of e_0^* .*

Proof. Since $e_0 \downarrow \equiv_{\text{CKA}} e_0$ and $e_1 \downarrow \equiv_{\text{CKA}} e_1$, by the fact that \equiv_{CKA} is a congruence we obtain $e_0 \downarrow + e_1 \downarrow \equiv_{\text{CKA}} e_0 + e_1$. Similar observations hold for the other operators. We conclude using Lemma 3.7. \square

It remains to consider the case where $e = e_0 \parallel e_1$. In doing so, our induction hypothesis is that any $f \in \mathcal{T}$ with $|f| < |e_0 \parallel e_1|$ has a closure, as well as any strict subterm of $e_0 \parallel e_1$.

4.1 Preclosure

To get to a closure of a parallel composition, we first need an operator on terms that is not a closure quite yet, but whose BKA-semantics is “closed enough” to cover the non-sequential elements of the CKA-semantics of the term.

Definition 4.2. *Let $e \in \mathcal{T}$. A preclosure of e is a term $\tilde{e} \in \mathcal{T}$ such that $\tilde{e} \equiv_{\text{CKA}} e$. Moreover, if $U \in \llbracket e \rrbracket_{\text{CKA}}$ is non-sequential, then $U \in \llbracket \tilde{e} \rrbracket_{\text{BKA}}$.*

Example 4.2. Suppose that $e_0 \parallel e_1 = (a \parallel b) \parallel c$. A preclosure of $e_0 \parallel e_1$ could be

$$\tilde{e} = a \parallel b \parallel c + (a \cdot b + b \cdot a) \parallel c + (b \cdot c + c \cdot b) \parallel a + (a \cdot c + c \cdot a) \parallel b$$

To verify this, note that $e \leq_{\text{CKA}} \tilde{e}$ by construction; remains to show that $\tilde{e} \leq_{\text{CKA}} e$. This is fairly straightforward: since $a \cdot b + b \cdot a \leq_{\text{CKA}} a \parallel b$, we have $(a \cdot b + b \cdot a) \parallel c \leq_{\text{CKA}} e$; the other terms are treated similarly. Consequently, $e \equiv_{\text{CKA}} \tilde{e}$. Furthermore, there are seven non-sequential pomsets in $\llbracket e \rrbracket_{\text{CKA}}$; they are

$$a \parallel b \parallel c \quad ab \parallel c \quad ba \parallel c \quad bc \parallel a \quad cb \parallel a \quad ac \parallel b \quad ca \parallel b$$

Each of these pomsets is found in $\llbracket \tilde{e} \rrbracket_{\text{BKA}}$. It should be noted that \tilde{e} is *not* a closure of e ; to see this, consider for instance that $abc \in \llbracket e \rrbracket_{\text{CKA}}$, while $abc \notin \llbracket \tilde{e} \rrbracket_{\text{BKA}}$.

The remainder of this section is dedicated to showing that, under the induction hypothesis, we can construct a preclosure for any parallelly composed term. This is not perfectly straightforward; for instance, consider the term $e_0 \parallel e_1$ discussed in Example 4.2. At first glance, one might be tempted to choose $e_0 \downarrow \parallel e_1 \downarrow$ as a preclosure, since $e_0 \downarrow$ and $e_1 \downarrow$ exist by the induction hypothesis. In that case, $e_0 \downarrow = a \parallel b + a \cdot b + b \cdot a$ is a closure of e_0 . Furthermore, $e_1 \downarrow = c$ is a closure of e_1 , by Lemma 4.2. However, $e_0 \downarrow \parallel e_1 \downarrow$ is not a preclosure of $e_0 \parallel e_1$, since $(a \cdot c) \parallel b$ is non-sequential and found in $\llbracket e_0 \parallel e_1 \rrbracket_{\text{CKA}}$, but not in $\llbracket e_0 \downarrow \parallel e_1 \downarrow \rrbracket_{\text{BKA}}$.

The problem is that the preclosure of e_0 and e_1 should also allow (partial) sequentialisation of *parallel parts* of e_0 and e_1 ; in this case, we need to sequentialise the a part of $a \parallel b$ with c , and leave b untouched. To do so, we need to be able to *split* $e_0 \parallel e_1$ into pairs of constituent terms, each of which represents a possible way to divvy up its parallel parts. For instance, we can split $e_0 \parallel e_1 = (a \parallel b) \parallel c$ parallelly into $a \parallel b$ and c , but also into a and $b \parallel c$, or into $a \parallel c$ and b . The definition below formalises this procedure.

Definition 4.3. Let $e \in \mathcal{T}$; Δ_e is the smallest relation on \mathcal{T} such that

$$\begin{array}{c} \frac{}{1 \Delta_e e} \quad \frac{}{e \Delta_e 1} \quad \frac{\ell \Delta_{e_0} r}{\ell \Delta_{e_1 + e_0} r} \quad \frac{\ell \Delta_{e_1} r}{\ell \Delta_{e_0 + e_1} r} \quad \frac{\ell \Delta_e r}{\ell \Delta_{e^*} r} \\[10pt] \frac{\ell \Delta_{e_0} r \quad \epsilon(e_1) = 1}{\ell \Delta_{e_0 \cdot e_1} r} \quad \frac{\ell \Delta_{e_1} r \quad \epsilon(e_0) = 1}{\ell \Delta_{e_0 \cdot e_1} r} \quad \frac{\ell_0 \Delta_{e_0} r_0 \quad \ell_1 \Delta_{e_1} r_1}{\ell_0 \parallel \ell_1 \Delta_{e_0 \parallel e_1} r_0 \parallel r_1} \end{array}$$

Given $e \in \mathcal{T}$, we refer to Δ_e as the *parallel splitting relation* of e , and to the elements of Δ_e as *parallel splices* of e . Before we can use Δ_e to construct the preclosure of e , we go over a number of properties of the parallel splitting relation. The first of these properties is that a given $e \in \mathcal{T}$ has only finitely many parallel splices. This will be useful later, when we involve *all* parallel splices of e in building a new term, i.e., to guarantee that the constructed term is finite.

Lemma 4.4. For $e \in \mathcal{T}$, Δ_e is finite.

We furthermore note that the parallel composition of any parallel splice of e is ordered below e by \leq_{BKA} . This guarantees that parallel splices never contain extra information, i.e., that their semantics do not contain pomsets that do not occur in the semantics of e . It also allows us to bound the width of the parallel splices by the width of the term being split, as a result of Lemma 3.10.

Lemma 4.5. Let $e \in \mathcal{T}$. If $\ell \Delta_e r$, then $\ell \parallel r \leq_{\text{BKA}} e$.

Corollary 4.1. Let $e \in \mathcal{T}$. If $\ell \Delta_e r$, then $|\ell| + |r| \leq |e|$.

Finally, we show that Δ_e is *dense* when it comes to parallel pomsets, meaning that if we have a parallelly composed pomset in the semantics of e , then we can find a parallel splice where one parallel component is contained in the semantics of one side of the pair, and the other component in that of the other.

Lemma 4.6. *Let $e \in \mathcal{T}$, and let V, W be pomsets such that $V \parallel W \in \llbracket e \rrbracket_{\text{BKA}}$. Then there exist $\ell, r \in \mathcal{T}$ with $\ell \Delta_e r$ such that $V \in \llbracket \ell \rrbracket_{\text{BKA}}$ and $W \in \llbracket r \rrbracket_{\text{BKA}}$.*

Proof. The proof proceeds by induction on e . In the base, we can discount the case where $e = 0$, for then the claim holds vacuously. This leaves us two cases.

- If $e = 1$, then $V \parallel W \in \llbracket e \rrbracket_{\text{BKA}}$ entails $V \parallel W = 1$. By Lemma 3.1, we find that $V = W = 1$. Since $1 \Delta_e 1$ by definition of Δ_e , the claim follows when we choose $\ell = r = 1$.
- If $e = a$ for some $a \in \Sigma$, then $V \parallel W \in \llbracket e \rrbracket_{\text{BKA}}$ entails $V \parallel W = a$. By Lemma 3.1, we find that either $V = 1$ and $W = a$, or $V = a$ and $W = 1$. In the former case, we can choose $\ell = 1$ and $r = a$, while in the latter case we can choose $\ell = a$ and $r = 1$. It is then easy to see that our claim holds in either case.

For the inductive step, there are four cases to consider.

- If $e = e_0 + e_1$, then $U_0 \parallel U_1 \in \llbracket e_i \rrbracket_{\text{BKA}}$ for some $i \in 2$. But then, by induction, we find $\ell, r \in \mathcal{T}$ with $\ell \Delta_{e_i} r$ such that $V \in \llbracket \ell \rrbracket_{\text{BKA}}$ and $W \in \llbracket r \rrbracket_{\text{BKA}}$. Since this implies that $\ell \Delta_e r$, the claim follows.
- If $e = e_0 \cdot e_1$, then there exist pomsets U_0, U_1 such that $V \parallel W = U_0 \cdot U_1$, and $U_i \in \llbracket e_i \rrbracket_{\text{BKA}}$ for all $i \in 2$. By Lemma 3.1, there are two cases to consider.
 - Suppose that $U_i = 1$ for some $i \in 2$, meaning that $V \parallel W = U_0 \cdot U_1 = U_{1-i} \in \llbracket e_{1-i} \rrbracket_{\text{BKA}}$ for this i . By induction, we find $\ell, r \in \mathcal{T}$ with $\ell \Delta_{e_{1-i}} r$, and $V \in \llbracket \ell \rrbracket_{\text{BKA}}$ as well as $W \in \llbracket r \rrbracket_{\text{BKA}}$. Since $U_i = 1 \in \llbracket e_i \rrbracket_{\text{BKA}}$, we have that $\epsilon(e_i) = 1$ by Lemma 3.9, and thus $\ell \Delta_e r$.
 - Suppose that $V = 1$ or $W = 1$. In the former case, $V \parallel W = W = U_0 \cdot U_1 \in \llbracket e \rrbracket_{\text{CKA}}$. We then choose $\ell = 1$ and $r = e$ to satisfy the claim. In the latter case, we can choose $\ell = e$ and $r = 1$ to satisfy the claim analogously.
- If $e = e_0 \parallel e_1$, then there exist pomsets U_0, U_1 such that $V \parallel W = U_0 \parallel U_1$, and $U_i \in \llbracket e_i \rrbracket_{\text{BKA}}$ for all $i \in 2$. By Lemma 3.5, we find pomsets V_0, V_1, W_0, W_1 such that $V = V_0 \parallel V_1$, $W = W_0 \parallel W_1$, and $U_i = V_i \parallel W_i$ for $i \in 2$. For $i \in 2$, we then find by induction $\ell_i, r_i \in \mathcal{T}$ with $\ell_i \Delta_{e_i} r_i$ such that $V_i \in \llbracket \ell_i \rrbracket_{\text{BKA}}$ and $W_i \in \llbracket r_i \rrbracket_{\text{BKA}}$. We then choose $\ell = \ell_0 \parallel \ell_1$ and $r = r_0 \parallel r_1$. Since $V = V_0 \parallel V_1$, it follows that $V \in \llbracket \ell \rrbracket_{\text{BKA}}$, and similarly we find that $W \in \llbracket r \rrbracket_{\text{BKA}}$. Since $\ell \Delta_e r$, the claim follows.
- If $e = e_0^*$, then there exist $U_0, U_1, \dots, U_{n-1} \in \llbracket e_0 \rrbracket_{\text{BKA}}$ such that $V \parallel W = U_0 \cdot U_1 \cdots U_{n-1}$. If $n = 0$, i.e., $V \parallel W = 1$, then $V = W = 1$. In that case, we can choose $\ell = e$ and $r = 1$ to find that $\ell \Delta_e r$, $V \in \llbracket \ell \rrbracket_{\text{BKA}}$ and $W \in \llbracket r \rrbracket_{\text{BKA}}$, satisfying the claim.
 If $n > 0$, we can assume without loss of generality that, for $0 \leq i < n$, it holds that $U_i \neq 1$. By Lemma 3.1, there are two subcases to consider.
 - Suppose that $V, W \neq 1$; then $n = 1$ (for otherwise $U_j = 1$ for some $0 \leq j < n$ by Lemma 3.1, which contradicts the above). Since $V \parallel W = U_0 \in \llbracket e_0 \rrbracket_{\text{BKA}}$, we find by induction $\ell, r \in \mathcal{T}$ with $\ell \Delta_{e_0} r$ such that $V \in \llbracket \ell \rrbracket_{\text{BKA}}$ and $W \in \llbracket r \rrbracket_{\text{BKA}}$. The claim then follows by the fact that $\ell \Delta_e r$.

- Suppose that $V = 1$ or $W = 1$. In the former case, $V \parallel W = W = U_0 \cdot U_1 \cdots U_{n-1} \in \llbracket e \rrbracket_{\text{CKA}}$. We then choose $\ell = 1$ and $r = e$ to satisfy the claim. In the latter case, we can choose $\ell = e$ and $r = 1$ to satisfy the claim analogously. \square

Example 4.3. Let $U = a \parallel c$ and $V = b$, and note that $U \parallel V \in \llbracket e_0 \parallel e_1 \rrbracket_{\text{CKA}}$. We can then find that $a \Delta_a 1$ and $1 \Delta_b b$, and thus $a \parallel 1 \Delta_{e_0} 1 \parallel b$. Since also $c \Delta_c 1$, it follows that $(a \parallel 1) \parallel c \Delta_{e_0 \parallel e_1} (1 \parallel b) \parallel 1$. We can then choose $\ell = (a \parallel 1) \parallel c$ and $r = (1 \parallel b) \parallel 1$ to find that $U \in \llbracket \ell \rrbracket_{\text{BKA}}$ and $V \in \llbracket r \rrbracket_{\text{BKA}}$, while $\ell \Delta_{e_0 \parallel e_1} r$.

With parallel splitting in hand, we can define an operator on terms that combines all parallel splices of a parallel composition in a way that accounts for all of their downward closures.

Definition 4.4. Let $e, f \in \mathcal{T}$, and suppose that, for every $g \in \mathcal{T}$ such that $|g| < |e| + |f|$, there exists a closure $g\downarrow$. The term $e \odot f$ is defined as follows:

$$e \odot f \triangleq e \parallel f + \sum_{\substack{\ell \Delta_{e \parallel f} r \\ |\ell|, |r| < |e \parallel f|}} \ell \downarrow \parallel r \downarrow$$

Note that $e \odot f$ is well-defined: the sum is finite since $\Delta_{e \parallel f}$ is finite by Lemma 4.4, and furthermore $\ell \downarrow$ and $r \downarrow$ exist, as we required that $|\ell|, |r| < |e \parallel f|$.

Example 4.4. Let us compute $e_0 \odot e_1$ and verify that we obtain a preclosure of $e_0 \parallel e_1$. Working through the definition, we see that $\Delta_{e_0 \parallel e_1}$ consists of the pairs

$$\begin{aligned} \langle (1 \parallel 1) \parallel 1, (a \parallel b) \parallel c \rangle & \quad \langle (1 \parallel 1) \parallel c, (a \parallel b) \parallel 1 \rangle & \quad \langle (1 \parallel b) \parallel 1, (a \parallel 1) \parallel c \rangle \\ \langle (1 \parallel b) \parallel c, (a \parallel 1) \parallel 1 \rangle & \quad \langle (a \parallel 1) \parallel 1, (1 \parallel b) \parallel c \rangle & \quad \langle (a \parallel 1) \parallel c, (1 \parallel b) \parallel 1 \rangle \end{aligned}$$

Since closure is invariant with respect to \equiv_{CKA} , we can simplify these terms by applying the axioms of CKA. After folding the unit subterms, we are left with

$$\langle 1, a \parallel b \parallel c \rangle \quad \langle c, a \parallel b \rangle \quad \langle b, a \parallel c \rangle \quad \langle b \parallel c, a \rangle \quad \langle a, b \parallel c \rangle \quad \langle a \parallel c, b \rangle$$

Recall that $a \parallel b + a \cdot b + b \cdot a$ is a closure of $a \parallel b$. Now, we find that

$$\begin{aligned} e_0 \odot e_1 &= (a \parallel b) \parallel c + c \parallel (a \parallel b + a \cdot b + b \cdot a) \\ &\quad + b \parallel (a \parallel c + a \cdot c + c \cdot a) + (b \parallel c + b \cdot c + c \cdot b) \parallel a \\ &\quad + a \parallel (b \parallel c + b \cdot c + c \cdot b) + (a \parallel c + a \cdot c + c \cdot a) \parallel b \\ &\equiv_{\text{CKA}} a \parallel b \parallel c + a \parallel (b \cdot c + c \cdot b) + b \parallel (a \cdot c + c \cdot a) + c \parallel (a \cdot b + b \cdot a) \end{aligned}$$

which was shown to be a preclosure of $e_0 \parallel e_1$ in Example 4.2.

The general proof of correctness for \odot as a preclosure plays out as follows.

Lemma 4.7. *Let $e, f \in \mathcal{T}$, and suppose that, for every $g \in \mathcal{T}$ with $|g| < |e| + |f|$, there exists a closure $g\downarrow$. Then $e \odot f$ is a preclosure of $e \parallel f$.*

Proof. We start by showing that $e \odot f \equiv_{\text{CKA}} e \parallel f$. First, note that $e \parallel f \leq_{\text{BKA}} e \odot f$ by definition of $e \odot f$. For the other direction, suppose that $\ell, r \in \mathcal{T}$ are such that $\ell \Delta_{e \parallel f} r$. By definition of closure, we know that $\ell\downarrow \parallel r\downarrow \equiv_{\text{CKA}} \ell \parallel r$. By Lemma 4.5, we have $\ell \parallel r \leq_{\text{BKA}} e \parallel f$. Since every subterm of $e \odot f$ is ordered below $e \parallel f$ by \leq_{CKA} , we have that $e \odot f \leq_{\text{CKA}} e \parallel f$. It then follows that $e \parallel f \equiv_{\text{CKA}} e \odot f$.

For the second requirement, suppose that $X \in \llbracket e \parallel f \rrbracket_{\text{CKA}}$ is non-sequential. We then know that there exists a $Y \in \llbracket e \parallel f \rrbracket_{\text{BKA}}$ such that $X \sqsubseteq Y$. This leaves us two cases to consider.

- If X is empty or primitive, then $Y = X$ by Lemma 3.2, thus $X \in \llbracket e \parallel f \rrbracket_{\text{BKA}}$. By the fact that $e \parallel f \leq_{\text{BKA}} e \odot f$ and by Lemma 3.8, we find $X \in \llbracket e \odot f \rrbracket_{\text{BKA}}$.
- If $X = X_0 \parallel X_1$ for non-empty pomsets X_0 and X_1 , then by Lemma 3.3 we find non-empty pomsets Y_0 and Y_1 with $Y = Y_0 \parallel Y_1$ such that $X_i \sqsubseteq Y_i$ for $i \in 2$. By Lemma 4.6, we find $\ell, r \in \mathcal{T}$ with $\ell \Delta_{e \parallel f} r$ such that $Y_0 \in \llbracket \ell \rrbracket_{\text{BKA}}$ and $Y_1 \in \llbracket r \rrbracket_{\text{BKA}}$. By Lemma 3.11, we find that $|\ell|, |r| \geq 1$. Corollary 4.1 then allows us to conclude that $|\ell|, |r| < |e \parallel f|$. This means that $\ell\downarrow \parallel r\downarrow \leq_{\text{BKA}} e \odot f$. Since $X_0 \in \llbracket \ell\downarrow \rrbracket_{\text{BKA}}$ and $X_1 \in \llbracket r\downarrow \rrbracket_{\text{BKA}}$ by definition of closure, we can derive by Lemma 3.8 that

$$X = X_0 \parallel X_1 \in \llbracket \ell\downarrow \parallel r\downarrow \rrbracket_{\text{BKA}} \subseteq \llbracket e \odot f \rrbracket_{\text{BKA}} \quad \square$$

4.2 Closure

The preclosure operator discussed above covers the non-sequential pomsets in the language $\llbracket e \parallel f \rrbracket_{\text{CKA}}$; it remains to find a term that covers the sequential pomsets contained in $\llbracket e \parallel f \rrbracket_{\text{CKA}}$.

To better give some intuition to the construction ahead, we first explore the observations that can be made when a sequential pomset $W \cdot X$ appears in the language $\llbracket e \parallel f \rrbracket_{\text{CKA}}$; without loss of generality, assume that W is non-sequential. In this setting, there must exist $U \in \llbracket e \rrbracket_{\text{BKA}}$ and $V \in \llbracket f \rrbracket_{\text{BKA}}$ such that $W \cdot X \sqsubseteq U \parallel V$. By Lemma 3.6, we find pomsets U_0, U_1, V_0, V_1 such that

$$W \sqsubseteq U_0 \parallel V_0 \quad X \sqsubseteq U_1 \parallel V_1 \quad U_0 \cdot U_1 \sqsubseteq U \quad V_0 \cdot V_1 \sqsubseteq V$$

This means that $U_0 \cdot U_1 \in \llbracket e \rrbracket_{\text{CKA}}$ and $V_0 \cdot V_1 \in \llbracket f \rrbracket_{\text{CKA}}$. Now, suppose we could find $e_0, e_1, f_0, f_1 \in \mathcal{T}$ such that

$$\begin{array}{lll} e_0 \cdot e_1 \leq_{\text{CKA}} e & U_0 \in \llbracket e_0 \rrbracket_{\text{CKA}} & U_1 \in \llbracket e_1 \rrbracket_{\text{CKA}} \\ f_0 \cdot f_1 \leq_{\text{CKA}} f & V_0 \in \llbracket f_0 \rrbracket_{\text{CKA}} & V_1 \in \llbracket f_1 \rrbracket_{\text{CKA}} \end{array}$$

Then we have $W \in \llbracket e_0 \odot f_0 \rrbracket_{\text{BKA}}$, and $X \in \llbracket e_1 \parallel f_1 \rrbracket_{\text{CKA}}$. Thus, if we can find a closure of $e_1 \parallel f_1$, then we have a term whose BKA-semantics contains $W \cdot X$.

There are two obstacles that need to be resolved before we can use the observations above to find the closure of $e \parallel f$. The first problem is that we need to be sure that this process of splitting terms into sequential components is at all possible, i.e., that we can split e into e_0 and e_1 with $e_0 \cdot e_1 \leq_{\text{CKA}} e$ and $U_i \in \llbracket e_i \rrbracket_{\text{CKA}}$ for $i \in 2$. We do this by designing a sequential analogue to the parallel splitting relation seen before. The second problem, which we will address later in this section, is whether this process of splitting a parallel term $e \parallel f$ according to the exchange law and finding a closure of remaining term $e_1 \parallel f_1$ is well-founded, i.e., if we can find “enough” of these terms to cover all possible ways of sequentialising $e \parallel f$. This will turn out to be possible, by using the fixpoint axioms of KA as in Sect. 3.3 with linear systems.

We start by defining the sequential splitting relation.³

Definition 4.5. Let $e \in \mathcal{T}$; ∇_e is the smallest relation on \mathcal{T} such that

$$\begin{array}{c} \frac{}{1 \nabla_1 1} \quad \frac{}{a \nabla_a 1} \quad \frac{}{1 \nabla_a a} \quad \frac{}{1 \nabla_{e_0^*} 1} \quad \frac{\ell \nabla_{e_0} r}{\ell \nabla_{e_0+e_1} r} \quad \frac{\ell \nabla_{e_1} r}{\ell \nabla_{e_0+e_1} r} \\[10pt] \frac{\ell \nabla_{e_0} r}{\ell \nabla_{e_0 \cdot e_1} r \cdot e_1} \quad \frac{\ell \nabla_{e_1} r}{e_0 \cdot \ell \nabla_{e_0 \cdot e_1} r} \quad \frac{\ell_0 \nabla_{e_0} r_0 \quad \ell_1 \nabla_{e_1} r_1}{\ell_0 \parallel \ell_1 \nabla_{e_0 \parallel e_1} r_0 \parallel r_1} \quad \frac{\ell \nabla_{e_0} r}{e_0^* \cdot \ell \nabla_{e_0^*} r \cdot e_0^*} \end{array}$$

Given $e \in \mathcal{T}$, we refer to ∇_e as the *sequential splitting relation* of e , and to the elements of ∇_e as *sequential splices* of e . We need to establish a few properties of the sequential splitting relation that will be useful later on. The first of these properties is that, as for parallel splitting, ∇_e is finite.

Lemma 4.8. For $e \in \mathcal{T}$, ∇_e is finite.

We also have that the sequential composition of splices is provably below the term being split. Just like the analogous lemma for parallel splitting, this guarantees that our sequential splices never give rise to semantics not contained in the split term. This lemma also yields an observation about the width of sequential splices when compared to the term being split.

Lemma 4.9. Let $e \in \mathcal{T}$. If $\ell, r \in \mathcal{T}$ with $\ell \nabla_e r$, then $\ell \cdot r \leq_{\text{CKA}} e$.

Corollary 4.2. Let $e \in \mathcal{T}$. If $\ell, r \in \mathcal{T}$ with $\ell \nabla_e r$, then $|\ell|, |r| \leq |e|$.

Lastly, we show that the splices cover every way of (sequentially) splitting up the semantics of the term being split, i.e., that ∇_e is dense when it comes to sequentially composed pomsets.

Lemma 4.10. Let $e \in \mathcal{T}$, and let V and W be pomsets such that $V \cdot W \in \llbracket e \rrbracket_{\text{CKA}}$. Then there exist $\ell, r \in \mathcal{T}$ with $\ell \nabla_e r$ such that $V \in \llbracket \ell \rrbracket_{\text{CKA}}$ and $W \in \llbracket r \rrbracket_{\text{CKA}}$.

Proof. The proof proceeds by induction on e . In the base, we can discount the case where $e = 0$, for then the claim holds vacuously. This leaves us two cases.

³ The contents of this relation are very similar to the set of *left- and right-spines* of a NetKAT expression as used in [5].

- If $e = 1$, then $V \cdot W = 1$; by Lemma 3.1, we find that $V = W = 1$. Since $1 \nabla_e 1$ by definition of ∇_e , the claim follows when we choose $\ell = r = 1$.
- If $e = a$ for some $a \in \Sigma$, then $V \cdot W = a$; by Lemma 3.1, we find that either $V = a$ and $W = 1$ or $V = 1$ and $W = a$. In the former case, we can choose $\ell = a$ and $r = 1$ to satisfy the claim; the latter case can be treated similarly.

For the inductive step, there are four cases to consider.

- If $e = e_0 + e_1$, then $V \cdot W \in \llbracket e_i \rrbracket_{\text{CKA}}$ for some $i \in 2$. By induction, we find $\ell, r \in \mathcal{T}$ with $\ell \nabla_{e_i} r$ such that $V \in \llbracket \ell \rrbracket_{\text{CKA}}$ and $W \in \llbracket r \rrbracket_{\text{CKA}}$. Since $\ell \nabla_e r$ in this case, the claim follows.
- If $e = e_0 \cdot e_1$, then there exist $U_0 \in \llbracket e_0 \rrbracket_{\text{CKA}}$ and $U_1 \in \llbracket e_1 \rrbracket_{\text{CKA}}$ such that $V \cdot W = U_0 \cdot U_1$. By Lemma 3.4, we find a series-parallel pomset X such that either $V \sqsubseteq U_0 \cdot X$ and $X \cdot W \sqsubseteq U_1$, or $V \cdot X \sqsubseteq U_0$ and $W \sqsubseteq X \cdot U_1$. In the former case, we find that $X \cdot W \in \llbracket e_1 \rrbracket_{\text{CKA}}$, and thus by induction $\ell', r \in \mathcal{T}$ with $\ell' \nabla_{e_1} r$ such that $X \in \llbracket \ell' \rrbracket_{\text{CKA}}$ and $W \in \llbracket r \rrbracket_{\text{CKA}}$. We then choose $\ell = e_0 \cdot \ell'$ to find that $\ell \nabla_e r$, as well as $V \sqsubseteq U_0 \cdot X \in \llbracket e_0 \rrbracket_{\text{CKA}} \cdot \llbracket \ell' \rrbracket_{\text{CKA}} = \llbracket \ell \rrbracket_{\text{CKA}}$ and thus $V \in \llbracket \ell \rrbracket_{\text{CKA}}$. The latter case can be treated similarly; here, we use the induction hypothesis on e_0 .
- If $e = e_0 \parallel e_1$, then there exist $U_0 \in \llbracket e_0 \rrbracket_{\text{CKA}}$ and $U_1 \in \llbracket e_1 \rrbracket_{\text{CKA}}$ such that $V \cdot W \sqsubseteq U_0 \parallel U_1$. By Lemma 3.6, we find series-parallel pomsets V_0, V_1, W_0, W_1 such that $V \sqsubseteq V_0 \parallel V_1$ and $W \sqsubseteq W_0 \parallel W_1$, as well as $V_i \cdot W_i \sqsubseteq U_i$ for all $i \in 2$. In that case, $V_i \cdot W_i \in \llbracket e_i \rrbracket_{\text{CKA}}$ for all $i \in 2$, and thus by induction we find $\ell_i, r_i \in \mathcal{T}$ with $\ell_i \nabla_{e_i} r_i$ such that $V_i \in \llbracket \ell_i \rrbracket_{\text{CKA}}$ and $W_i \in \llbracket r_i \rrbracket_{\text{CKA}}$. We choose $\ell = \ell_0 \parallel \ell_1$ and $r = r_0 \parallel r_1$ to find that $V \in \llbracket \ell_0 \parallel r_0 \rrbracket_{\text{CKA}}$ and $W \in \llbracket \ell_1 \parallel r_1 \rrbracket_{\text{CKA}}$, as well as $\ell \nabla_e r$.
- If $e = e_0^*$, then there exist $U_0, U_1, \dots, U_{n-1} \in \llbracket e_0 \rrbracket_{\text{CKA}}$ such that $V \cdot W = U_0 \cdot U_1 \cdots U_{n-1}$. Without loss of generality, we can assume that for $0 \leq i < n$ it holds that $U_i \neq 1$. In the case where $n = 0$ we have that $V \cdot W = 1$, thus $V = W = 1$, we can choose $\ell = r = 1$ to satisfy the claim.

For the case where $n > 0$, we find by Lemma 3.4 an $0 \leq m < n$ and series-parallel pomsets X, Y such that $X \cdot Y \sqsubseteq U_m$, and $V \sqsubseteq U_0 \cdot U_1 \cdots U_{m-1} \cdot X$ and $W \sqsubseteq Y \cdot U_{m+1} \cdot U_{m+2} \cdots U_n$. Since $X \cdot Y \sqsubseteq U_m \in \llbracket e_0 \rrbracket_{\text{CKA}}$ and thus $X \cdot Y \in \llbracket e_0 \rrbracket_{\text{CKA}}$, we find by induction $\ell', r' \in \mathcal{T}$ with $\ell' \nabla_{e_0} r'$ and $X \in \llbracket \ell' \rrbracket_{\text{CKA}}$ and $Y \in \llbracket r' \rrbracket_{\text{CKA}}$. We can then choose $\ell = e_0^* \cdot \ell'$ and $r = r' \cdot e_0^*$ to find that $V \sqsubseteq U_0 \cdot U_1 \cdots U_{m-1} \cdot X \in \llbracket e_0^* \rrbracket_{\text{CKA}} \cdot \llbracket \ell' \rrbracket_{\text{CKA}} = \llbracket \ell \rrbracket_{\text{CKA}}$ and $W \sqsubseteq Y \cdot U_{m+1} \cdot U_{m+2} \cdots U_n \in \llbracket r' \rrbracket_{\text{CKA}} \cdot \llbracket e_0^* \rrbracket_{\text{CKA}} = \llbracket r \rrbracket_{\text{CKA}}$, and thus that $V \in \llbracket \ell \rrbracket_{\text{CKA}}$ and $W \in \llbracket r \rrbracket_{\text{CKA}}$. Since $\ell \nabla_e r$ holds, the claim follows. \square

Example 4.5. Let U be the pomset ca and let V be bc . Furthermore, let e be the term $(a \cdot b + c)^*$, and note that $U \cdot V \in \llbracket e \rrbracket_{\text{CKA}}$. We then find that $a \nabla_a 1$, and thus $a \nabla_{a \cdot b} 1 \cdot b$. We can now choose $\ell = (a \cdot b + c)^* \cdot a$ and $r = (1 \cdot b) \cdot (a \cdot b + c)^*$ to find that $U \in \llbracket \ell \rrbracket_{\text{CKA}}$ and $V \in \llbracket r \rrbracket_{\text{CKA}}$, while $\ell \nabla_e r$.

We know how to split a term sequentially. To resolve the second problem, we need to show that the process of splitting terms repeatedly ends somewhere. This is formalised in the notion of *right-hand remainders*, which are the terms that can appear as the right hand of a sequential splice of a term.

Definition 4.6. Let $e \in \mathcal{T}$. The set of (right-hand) remainders of e , written $R(e)$, is the smallest satisfying the rules

$$\frac{}{e \in R(e)} \qquad \frac{f \in R(e) \quad \ell \nabla_f r}{r \in R(e)}$$

Lemma 4.11. Let $e \in \mathcal{T}$. $R(e)$ is finite.

With splitting and remainders we are in a position to define the linear system that will yield the closure of a parallel composition. Intuitively, we can think of this system as an automaton: every variable corresponds to a state, and every row of the matrix describes the “transitions” of the corresponding state, while every element of the vector describes the language “accepted” by that state without taking a single transition. Solving the system for a least fixpoint can be thought of as finding an expression that describes the language of the automaton.

Definition 4.7. Let $e, f \in \mathcal{T}$, and suppose that, for every $g \in \mathcal{T}$ such that $|g| < |e| + |f|$, there exists a closure $g\downarrow$. We choose

$$I_{e,f} = \{g \parallel h : g \in R(e), h \in R(f)\}$$

The $I_{e,f}$ -vector $p_{e,f}$ and $I_{e,f}$ -matrix $M_{e,f}$ are chosen as follows.

$$p_{e,f}(g \parallel h) \triangleq g \parallel f \qquad M_{e,f}(g \parallel h, g' \parallel h') \triangleq \sum_{\substack{\ell_g \nabla_g g' \\ \ell_h \nabla_h h'}} \ell_g \odot \ell_h$$

$I_{e,f}$ is finite by Lemma 4.11. We write $\mathfrak{L}_{e,f}$ for the $I_{e,f}$ -linear system $\langle M_{e,f}, p_{e,f} \rangle$.

We can check that $M_{e,f}$ is well-defined. First, the sum is finite, because ∇_g and ∇_h are finite by Lemma 4.8. Second, if $g \parallel h \in I$ and $\ell_g, r_g, \ell_h, r_h \in \mathcal{T}$ such that $\ell_g \nabla_g r_g$ and $\ell_h \nabla_h r_h$, then $|\ell_g| \leq |g| \leq |e|$ and $|\ell_h| \leq |h| \leq |f|$ by Corollary 4.2, and thus, if $d \in \mathcal{T}$ such that $|d| < |\ell_g| + |\ell_h|$, then $|d| < |e| + |f|$, and therefore a closure of d exists, meaning that $\ell_g \odot \ell_h$ exists, too.

The least solution to $\mathfrak{L}_{e,f}$ obtained through Lemma 3.12 is the I -vector denoted by $s_{e,f}$. We write $e \otimes f$ for $s_{e,f}(e \parallel f)$, i.e., the least solution at $e \parallel f$.

Using the previous lemmas, we can then show that $e \otimes f$ is indeed a closure of $e \parallel f$, provided that we have closures for all terms of strictly lower width. The intuition of this proof is that we use the uniqueness of least fixpoints to show that $e \parallel f \equiv_{\text{CKA}} e \otimes f$, and then use the properties of preclosure and the normal form of series-parallel pomsets to show that $\llbracket e \parallel f \rrbracket_{\text{CKA}} = \llbracket e \otimes f \rrbracket_{\text{BKA}}$.

Lemma 4.12. Let $e, f \in \mathcal{T}$, and suppose that, for every $g \in \mathcal{T}$ with $|g| < |e| + |f|$, there exists a closure $g\downarrow$. Then $e \otimes f$ is a closure of $e \parallel f$.

Proof. We begin by showing that $e \parallel f \equiv_{\text{CKA}} e \otimes f$. We can see that $p_{e,f}$ is a solution to $\mathfrak{L}_{e,f}$, by calculating for $g \parallel h \in I_{e,f}$:

$$\begin{aligned}
& (p_{e,f} + M_{e,f} \cdot p_{e,f})(g \parallel h) \\
&= g \parallel h + \sum_{r_g \parallel r_h \in I} \left(\sum_{\ell_g \nabla_g r_g} \ell_g \odot \ell_h \right) \cdot (r_g \parallel r_h) && (\text{def. } M_{e,f}, p_{e,f}) \\
&\equiv_{\text{CKA}} g \parallel h + \sum_{r_g \parallel r_h \in I} \sum_{\ell_h \nabla_h r_h} (\ell_g \odot \ell_h) \cdot (r_g \parallel r_h) && (\text{distributivity}) \\
&\equiv_{\text{CKA}} g \parallel h + \sum_{r_g \parallel r_h \in I} \sum_{\ell_h \nabla_h r_h} (\ell_g \parallel \ell_h) \cdot (r_g \parallel r_h) && (\text{Lemma 4.7}) \\
&\leq_{\text{CKA}} g \parallel h + \sum_{r_g \parallel r_h \in I} \sum_{\ell_h \nabla_h r_h} (\ell_g \cdot r_g) \parallel (\ell_h \cdot r_h) && (\text{exchange}) \\
&\leq_{\text{CKA}} g \parallel h + \sum_{r_g \parallel r_h \in I} \sum_{\ell_h \nabla_h r_h} g \parallel h && (\text{Lemma 4.9}) \\
&\equiv_{\text{CKA}} g \parallel h && (\text{idempotence}) \\
&= p_{e,f}(g \parallel h) && (\text{def. } p_{e,f})
\end{aligned}$$

To see that $p_{e,f}$ is the *least* solution to $\mathfrak{L}_{e,f}$, let $q_{e,f}$ be a solution to $\mathfrak{L}_{e,f}$. We then know that $M_{e,f} \cdot q_{e,f} + p_{e,f} \leq_{\text{CKA}} q_{e,f}$; thus, in particular, $p_{e,f} \leq_{\text{CKA}} q_{e,f}$. Since the least solution to a linear system is unique up to \equiv_{CKA} , we find that $s_{e,f} \equiv_{\text{CKA}} p_{e,f}$, and therefore that $e \otimes f = s_{e,f}(e \parallel f) \equiv_{\text{CKA}} p_{e,f}(e \parallel f) = e \parallel f$.

It remains to show that if $U \in \llbracket e \parallel f \rrbracket_{\text{CKA}}$, then $U \in \llbracket e \otimes f \rrbracket_{\text{BKA}}$. To show this, we show the more general claim that if $g \parallel h \in I$ and $U \in \llbracket g \parallel h \rrbracket_{\text{CKA}}$, then $U \in \llbracket s_{e,f}(g \parallel h) \rrbracket_{\text{BKA}}$. Write $U = U_0 \cdot U_1 \cdots U_{n-1}$ such that for $0 \leq i < n$, U_i is non-sequential (as in Corollary 3.1). The proof proceeds by induction on n . In the base, we have that $n = 0$. In this case, $U = 1$, and thus $U \in \llbracket g \parallel h \rrbracket_{\text{BKA}}$ by Lemma 3.2. Since $g \parallel h = p_{e,f}(g \parallel h) \leq_{\text{BKA}} s_{e,f}(g \parallel h)$, it follows that $U \in \llbracket s_{e,f}(g \parallel h) \rrbracket_{\text{BKA}}$ by Lemma 3.8.

For the inductive step, assume the claim holds for $n-1$. We write $U = U_0 \cdot U'$, with $U' = U_1 \cdot U_2 \cdots U_{n-1}$. Since $U_0 \cdot U' \in \llbracket g \parallel h \rrbracket_{\text{CKA}}$, there exist $W \in \llbracket g \rrbracket_{\text{CKA}}$ and $X \in \llbracket h \rrbracket_{\text{CKA}}$ such that $U_0 \cdot U' \sqsubseteq W \parallel X$. By Lemma 3.6, we find pomsets W_0, W_1, X_0, X_1 such that $W_0 \cdot W_1 \sqsubseteq W$ and $X_0 \cdot X_1 \sqsubseteq X$, as well as $U_0 \sqsubseteq W_0 \parallel X_0$ and $U' \sqsubseteq W_1 \parallel X_1$. By Lemma 4.10, we find $\ell_g, r_g, \ell_h, r_h \in \mathcal{T}$ with $\ell_g \nabla_g r_g$ and $\ell_h \nabla_h r_h$, such that $W_0 \in \llbracket \ell_g \rrbracket_{\text{CKA}}$, $W_1 \in \llbracket r_g \rrbracket_{\text{CKA}}$, $X_0 \in \llbracket \ell_h \rrbracket_{\text{CKA}}$ and $X_1 \in \llbracket r_h \rrbracket_{\text{CKA}}$.

From this, we know that $U_0 \in \llbracket \ell_g \parallel \ell_h \rrbracket_{\text{CKA}}$ and $U' \in \llbracket r_g \parallel r_h \rrbracket_{\text{CKA}}$. Since U_0 is non-sequential, we have that $U_0 \in \llbracket \ell_g \odot \ell_h \rrbracket_{\text{BKA}}$. Moreover, by induction we find that $U' \in \llbracket s_{e,f}(r_g \parallel r_h) \rrbracket_{\text{BKA}}$. Since $\ell_g \odot \ell_h \leq_{\text{BKA}} M_{e,f}(g \parallel h, r_g \parallel r_h)$ by definition of $M_{e,f}$, we furthermore find that

$$(\ell_g \odot \ell_h) \cdot s_{e,f}(r_g \parallel r_h) \leq_{\text{BKA}} M_{e,f}(g \parallel h, r_g \parallel r_h) \cdot s_{e,f}(r_g \parallel r_h)$$

Since $r_g \parallel r_h \in I$, we find by definition of the solution to a linear system that

$$M_{e,f}(g \parallel h, r_g \parallel r_h) \cdot s_{e,f}(r_g \parallel r_h) \leq_{\text{BKA}} s_{e,f}(g \parallel h)$$

By Lemma 3.8 and the above, we conclude that $U = U_0 \cdot U' \in \llbracket s_{e,f}(g \parallel h) \rrbracket_{\text{BKA}}$. \square

For a concrete example where we find a closure of a (non-trivial) parallel composition by solving a linear system, we refer to Appendix A.

With closure of parallel composition, we can construct a closure for any term and therefore conclude completeness of CKA.

Theorem 4.1. *Let $e \in \mathcal{T}$. We can construct a closure e_{\downarrow} of e .*

Proof. The proof proceeds by induction on $|e|$ and the structure of e , i.e., by considering f before g if $|f| < |g|$, or if f is a strict subterm of g (in which case $|f| \leq |g|$ also holds). It is not hard to see that this induces a well-ordering on \mathcal{T} .

Let e be a term of width n , and suppose that the claim holds for all terms of width at most $n - 1$, and for all strict subterms of e . There are three cases.

- If $e = 0$, $e = 1$ or $e = a$ for some $a \in \Sigma$, the claim follows from Lemma 4.2.
- If $e = e_0 + e_1$, or $e = e_0 \cdot e_1$, or $e = e_0^*$, the claim follows from Lemma 4.3.
- If $e = e_0 \parallel e_1$, then $e_0 \otimes e_1$ exists by the induction hypothesis. By Lemma 4.12, we then find that $e_0 \otimes e_1$ is a closure of e . \square

Corollary 4.3. *Let $e, f \in \mathcal{T}$. If $\llbracket e \rrbracket_{\text{CKA}} = \llbracket f \rrbracket_{\text{CKA}}$, then $e \equiv_{\text{CKA}} f$.*

Proof. Follows from Theorem 4.1 and Lemma 4.1. \square

5 Discussion and Further Work

By building a syntactic closure for each series-rational expression, we have shown that the standard axiomatisation of CKA is complete with respect to the CKA-semantics of series-rational terms. Consequently, the algebra of closed series-rational pomset languages forms the free CKA.

Our result leads to several decision procedures for the equational theory of CKA. For instance, one can compute the closure of a term as described in the present paper, and use an existing decision procedure for BKA [3, 12, 20]. Note however that although this approach seems suited for theoretical developments (such as formalising the results in a proof assistant), its complexity makes it less appealing for practical use. More practically, one could leverage recent work by Brunet et al. [3], which provides an algorithm to compare closed series-rational pomset languages. Since this is the free concurrent Kleene algebra, this algorithm can now be used to decide the equational theory of CKA. We also obtain from the latter paper that this decision problem is EXPSpace-complete.

We furthermore note that the algorithm to compute downward closure can be used to extend half of the result from [14] to a Kleene theorem that relates the CKA-semantics of expressions to the pomset automata proposed there: if $e \in \mathcal{T}$, we can construct a pomset automaton A with a state q such that $L_A(q) = \llbracket e \rrbracket_{\text{CKA}}$.

Having established pomset automata as an operational model of CKA, a further question is whether these automata are amenable to a bisimulation-based equivalence algorithm, as is the case for finite automata [10]. If this is the case, optimisations such as those in [2] might have analogues for pomset automata that can be found using the coalgebraic method [23].

While this work was in development, an unpublished draft by Laurence and Struth [19] appeared, with a first proof of completeness for CKA. The general outline of their proof is similar to our own, in that they prove that closure of pomset languages preserves series-rationality, and hence there exists a syntactic closure for every series-rational expression. However, the techniques used to establish this fact are quite different from the developments in the present paper. First, we build the closure via syntactic methods: explicit splitting relations and solutions of linear systems. Instead, their proof uses automata theoretic constructions and algebraic closure properties of regular languages; in particular, they rely on congruences of finite index and language homomorphisms. We believe that our approach leads to a substantially simpler and more transparent proof. Furthermore, even though Laurence and Struth do not seem to use any fundamentally non-constructive argument, their proof does not obviously yield an algorithm to effectively compute the closure of a given term. In contrast, our proof is explicit enough to be implemented directly; we wrote a simple Python script (under six hundred lines) to do just that [16].

A crucial ingredient in this work was the computation of least solutions of linear systems. This kind of construction has been used on several occasions for the study of Kleene algebras [1, 4, 18], and we provide here yet another variation of such a result. We feel that linear systems may not have yet been used to their full potential in this context, and could still lead to interesting developments.

A natural extension of the work conducted here would be to turn our attention to the signature of concurrent Kleene algebra that includes a “parallel star” operator e^{\parallel} . The completeness result of Laurence and Struth [20] holds for BKA with the parallel star, so in principle one could hope to extend our syntactic closure construction to include this operator. Unfortunately, using the results of Laurence and Struth, we can show that this is not possible. They defined a notion of *depth* of a series-parallel pomset, intuitively corresponding to the nesting of parallel and sequential components. An important step in their development consists of proving that for every series-parallel-rational language there exists a finite upper bound on the depth of its elements. However, the language $\llbracket a^{\parallel} \rrbracket_{\text{CKA}}$ does not enjoy this property: it contains every series-parallel pomset exclusively labelled with the symbol a . Since we can build such pomsets with arbitrary depth, it follows that there does not exist a syntactic closure of the term a^{\parallel} . New methods would thus be required to tackle the parallel star operator.

Another aspect of CKA that is not yet developed to the extent of KA is the coalgebraic perspective. We intend to investigate whether the coalgebraic tools developed for KA can be extended to CKA, which will hopefully lead to efficient bisimulation-based decision procedures [2, 5].

Acknowledgements. We thank the anonymous reviewers for their insightful comments. This work was partially supported by the ERC Starting Grant ProFoundNet (grant code 679127).

A Worked Example: A Non-trivial Closure

In this appendix, we solve an instance of a linear system as defined in Definition 4.7 for a given parallel composition. For the sake of brevity, the steps are somewhat coarse-grained; the reader is encouraged to reproduce the steps by hand.

Consider the expression $e \parallel f = a^* \parallel b$. The linear system $\mathfrak{L}_{e,f}$ that we obtain from this expression consists of six inequations; in matrix form (with zeroes omitted), this system is summarised as follows:⁴

$$\begin{array}{l} 1 \parallel 1 \\ 1 \parallel b \\ a \cdot a^* \parallel 1 \\ a^* \parallel 1 \\ a \cdot a^* \parallel b \\ a^* \parallel b \end{array} \left(\begin{array}{ccccccccc} 1 & & & & & & & & 1 \\ b & 1 & & & & & & & b \\ a & & a^* & & a \cdot a^* & & & & a \cdot a^* \\ & & a^* & & a^* \cdot a & & & & a^* \\ a \parallel b & a & a^* \parallel b & a \cdot a^* \parallel b & a^* & a \cdot a^* & a \cdot a^* \parallel b & & \\ b & 1 & a^* \parallel b & a \cdot a^* \parallel b & a^* & a \cdot a^* & a^* \parallel b & & \end{array} \right)$$

Let us proceed under the assumption that x is a solution to the system; the constraint imposed on x by the first two rows is given by the inequations

$$x(1 \parallel 1) + 1 \leq_{\text{CKA}} x(1 \parallel 1) \quad (1)$$

$$b \cdot x(1 \parallel 1) + x(1 \parallel b) + b \leq_{\text{CKA}} x(1 \parallel b) \quad (2)$$

Because these inequations do not involve the other positions of the system, we can solve them in isolation, and use their solutions to find solutions for the remaining positions; it turns out that choosing $x(1 \parallel 1) = 1$ and $x(1 \parallel b) = b$ suffices here.

We carry on to fill these values into the inequations given by the third and fourth row of the linear system. After some simplification, these work out to be

$$a \cdot a^* + a \cdot a^* \cdot x(a^* \parallel 1) + a^* \cdot x(a \cdot a^* \parallel 1) \leq_{\text{CKA}} x(a \cdot a^* \parallel 1) \quad (3)$$

$$a^* + a^* \cdot a \cdot x(a^* \parallel 1) + a^* \cdot x(a \cdot a^* \parallel 1) \leq_{\text{CKA}} x(a^* \parallel 1) \quad (4)$$

Applying the least fixpoint axiom to (3) and simplifying, we obtain

$$a \cdot a^* + a \cdot a^* \cdot x(a^* \parallel 1) \leq_{\text{CKA}} x(a \cdot a^* \parallel 1) \quad (5)$$

Substituting this into (4) and simplifying, we find that

$$a^* + a \cdot a^* \cdot x(a^* \parallel 1) \leq_{\text{CKA}} x(a^* \parallel 1) \quad (6)$$

This inequation, in turn, gives us that $a^* \leq_{\text{CKA}} x(a^* \parallel 1)$ by the least fixpoint axiom. Plugging this back into (3) and simplifying, we find that

$$a \cdot a^* + a^* \cdot x(a \cdot a^* \parallel 1) \leq_{\text{CKA}} x(a \cdot a^* \parallel 1) \quad (7)$$

⁴ Actually, the system obtained from $a^* \parallel b$ as a result of Definition 4.7 is slightly larger; it also contains rows and columns labelled by $1 \cdot a^* \parallel 1$ and $1 \cdot a^* \parallel b$; these turn out to be redundant. We omit these rows from the example for simplicity.

Again by the least fixpoint axiom, this tells us that $a \cdot a^* \leq_{\text{CKA}} x(a \cdot a^* \parallel 1)$. One easily checks that $x(a \cdot a^* \parallel 1) = a \cdot a^*$ and $x(a^* \parallel 1) = a^*$ are solutions to (3) and (4); by the observations above, they are also the least solutions.

It remains to find the least solutions for the final two positions. Filling in the values that we already have, we find the following for the fifth row:

$$\begin{aligned} a \parallel b + a \cdot b + (a^* \parallel b) \cdot a \cdot a^* + (a \cdot a^* \parallel b) \cdot a^* \\ + a^* \cdot x(a \cdot a^* \parallel b) + a \cdot a^* \cdot x(a^* \parallel b) + a \cdot a^* \parallel b \leq_{\text{CKA}} x(a \cdot a^* \parallel b) \end{aligned} \quad (8)$$

Applying the exchange law⁵ to the first three terms, we find that they are contained in $(a \cdot a^* \parallel b) \cdot a^*$, as is the last term; (8) thus simplifies to

$$(a \cdot a^* \parallel b) \cdot a^* + a^* \cdot x(a \cdot a^* \parallel b) + a \cdot a^* \cdot x(a^* \parallel b) \leq_{\text{CKA}} x(a \cdot a^* \parallel b) \quad (9)$$

By the least fixpoint axiom, we find that

$$a^* \cdot (a \cdot a^* \parallel b) \cdot a^* + a \cdot a^* \cdot x(a^* \parallel b) \leq_{\text{CKA}} x(a \cdot a^* \parallel b) \quad (10)$$

For the sixth row, we find that after filling in the solved positions, we have

$$\begin{aligned} b + b + (a^* \parallel b) \cdot a \cdot a^* + (a \cdot a^* \parallel b) \cdot a^* \\ + a^* \cdot x(a \cdot a^* \parallel b) + a \cdot a^* \cdot x(a^* \parallel b) + a^* \parallel b \leq_{\text{CKA}} x(a^* \parallel b) \end{aligned} \quad (11)$$

Simplifying and applying the exchange law as before, it follows that

$$(a^* \parallel b) \cdot a^* + a^* \cdot x(a \cdot a^* \parallel b) + a \cdot a^* \cdot x(a^* \parallel b) \leq_{\text{CKA}} x(a^* \parallel b) \quad (12)$$

We then substitute (10) into (12) to find that

$$(a^* \parallel b) \cdot a^* + a \cdot a^* \cdot x(a^* \parallel b) \leq_{\text{CKA}} x(a^* \parallel b) \quad (13)$$

which, by the least fixpoint axiom, tells us that $a^* \cdot (a^* \parallel b) \cdot a^* \leq_{\text{CKA}} x(a^* \parallel b)$. Plugging the latter back into (9), we find that

$$a^* \cdot (a \cdot a^* \parallel b) \cdot a^* + a \cdot a^* \cdot a^* \cdot (a^* \parallel b) \cdot a^* \leq_{\text{CKA}} x(a \cdot a^* \parallel b) \quad (14)$$

which can, using the exchange law, be reworked into

$$a^* \cdot (a \cdot a^* \parallel b) \cdot a^* \leq_{\text{CKA}} x(a \cdot a^* \parallel b) \quad (15)$$

Now, if we choose $x(a \cdot a^* \parallel b) = a^* \cdot (a \cdot a^* \parallel b) \cdot a^*$ and $x(a^* \parallel b) = a^* \cdot (a^* \parallel b) \cdot a^*$, we find that these choices satisfy (9) and (12)—making them part of a solution; by construction, they are also the least solutions.

In summary, x is a solution to the linear system, and by construction it is also the least solution. The reader is encouraged to verify that our choice of $x(a^* \parallel b)$ is indeed a closure of $a^* \parallel b$.

⁵ A caveat here is that applying the exchange law indiscriminately may lead to a term that is not a closure (specifically, it may violate the semantic requirement in Definition 4.1). The algorithm used to solve arbitrary linear systems in Lemma 3.12 does not make use of the exchange law to simplify terms, and thus avoids this pitfall.

References

1. Backhouse, R.: Closure algorithms and the star-height problem of regular languages. Ph.D. thesis, University of London (1975)
2. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: Proceedings of the Principles of Programming Languages (POPL), pp. 457–468 (2013)
3. Brunet, P., Pous, D., Struth, G.: On decidability of concurrent Kleene algebra. In: Proceedings of the Concurrency Theory (CONCUR), pp. 28:1–28:15 (2017)
4. Conway, J.H.: Regular Algebra and Finite Machines. Chapman and Hall Ltd., London (1971)
5. Foster, N., Kozen, D., Milano, M., Silva, A., Thompson, L.: A coalgebraic decision procedure for NetKAT. In: Proceedings of the Principles of Programming Languages (POPL), pp. 343–355 (2015)
6. Gischer, J.L.: The equational theory of pomsets. *Theor. Comput. Sci.* **61**, 199–224 (1988)
7. Grabowski, J.: On partial languages. *Fundam. Inform.* **4**(2), 427 (1981)
8. Hoare, T., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra. In: Proceedings of the Concurrency Theory (CONCUR), pp. 399–414 (2009)
9. Hoare, T., van Staden, S., Möller, B., Struth, G., Zhu, H.: Developments in concurrent Kleene algebra. *J. Log. Algebr. Meth. Program.* **85**(4), 617–636 (2016)
10. Hopcroft, J.E., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Technical report, TR71-114, December 1971
11. Horn, A., Kroening, D.: On partial order semantics for SAT/SMT-based symbolic encodings of weak memory concurrency. In: Graf, S., Viswanathan, M. (eds.) FORTE 2015. LNCS, vol. 9039, pp. 19–34. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19195-9_2
12. Jategaonkar, L., Meyer, A.R.: Deciding true concurrency equivalences on safe, finite nets. *Theor. Comput. Sci.* **154**(1), 107–143 (1996)
13. Jipsen, P., Moshier, M.A.: Concurrent Kleene algebra with tests and branching automata. *J. Log. Algebr. Methods Program.* **85**(4), 637–652 (2016)
14. Kappé, T., Brunet, P., Luttik, B., Silva, A., Zanasi, F.: Brzozowski goes concurrent—a Kleene theorem for pomset languages. In: Proceedings of the Concurrency Theory (CONCUR), pp. 25:1–25:16 (2017)
15. Kappé, T., Brunet, P., Silva, A., Zanasi, F.: Concurrent Kleene algebra: free model and completeness. <https://arxiv.org/abs/1710.02787>
16. Kappé, T., Brunet, P., Silva, A., Zanasi, F.: Tools for concurrent Kleene algebra, Sep 2017. <https://doi.org/10.5281/zenodo.926823>
17. Kleene, S.C.: Representation of events in nerve nets and finite automata. In: Shannon, C.E., McCarthy, J. (eds.) Automata Studies, pp. 3–41. Princeton University Press, Princeton (1956)
18. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. Comput.* **110**(2), 366–390 (1994)
19. Laurence, M.R., Struth, G.: Completeness theorems for pomset languages and concurrent Kleene algebras. <https://arxiv.org/abs/1705.05896>
20. Laurence, M.R., Struth, G.: Completeness theorems for Bi-Kleene algebras and series-parallel rational pomset languages. In: Höfner, P., Jipsen, P., Kahl, W., Müller, M.E. (eds.) RAMICS 2014. LNCS, vol. 8428, pp. 65–82. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06251-8_5
21. Levi, F.W.: On semigroups. *Bull. Calcutta Math. Soc.* **36**(141–146), 82 (1944)

22. Lodaya, K., Weil, P.: Series-parallel languages and the bounded-width property. *Theor. Comput. Sci.* **237**(1), 347–380 (2000)
23. Rot, J., Bonsangue, M., Rutten, J.: Coalgebraic bisimulation-up-to. In: van Emde Boas, P., Groen, F.C.A., Italiano, G.F., Nawrocki, J., Sack, H. (eds.) *SOFSEM 2013*. LNCS, vol. 7741, pp. 369–381. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35843-2_32

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Concurrency and Distribution



Correctness of a Concurrent Object Collector for Actor Languages

Juliana Franco^{1(✉)}, Sylvan Clebsch², Sophia Drossopoulou¹, Jan Vitek^{3,4},
and Tobias Wrigstad⁵

¹ Imperial College London, London, UK
j.vicente-franco@imperial.ac.uk

² Microsoft Research Cambridge, Cambridge, UK

³ Northeastern University, Boston, USA

⁴ CVUT, Prague, Czech Republic

⁵ Uppsala University, Uppsala, Sweden

Abstract. ORCA is a garbage collection protocol for actor-based programs. Multiple actors may mutate the heap while the collector is running without any dedicated synchronisation. ORCA is applicable to any actor language whose type system prevents data races and which supports causal message delivery. We present a model of ORCA which is parametric to the host language and its type system. We describe the interplay between the host language and the collector. We give invariants preserved by ORCA, and prove its soundness and completeness.

1 Introduction

Actor-based systems are massively parallel programs in which individual actors communicate by exchanging messages. In such systems it is essential to be able to manage data automatically with as little synchronisation as possible. In previous work [9, 12], we introduced the ORCA protocol for garbage collection in actor-based systems. ORCA is language-agnostic, and it allows for concurrent collection of objects in actor-based programs with no additional locking or synchronisation, no copying on message passing and no stop-the-world steps. ORCA can be implemented in any actor-based system or language that has a type system which prevents data races and that supports causal message delivery. There are currently two instantiations of ORCA, one is for Pony [8, 11] and the other for Encore [5]. We hypothesise that ORCA could be applied to other actor-based systems that use static types to enforce isolation [7, 21, 28, 36]. For libraries, such as Akka, which provide actor-like facilities, pluggable type systems could be used to enforce isolation [20].

This paper develops a formal model of ORCA. More specifically, the paper contributions are:

1. Identification of the requirements that the host language must statically guarantee;

2. Description and model of ORCA at a language-agnostic level;
3. Identification of invariants that ensure global consistency without synchronisation;
4. Proofs of *soundness*, *i.e.* live objects will not be collected, and proofs of *completeness*, *i.e.* all garbage will be identified as such.

A formal model facilitates the understanding of how ORCA can be applied to different languages. It also allows us to explore extensions such as shared mutable state across actors [40], reduction of tracing of immutable references [12], or incorporation of borrowing [4]. Alternative implementations of ORCA that rely on deep copying (*e.g.*, to reduce type system complexity) across actors on different machines can also be explored through our formalism.

Developing a formal model of ORCA presents challenges:

Can the model be parametric in the host language? We achieved parametricity by concentrating on the effects rather than the mechanisms of the language. We do not model language features, instead, we model actor behaviour through non-deterministic choice between heap mutation and object creation.

All other actions, such as method call, conditionals, loops etc., are irrelevant.

Can the model be parametric in the host type system? We achieved parametricity by concentrating on the guarantees rather than the mechanism afforded by the type system. We do not define judgments, but instead, assume the existence of judgements which determines whether a path is readable or writeable from a given actor. Through an (uninterpreted) precondition to any heap mutation, we require that no aliasing lets an object writeable from an actor be readable/writable from any other actor.

How to relax atomicity? ORCA relies on a global invariant that relates the number of references to any data object and the number of messages with a path to that object. This invariant only holds if actors execute atomically. Since we desire actors to run in parallel, we developed a more subtle, and weaker, definition of the invariant.

The full proofs and omitted definitions are available in appendix [16].

2 Host Language Requirements

ORCA makes some assumptions about its host language, we describe them here.

2.1 Actors and Objects

Actors are active entities with a thread of control, while objects are data structures. Both actors and objects may have fields and methods. Method calls on objects are synchronous, whereas method calls on actors amount to asynchronous message sends—they all called *behaviours*. Messages are stored in a FIFO queue. When idle, an actor processes the top message from its queue. At any given point of time an actor may be either idle, executing a behaviour, or collecting garbage.

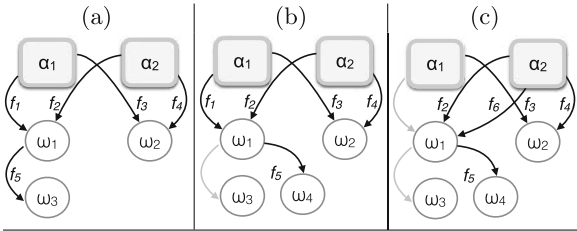


Fig. 1. Actors and objects. Full arrows are references, grey arrows are overwritten references: references that no longer exist.

Actor	Path	Capapability	Actor	Path	Capability
α_1	this.f ₁	write	α_2	this.f ₂	tag
	this.f ₁ .f ₅	write		this.f ₂ .f ₅	\perp
	this.f ₃	read		this.f ₄	read
				this.f ₆	write
				this.f ₆ .f ₅	write

Fig. 2. Capabilities. Heap mutation may modify what object is reachable through a path, but not the path’s capability.

Figure 1 shows actors α_1 and α_2 , objects ω_1 to ω_4 . In [16] we show how to create this object graph in Pony. In Fig. 1(a), actor α_1 points to object ω_1 through field f_1 to ω_2 through field f_3 , and object ω_1 points to ω_3 through field f_5 . In Fig. 1(b), actor α_1 creates ω_4 and assigns it to `this.f1.f5`. In Fig. 1(c), α_1 has given up its reference to ω_1 and sent it to `act2` which stored it in field f_6 . Note that the process of sending sent not only ω_1 but also implicitly ω_4 .

2.2 Mutation, Transfer and Accessibility

Message passing is the only way to share objects. This falls out of the capability system. If an actor shares an object with another actor, then either it gives up the object or neither actor has a write capability to that object. For example, after α_1 sends ω_1 to α_2 , it cannot mutate ω_1 . As a consequence, heap mutation only decreases accessibility, while message sends can transfer accessibility from sender to receiver. When sending immutable data the sender does not need to transfer accessibility. However, when it sends a mutable object it cannot keep the ability to read or to write the object. Thus, upon message send of a mutable object, the actor must consume, or destroy, its reference to that object.

2.3 Capabilities and Accessibility

ORCA assumes that a host language’s type system assigns *access rights* to paths. A path is a sequence of field names. We call these access rights *capabilities*.

We expect the following three capabilities; **read**, **write**, **tag**. The first two allow reading and writing an object’s fields respectively. The **tag** capability only allows

identity comparison and sending the object in a message. The type system must ensure that actors have no read-write races. This is natural for actor languages [5, 7, 11, 21].

Figure 2 shows capabilities assigned to the paths in Fig. 1: $\alpha_1.f_1.f_5$ has capability **write**, thus α_1 can read and write to the object reachable from that path. Note that capabilities assigned to paths are immutable, while the contents of those paths may change. For example, in Fig. 1(a), α_1 can write to ω_3 through path $f_1.f_5$, while in Fig. 1(b) it can write to ω_4 through the same path. In Fig. 1(a) and (b), α_2 can use the address of ω_1 but cannot read or write it, due to the **tag** capability, and therefore cannot access ω_3 (in Fig. 1(a)) nor ω_4 (in Fig. 1(b)). However, in Fig. 1(c) the situation reverses: α_2 , which received ω_1 with **write** capability is now able to reach it through field f_6 , and therefore ω_4 . Notice that the existence of a path from an actor to an object does not imply that the object is accessible to the actor: In Fig. 1(a), there is a path from α_2 to ω_3 , but α_2 cannot access ω_3 . Capabilities protect against data races by ensuring that if an object can be mutated by an actor, then no other actor can access its fields.

2.4 Causality

ORCA uses messages to deliver protocol-related information, it thus requires causal delivery. Messages must be delivered after any and all messages that caused them. Causality is the smallest transitive relation, such that if a message m' is sent by some actor after it received or sent m , then m is a cause of m' . Causal delivery entails that m' be delivered after m .

For example, if actor α_1 sends m_1 to actor α_2 , then sends m_2 to actor α_3 , and α_3 receives m_2 and sends m_3 to α_2 , then m_1 is a cause of m_2 , and m_2 is a cause of m_3 . Causal delivery requires that α_2 receive m_1 before receiving m_3 . No requirements are made on the order of delivery to different actors.

3 Overview of ORCA

We introduce ORCA and discuss how to localise the necessary information to guarantee safe deallocation of objects in the presence of sharing. Every actor has a local heap in which it allocates objects. An actor *owns* the objects it has allocated, and ownership is fixed for an object's life-time, but actors are free to reference objects that they do not own. Actors are obligated to collect their own objects once these are no longer needed. While collecting, an actor must be able to determine whether an object can be deallocated using only local information. This allows all other actors to make progress at any point.

3.1 Mutation and Collection

ORCA relies on capabilities for actors to reference objects owned by other actors and to support concurrent mutation to parts of the heap that are not being concurrently collected. Capabilities avoid the need for barriers.

I₁ An object accessible with write capability from an actor is not accessible with read or write capability from any other actor.

This invariant ensures an actor, while executing garbage collection, can safely trace any object to which it has read or write access without the need to protect against concurrent mutation from other actors.

3.2 Local Collection

An actor can collect its objects based on local information without consulting other actors. For this to be safe, the actor must know that an owned, locally inaccessible, object is also globally inaccessible (*i.e.*, inaccessible from any other actors or messages)¹. Shared objects are reference counted by their owner to ensure:

I₂ An object accessible from a message queue or from a non-owning actor has reference count larger than zero in the owning actor.

Thus, a locally inaccessible object with a reference count of 0 can be collected.

3.3 Messages and Collection

I₁ and **I₂** are sufficient to ensure that local collection is safe. Maintaining **I₂** is not trivial as accessibility is affected by message sends. Moreover, it is possible for an actor to share a **read** object with another actor through a message. What if that actor drops its reference to the object? The object's owner should be informed so it can decrease its reference count. What happens when an actor receives an object in a message? The object's owner should be informed, so that it can increase its reference count. To reduce message traffic, ORCA uses *distributed, weighted, deferred* reference counts. Each actor maintains reference counts that tracks the sharing of its objects. It also maintains counts for “foreign objects”, tracking references to objects owned by other actors. This reference count for non-owning actors is what allows sending/receiving objects without having to inform their owner while maintaining **I₂**. For any object or actor ι , we denote with $LRC(\iota)$ the reference count for ι in ι 's owner, and with $FRC(\iota)$ we denote the sum of the reference counts for ι in all other actors. The counts do not reflect the number of references, rather the existence of references:

I₃ If a non-owning actor can access an object through a path from its fields or call stack, its reference count for this object is greater than 0.

An object is globally accessible if it is accessible from any actor or from a message in some queue. Messages include reference increment or decrement messages—these are ORCA-level messages and they are not visible to applications. We introduce two logical counters: $AMC(\iota)$ to account for the number of application

¹ For example, in Fig. 1(c) ω_4 is locally inaccessible, but globally accessible.

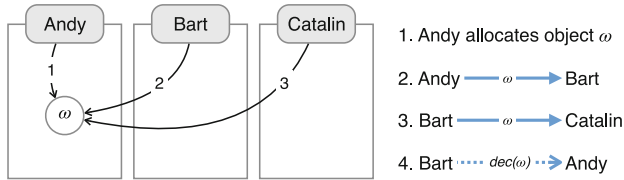


Fig. 3. Black arrows are references, numbered in creation order. Blue solid arrows are application messages and blue dashed arrows ORCA-level message. (Color figure online)

messages with paths to ι , and $OMC(\iota)$ to account for ORCA-level messages with reference count increment and decrement requests. These counters are not present at run-time, but they will be handy for reasoning about ORCA. The owner’s view of an object is described by the LRC and the OMC, while the foreign view is described by the FRC and the AMC. These two views must agree:

$$\mathbf{I}_4 \quad \forall \iota. \text{LRC}(\iota) + \text{OMC}(\iota) = \text{AMC}(\iota) + \text{FRC}(\iota)$$

\mathbf{I}_2 , \mathbf{I}_3 and \mathbf{I}_4 imply that a locally inaccessible object with $\text{LRC} = 0$ can be reclaimed.

3.4 Example

Consider actors Andy, Bart and Catalin, and steps from Fig. 3.

Initial State. Let ω be a newly allocated object. As it is only accessible to its owning actor, Andy, there is no entry for it in any RC.

Sharing ω . When Andy shares ω with Bart, ω is placed on Bart’s message queue, meaning that $\text{AMC}(\omega) = 1$. This is reflected by setting $\text{RC}_{\text{Andy}}(\omega)$ to 1. This preserves \mathbf{I}_4 and the other invariants. When Bart takes the message with ω from his queue, $\text{AMC}(\omega)$ becomes zero, and Bart sets his foreign reference count for ω to 1, that is, $\text{RC}_{\text{Bart}}(\omega) = 1$. When Bart shares ω with Catalin, we get $\text{AMC}(\omega) = 1$. To preserve \mathbf{I}_4 , Bart could set $\text{RC}_{\text{Bart}}(\omega)$ to 0, but this would break \mathbf{I}_3 . Instead, Bart sends an ORCA-level message to Andy, asking him to increment his (local) reference count by some n , and sets his own $\text{RC}_{\text{Bart}}(\omega)$ to n .² This preserves \mathbf{I}_4 and the other invariants. When Catalin receives the message later on, she will behave similarly to Bart in step 2, and set $\text{RC}_{\text{Catalin}}(\omega) = 1$.

The general rule is that when an actor sends one of its objects, it increments the corresponding (local) RC by 1 (reflecting the increasing number of foreign references) but when it sends a non-owned object, it decrements the corresponding (foreign) RC (reflecting a transfer of some of its stake in the object). Special care needs to be taken when the sender’s RC is 1.

² This step can be understood as if Bart “borrowed” n units from Andy, added $n - 1$ to his own RC, and gave 1 to the AMC, to reach Catalin eventually.

Further note that if **Andy**, the owner of ω , received ω , he would decrease his counter for ω rather than increase it, as his reference count denotes foreign references to ω . When an actor receives one of its owned objects, it *decrements* the corresponding (local) RC by 1 but when it receives a non-owned object, it *increments* the corresponding (foreign) RC by 1.

Dropping References to ω . Subsequent to sharing ω with **Catalin**, **Bart** performs GC, and traces his heap without reaching ω (maybe because it did not store ω in a field). This means that **Bart** has given up his stake in ω . This is reflected by sending a message to **Andy** to decrease his RC for ω by n , and setting **Bart**'s RC for ω to 0. **Andy**'s local count of the foreign references to ω are decreased piecemeal like this, until $LRC(\omega)$ reaches zero. At this point, tracing **Andy**'s local heap can determine if ω should be collected.

Further Aspects. We briefly outline further aspects which play a role in ORCA.

Concurrency. Actors execute concurrently. For example, sharing of ω by **Bart** and **Catalin** can happen in parallel. As long as **Bart** and **Catalin** have foreign references to ω , they may separately, and in parallel cause manipulation of the global number of references to ω . These manipulations will be captured locally at each site through FRC, and through increment and decrement messages to **Andy** (OMC).

Causality. Increment and decrement messages may arrive in any order. **Andy**'s queue will serialise them, *i.e.* concurrent asynchronous reference count manipulations will be ordered and executed sequentially. Causality is key here, as it prevents ORCA-level messages to be overtaken by application messages which cause RCs to be decremented; thus causality keeps counters non-negative.

Composite Objects. Objects message must be traced to find the transitive closure of accessible data. For example, when passing ω_1 in a message in Fig. 1(c), objects accessible through it, *e.g.*, ω_4 will be traced. This is mandated by **I₃** and **I₄**.

Finally, we reflect on the nature of reference counts: they are *distributed*, in the sense that an object's owner and every actor referencing it keep separate counts; *weighted*, in that they do not reflect the number of aliases; and *deferred*, in that they are not manipulated immediately on alias creation or destruction, and that non-local increments/decrements are handled asynchronously.

4 The ORCA Protocol

We assume enumerable, disjoint sets *ActorAddr* and *ObjAddr*, for addresses of actors and objects. The union of the two is the set of addresses including *null*. We require a mapping *Class* that gives the name of the class of each actor in a given configuration, and a mapping *O* that returns the owner of an address

$$\begin{aligned} Addr &= ActorAddr \uplus ObjAddr \uplus \{null\} \\ Class: Config \times ActorAddr &\rightarrow ClassId \\ O: Addr &\rightarrow ActorAddr \end{aligned}$$

such that the owner of an actor is the actor itself, *i.e.*, $\forall \alpha \in \text{ActorAddr}. \mathcal{O}(\alpha) = \alpha$.

Definition 1 describes run-time configurations, \mathcal{C} . They consist of a heap, χ , which maps addresses and field identifiers to addresses,³ and an actor map, as , from actor addresses to actors. Actors consist of a frame, a queue, a reference count table, a state, a working set, marks, and a program counter. Frames are either empty, or consist of the identifier for the currently executing behaviour, and a mapping from variables to addresses. Queues are sequences of messages. A message is either an *application message* of the form $\text{app}(\phi)$ denoting a high-level language message with the frame ϕ , or an ORCA message, of the form $\text{orca}(\iota : z)$, denoting an in-flight request for a reference count change for ι by z . The state distinguishes whether the actor is idle, or executing some behaviour, or performing garbage collection. We discuss states, working sets, marks, and program counters in Sect. 4.3. We use naming conventions: $\alpha \in \text{ActorAddr}$; $\omega \in \text{ObjAddr}$; $\iota \in \text{Addr}$; $z \in \mathbb{Z}$; $n \in \mathbb{N}$; $b \in \text{BId}$; $x \in \text{VarId}$; $A \in \text{ClassId}$; and ιs for a sequence of addresses $\iota_1 \dots \iota_n$. We write $\mathcal{C}.\text{heap}$ for \mathcal{C} 's heap; and $\alpha.\text{qu}_{\mathcal{C}}$, or $\alpha.\text{rc}_{\mathcal{C}}$, or $\alpha.\text{frame}_{\mathcal{C}}$, or $\alpha.\text{st}_{\mathcal{C}}$ for the queue, reference count table, frame or state of actor α in configuration \mathcal{C} , respectively.

Definition 1 (Runtime entities and notation)

$$\begin{aligned}
 \mathcal{C} \in \text{Config} &= \text{Heap} \times \text{Actors} \\
 \chi \in \text{Heap} &= (\text{Addr} \setminus \{\text{null}\}) \times \text{FId} \rightarrow \text{Addr} \\
 as \in \text{Actors} &= \text{ActorAddr} \rightarrow \text{Actor} \\
 a \in \text{Actor} &= \text{Frame} \times \text{Queue} \times \text{ReferenceCounts} \\
 &\quad \times \text{State} \times \text{Workset} \times \text{Marks} \times \text{PC} \\
 \phi \in \text{Frame} &= \emptyset \cup (\text{BId} \times \text{LocalMap}) \\
 \psi \in \text{LocalMap} &= \text{VarId} \rightarrow \text{Addr} \\
 q \in \text{Queue} &= \text{Message}^* \\
 m \in \text{Message} &::= \text{orca}(\iota : z) \mid \text{app}(\phi) \\
 rc \in \text{ReferenceCounts} &= \text{Addr} \rightarrow \mathbb{N}
 \end{aligned}$$

State, Workset, Marks, and PC described in Definition 7.

Example: Figure 4 shows \mathcal{C}_0 , our running example for a runtime configuration. It has three actors: α_1 – α_3 , represented by light grey boxes, and eight objects, ω_1 – ω_8 , represented by circles. We show ownership by placing the objects in square boxes, *e.g.* $\mathcal{O}(\omega_7) = \alpha_1$. We show references through arrows, *e.g.* ω_6 references ω_8 through field f_7 , that is, $\mathcal{C}_0.\text{heap}(\omega_6, f_7) = \omega_8$. The frame of α_2 contains behaviour identifier b' , and maps x' to ω_8 . All other frames are empty. The message queue of α_1 contains an application message for behaviour b and argument ω_5 for x , the queue of α_2 is empty, and the queue of α_3 an ORCA message for ω_7 . The bottom part shows reference count tables: $\alpha_1.\text{rc}_{\mathcal{C}_0}(\alpha_1) = 21$,

³ Note that we omitted the class of objects. As our model is parametric with the type system, we can abstract from classes, and simplify our model.

and $\alpha_1.\text{rc}_{C_0}(\omega_7) = 50$. Entries of owned addresses are shaded. Since α_2 owns α_2 and ω_2 , the entries for $\alpha_2.\text{rc}_{C_0}(\alpha_2)$ and $\alpha_2.\text{rc}_{C_0}(\omega_2)$ are shaded. Note that α_1 has a non-zero entry for ω_7 , even though there is no path from α_1 to ω_7 . There is no entry for ω_1 ; no such entry is needed, because no actor except for its owner has a path to it. The 0 values indicate potentially non-existent entries in the corresponding tables; for example, the reference count table for actor α_3 needs only to contain entries for α_1 , α_3 , ω_3 , and ω_4 . Ownership does not restrict access to an address: *e.g.* actor α_1 does not own object ω_3 , yet may access it through the path $\text{this}.f_1.f_2.f_3$, may read its field through $\text{this}.f_1.f_2.f_3.f_4$, and may mutate it, *e.g.* by $\text{this}.f_1.f_2.f_3 = \text{this}.f_1$.

Lookup of fields in a configuration is defined in the obvious way, *i.e.*

Definition 2. $\mathcal{C}(\iota.f) \equiv \mathcal{C}.\text{heap}(\iota, f)$, and $\mathcal{C}(\iota.\bar{f}.f') \equiv \mathcal{C}.\text{heap}(\mathcal{C}(\iota.\bar{f}), f')$

4.1 Capabilities and Accessibility

ORCA considers three capabilities:

$$\kappa \in \text{Capability} = \{\text{read}, \text{write}, \text{tag}\},$$

where **read** allows reading, **write** allows reading and writing, and **tag** forbids both read and write, but allows the use of an object's address. To describe the capability at which objects are visible from actors we use the concepts of *static* and *dynamic paths*.

Static paths consist of the keyword **this** (indicating a path starting at the current actor), or the name of a behaviour, b , and a variable, x , (indicating a path starting at local variable x from a frame of b), followed by any number of fields, f .

$$sp ::= \text{this} \mid b.x \mid sp.f$$

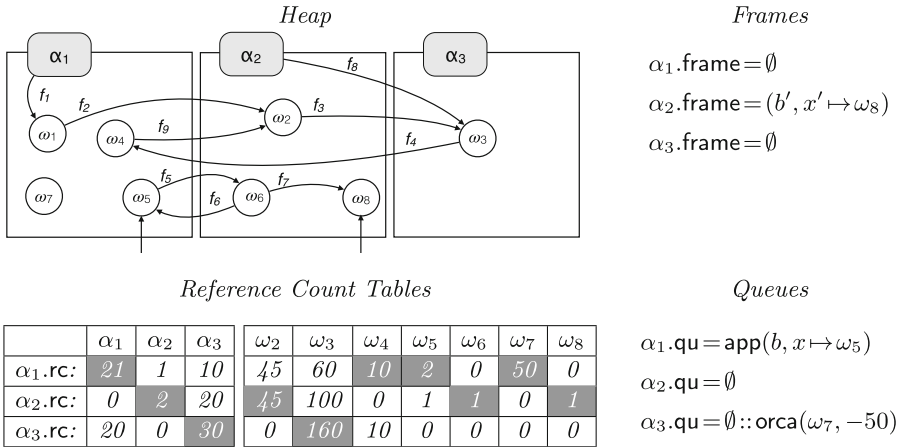


Fig. 4. Configuration C_0 . ω_1 is absent in the ref. counts, it has not been shared.

The host language must assign these capabilities to static paths. Thus, we assume it provides a static judgement of the form

$$A \vdash sp : \kappa \quad \text{where } A \in \text{ClassId}$$

meaning that a static path sp has capability $capability$ when “seen” from a class A . We highlight static judgments, *i.e.*, those provided by the type system in blue.

We expect the type system to guarantee that read and write access rights are “deep”, meaning that all paths to a read capability must go through other read or write capabilities (**A1**), and all paths to a write capability must go through write capabilities (**A2**).

Axiom 1 For class identifier A , static path sp , field f , capability κ , we assume:

- A1** $A \vdash sp.f : \kappa \quad \longrightarrow \quad \exists \kappa' \neq \text{tag}. A \vdash sp : \kappa'.$
A2 $A \vdash sp.f : \text{write} \quad \longrightarrow \quad A \vdash sp : \text{write}.$

Such requirements are satisfied by many type systems with read-only references or immutability (*e.g.* [7, 11, 18, 23, 29, 33, 37, 41]). An implication of **A1** and **A2** is that capabilities degrade with growing paths, *i.e.*, the prefix of a path has more rights than its extensions. More precisely: $A \vdash sp : \kappa$ and $A \vdash sp.f : \kappa'$ imply that $\kappa \leq \kappa'$, where we define $\text{write} < \text{read} < \text{tag}$, and $\kappa \leq \kappa'$ *iff* $\kappa = \kappa'$ or $\kappa < \kappa'$.

Example: Table 1 shows capabilities for some paths from Fig. 4. Thus, $A_1 \vdash \text{this}.f_1 : \text{write}$, and $A_2 \vdash b'.x' : \text{write}$, and $A_2 \vdash \text{this}.f_8 : \text{tag}$. The latter, together with **A1** gives that $A_2 \not\vdash \text{this}.f_8.f : \kappa$ for all κ and f .

As we shall see later, the existence of a path does not imply that the path may be navigated. For example, $\mathcal{C}_0(\alpha_2.f_8.f_4) = \omega_4$, but actor α_2 cannot access ω_4 because of $A_2 \vdash \text{this}.f_8 : \text{tag}$.

Moreover, it is possible for a path to have a capability, while not being defined. For example, Table 1 shows $A_1 \vdash \text{this}.f_1.f_2 : \text{write}$ and it would be possible to have $\mathcal{C}_i(\alpha_1.f_1) = \text{null}$, for some configuration \mathcal{C}_i that derives from \mathcal{C}_0 .

Table 1. Capabilities for paths, where $A_1 = \text{Class}(\alpha_1)$ and $A_2 = \text{Class}(\alpha_2)$.

ClassId	Path	Capability
A ₁	this.f ₁	write
	this.f ₁ .f ₂	write
	this.f ₁ .f ₂ .f ₃	write
	this.f ₁ .f ₂ .f ₃ .f ₄	tag
	b.x	write
	b.x.f ₅	write
	b.x.f ₅ .f ₇	tag
	b.x.f ₅ .f ₆	write
ClassId	Path	Capability
A ₂	this.f ₈	tag
	b'.x'	write

Dynamic paths (in short paths p) start at the actor's fields, or frame, or at some pending message in an actor's queue (the latter cannot be navigated yet, but will be able to be navigated later on when the message is taken off the queue). Dynamic paths may be local paths (lp) or message paths. Local paths consist of this or a variable x followed by any number of fields f . In such paths, this is the current actor, and x is a local variable from the current frame. Message paths consist of $k.x$ followed by a sequence of fields. If $k \geq 0$, then $k.x$ indicates the local variable x from the k -th message from the queue; $k = -1$ indicates variables from either (a) a message that has been popped from the queue, but whose frame has not yet been pushed onto the stack, or (b) a message whose frame has been created but not yet been pushed onto the queue. Thus, $k = -1$ indicates that either (a) a frame will be pushed onto the stack, during message receiving, or (b) a message will be pushed onto the queue during message sending.

$$p \in \text{Path} ::= lp \mid mp \qquad lp ::= \text{this} \mid x \mid lp.f \qquad mp ::= k.x \mid mp.f$$

We define accessibility as the lookup of a path provided that the capability for this path is defined. The *partial* function \mathcal{A} returns a pair: the address accessible from actor α following path p , and the capability of α on p . A path of the form $p.\text{owner}$ returns the owner of the object accessible though p and capability tag .

Definition 3 (accessibility). *The partial function*

$$\mathcal{A} : \text{Config} \times \text{ActorAddr} \times \text{Path} \rightarrow (\text{Addr} \times \text{Capability})$$

is defined as

$$\begin{aligned} \mathcal{A}_C(\alpha, \text{this}.\bar{f}) &= (\iota, \kappa) & \text{iff } \mathcal{C}(\alpha, \bar{f}) &= \iota \wedge \text{Class}(\alpha) \vdash \text{this}.\bar{f} : \kappa \\ \mathcal{A}_C(\alpha, x.\bar{f}) &= (\iota, \kappa) & \text{iff } \exists b.\psi. [\alpha.\text{frame}_C = (b, \psi) \wedge \mathcal{C}(\psi(x), \bar{f}) = \iota \\ & & \wedge \text{Class}(\alpha) \vdash b.x.\bar{f} : \kappa] \\ \mathcal{A}_C(\alpha, k.x.\bar{f}) &= (\iota, \kappa) & \text{iff } k \geq 0 \wedge \exists b.\psi. [\alpha.\text{qu}_C[k] = \text{app}(b, \psi) \wedge \\ & & \mathcal{C}(\psi(x), \bar{f}) = \iota \wedge \text{Class}(\alpha) \vdash b.x.\bar{f} : \kappa] \\ \mathcal{A}_C(\alpha, -1.x.\bar{f}) &= (\iota, \kappa) & \text{iff } \alpha \text{ is executing Sending or Receiving, and ...} \\ & & \text{continued in Definition 9.} \\ \mathcal{A}_C(\alpha, p.\text{owner}) &= (\alpha', \text{tag}) & \text{iff } \exists \iota. [\mathcal{A}_C(\alpha, p) = (\iota, -) \wedge \mathcal{O}(\iota) = \alpha'] \end{aligned}$$

We use $\mathcal{A}_C(\alpha, p) = \iota$ as shorthand for $\exists \kappa. \mathcal{A}_C(\alpha, p) = (\iota, \kappa)$. The second and third case above ensure that the capability of a message path is the same as when the message has been taken off the queue and placed on the frame.

Example: We obtain that $\mathcal{A}_{C_0}(\alpha_1, \text{this}.f_1.f_2.f_3) = (\omega_3, \text{write})$, from the fact that Fig. 4 says that $\mathcal{C}_0(\alpha_1, f_1.f_2.f_3) = \omega_3$ and from the fact that Table 1 says that $A_1 \vdash \text{this}.f_1.f_2.f_3 : \text{write}$. Similarly, $\mathcal{A}_{C_0}(\alpha_2, \text{this}.f_8) = (\omega_3, \text{tag})$, and $\mathcal{A}_{C_0}(\alpha_2, x') = (\omega_8, \text{write})$, and $\mathcal{A}_{C_0}(\alpha_1, 0.x.f_5.f_7) = (\omega_8, \text{tag})$.

Both $\mathcal{A}_{C_0}(\alpha_1, \text{this}.f_1.f_2.f_3)$, and $\mathcal{A}_{C_0}(\alpha_2, \text{this}.f_8)$ describe paths from actors' fields, while $\mathcal{A}_{C_0}(\alpha_2, x')$ describes a path from the actor's frame, and finally $\mathcal{A}_{C_0}(\alpha_1, 0.x.f_5.f_7)$ is a path from the message queue.

Accessibility describes what may be read or written to: $\mathcal{A}_{C_0}(\alpha_1, \text{this}.f_1.f_2.f_3) = (\omega_3, \text{write})$, therefore actor α_1 may mutate object ω_3 . However, this mutation is not

visible by α_2 , even though $\mathcal{C}_0(\alpha_2.f_8) = \omega_3$, because $\mathcal{A}_{\mathcal{C}_0}(\alpha_2, \text{this}.f_8) = (\omega_3, \text{tag})$, which means that actor α_2 has only opaque access to ω_3 .

Accessibility plays a role in collection: If the reference f_3 were to be dropped it would be safe to collect ω_4 ; even though there exists a path from α_2 to ω_4 ; object ω_4 is not accessible to α_2 : the path $\text{this}.f_8.f_4$ leads to ω_4 but will never be navigated ($\mathcal{A}_{\mathcal{C}_0}(\alpha_2, \text{this}.f_8.f_4)$ is undefined). Also, $\mathcal{A}_{\mathcal{C}}(\alpha_2, \text{this}.f_8.\text{owner}) = (\alpha_3, \text{tag})$; thus, as long as ω_4 is accessible from some actor, *e.g.* through $\mathcal{C}(\alpha_2.f_8) = \omega_4$, actor α_3 will not be collected.

Because the class of an actor as well as the capability attached to a static path are constant throughout program execution, the capabilities of paths starting from an actor's fields or from the same frame are also constant.

Lemma 1. *For actor α , fields \bar{f} , behaviour b , variable x , fields \bar{f} , capabilities κ, κ' , configurations \mathcal{C} and \mathcal{C}' , such that \mathcal{C} reduces to \mathcal{C}' in one or more steps:*

$$\begin{aligned} & - \mathcal{A}_{\mathcal{C}}(\alpha, \text{this}.\bar{f}) = (\iota, \kappa) \quad \wedge \quad \mathcal{A}_{\mathcal{C}'}(\alpha, \text{this}.\bar{f}) = (\iota', \kappa') \quad \longrightarrow \quad \kappa = \kappa' \\ & - \mathcal{A}_{\mathcal{C}}(\alpha, x.\bar{f}) = (\iota, \kappa) \quad \wedge \quad \mathcal{A}_{\mathcal{C}'}(\alpha, x.\bar{f}) = (\iota', \kappa') \quad \wedge \\ & \quad \alpha.\text{frame}_{\mathcal{C}} = (b, _) \quad \wedge \quad \alpha.\text{frame}_{\mathcal{C}'} = (b, _) \quad \longrightarrow \quad \kappa = \kappa' \end{aligned}$$

4.2 Well-Formed Configurations

We characterise data-race free configurations ($\models \mathcal{C} \diamond$):

Definition 4 (Data-race freedom). $\models \mathcal{C} \diamond$ *iff*

$\forall \alpha, \alpha', p, p', \kappa, \kappa'.$

$$\alpha \neq \alpha' \quad \wedge \quad \mathcal{A}_{\mathcal{C}}(\alpha, p) = (\iota, \kappa) \quad \wedge \quad \mathcal{A}_{\mathcal{C}}(\alpha', p') = (\iota, \kappa')$$

\longrightarrow

$$\kappa \sim \kappa'$$

where we define

$$\kappa \sim \kappa' \quad \text{iff} \quad [(\kappa = \text{write} \longrightarrow \kappa' = \text{tag}) \quad \wedge \quad (\kappa' = \text{write} \longrightarrow \kappa = \text{tag})]$$

This definition captures invariant **I₁**. The remaining invariants depend on the four derived counters introduced in Sect. 3. Here we define LRC and FRC, and give a preliminary definition of AMC and OMC.

Definition 5 (Derived counters—preliminary for AMC and_{ss} OMC)

$$\text{LRC}_{\mathcal{C}}(\iota) \equiv \mathcal{O}(\iota).\text{rc}_{\mathcal{C}}(\iota)$$

$$\text{FRC}_{\mathcal{C}}(\iota) \equiv \sum_{\alpha \neq \mathcal{O}(\iota)} \alpha.\text{rc}_{\mathcal{C}}(\iota)$$

$$\text{OMC}_{\mathcal{C}}(\iota) \equiv \sum_j \begin{cases} z & \text{if } \mathcal{O}(\iota).\text{qu}_{\mathcal{C}}[j] = \text{orca}(\iota : z) \\ 0 & \text{otherwise} \end{cases} + \dots \text{c.f. Definition 12}$$

$$\text{AMC}_{\mathcal{C}}(\iota) \equiv \# \{ (\alpha, k) \mid k > 0 \wedge \exists x.\bar{f}.\mathcal{A}_{\mathcal{C}}(\alpha, k.x.\bar{f}) = \iota \} + \dots \text{c.f. Definition 12}$$

where $\#$ denotes cardinality.

For the time being, we will be reading this preliminary definition as if ... stood for 0. This works under the assumption the procedures are atomic. However Sect. 5.3, when we consider fine-grained concurrency, will refine the definition of AMC and OMC so as to also consider whether an actor is currently in the process of sending or receiving a message from which the address is accessible. For the time being, we continue with the preliminary reading.

Example: Assuming that in \mathcal{C}_0 none of the actors is sending or receiving, we have $\text{LRC}_{\mathcal{C}_0}(\omega_3) = 160$, and $\text{FRC}_{\mathcal{C}_0}(\omega_3) = 160$, and $\text{OMC}_{\mathcal{C}_0}(\omega_3) = 0$, and $\text{AMC}_{\mathcal{C}_0}(\omega_3) = 0$. Moreover, $\text{AMC}_{\mathcal{C}_0}(\omega_6) = \text{AMC}_{\mathcal{C}_0}(\alpha_2) = 1$: neither ω_6 nor α_2 are arguments in application messages, but they are indirectly reachable through the first message on α_1 's queue.

A well-formed configuration requires: **I₁–I₄**: introduced in Sect. 3; **I₅**: the RC's are non-negative; **I₆**: accessible paths are not dangling; **I₇**: processing message queues will not turn RC's negative; **I₈**: actors' contents is in accordance with their state. The latter two will be described in Definition 14.

Definition 6 (Well-formed configurations—preliminary). $\models \mathcal{C}$, iff for all $\alpha, \alpha_o, \iota, \iota', p, lp$, and mp , such that $\alpha_o = \mathcal{O}(\iota) \neq \alpha$:

- I₁** $\models \mathcal{C} \diamond$
- I₂** $[\mathcal{A}_\mathcal{C}(\alpha, p) = \iota \vee \mathcal{A}_\mathcal{C}(\alpha_o, mp) = \iota] \longrightarrow \text{LRC}_\mathcal{C}(\iota) > 0$
- I₃** $\mathcal{A}_\mathcal{C}(\alpha, lp) = \iota \longrightarrow \alpha.\text{rc}_\mathcal{C}(\iota) > 0$
- I₄** $\text{LRC}_\mathcal{C}(\iota) + \text{OMC}_\mathcal{C}(\iota) = \text{FRC}_\mathcal{C}(\iota) + \text{AMC}_\mathcal{C}(\iota)$
- I₅** $\alpha.\text{rc}_\mathcal{C}(\iota') \geq 0$
- I₆** $\mathcal{A}_\mathcal{C}(\alpha, p) = \iota \longrightarrow \mathcal{C}.\text{heap}(\iota) \neq \perp$
- I₇, I₈** *description in Definition 14.*

For ease of notation, we take **I₅** to mean that if $\alpha.\text{rc}_\mathcal{C}(\iota')$ is defined, then it is positive. And we take any undefined entry of $\alpha.\text{rc}_\mathcal{C}(\iota)$ to be 0.

4.3 Actor States

We now complete the definition of runtime entities (Definition 1), and describe the states of an actor, the worksets, the marks, and program counters. (Definition 7). We distinguish the following states: idle (IDLE), collecting (COLLECT), receiving (RECEIVE), sending a message (SEND), or executing the synchronous part of a behaviour (EXECUTE). We discuss these states in more detail next.

Except for the idle state, IDLE, all states use auxiliary data structures: *worksets*, denoted by **ws**, which stores a set of addresses; *marks* maps, denoted by **ms**, from addresses to R (reachable) or U (unreachable), and program counters. Frames are relevant when in states EXECUTE, or SEND, and otherwise are assumed to be empty. Worksets are used to store all addresses traced from a message or from the actor itself, and are relevant when in states SEND, or RECEIVE, or COLLECT, and otherwise are empty. Marks are used to calculate reachability and are used in state COLLECT, and are ignored otherwise. The program counters record the instruction an actor will execute next; they range between 4 and 27 and are ghost state, *i.e.* only used in the proofs.

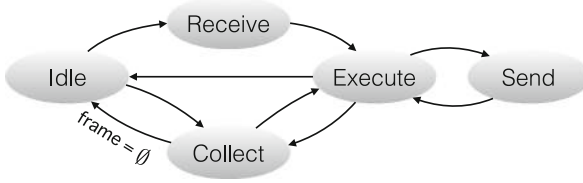


Fig. 5. State transitions diagram for an actor.

Definition 7 (Actor States, Working sets, and Marks)

$$\begin{aligned}
 \text{st} \in \text{State} &::= \text{IDLE} \mid \text{EXECUTE} \mid \text{SEND} \mid \text{RECEIVE} \mid \text{COLLECT} \\
 \text{ws} \in \text{Workset} &= \mathcal{P}(\text{Addr}) \\
 \text{ms} \in \text{Marks} &= \text{Addr} \rightarrow \{\text{R}, \text{U}\} \\
 \text{pc} \in \text{PC} &= [4..27]
 \end{aligned}$$

We write $\alpha.\text{st}_{\mathcal{C}}$, or $\alpha.\text{ws}_{\mathcal{C}}$, or $\alpha.\text{ms}_{\mathcal{C}}$, or $\alpha.\text{pc}_{\mathcal{C}}$ for the state, working set, marks, or the program counter of α in \mathcal{C} , respectively.

Actors may transition between states. The state transitions are depicted in Fig. 5. For example, an actor in the idle state (IDLE) may receive an **orca** message (remaining in the same state), receive an **app** message (moving to the RECEIVE state), or start garbage collection (moving to the COLLECT state).

In the following sections we describe the actions an actor may perform. Following the style of [17,26,27] we describe actors’ actions through pseudo-code procedures, which have the form:

procedure_name $\langle\alpha\rangle$:
 $\xrightarrow{\text{condition}}$
 { instructions }

We let α denote the executing actor, and the left-hand side of the arrow describes the **condition** that must be satisfied in order to execute the **instructions** on the arrow’s right-hand side. Any actor may execute concurrently with other actors. To simplify notation, we assume an implicit, globally accessible configuration \mathcal{C} . Thus, instruction $\alpha.\text{state} := \text{EXECUTE}$ is short for updating the state of α in \mathcal{C} to be EXECUTE. We elide configurations when obvious, *e.g.* $\alpha.\text{frame} = \phi$ is short for requiring that in \mathcal{C} the frame of α is ϕ , but we mention them when necessary—*e.g.* $\models \mathcal{C}[\iota_1, f \mapsto \iota_2] \Diamond$ expresses that the configuration that results from updating field f in ι_1 is data-race free.

Tracing Function. Both garbage collection, and application message sending/receiving need to find all objects accessible from the current actor and/or from the message arguments. We define two functions: **trace.this** finds all addresses which are accessible from the current actor, and **trace.frame** finds all addresses which are accessible through a stack frame (but not from the current actor, this).

```

1  GarbageCollection( $\alpha$ ):
2     $\alpha.st = \text{IDLE} \vee \alpha.st = \text{EXECUTE}$ 
3     $\rightarrow$ 
4    {
5       $\alpha.st := \text{COLLECT}$ 
6       $\alpha.ms := \emptyset$ 
7
8      // marking as unreachable
9      forall  $\iota$  with  $\alpha = \mathcal{O}(\iota) \vee \alpha.rc(\iota) > 0$  do  $\alpha.ms := \alpha.ms[\iota \mapsto \text{U}]$ 
10
11     // tracing and marking locally accessible as reachable
12     forall  $\iota \in \text{trace\_this}(\alpha) \cup \text{trace\_frame}(\alpha.frame)$  do  $\alpha.ms := \alpha.ms[\iota \mapsto \text{R}]$ 
13
14     // marking owned and globally accessible as reachable
15     forall  $\iota$  with  $\alpha = \mathcal{O}(\iota) \wedge \alpha.rc(\iota) > 0$  do  $\alpha.ms := \alpha.ms[\iota \mapsto \text{R}]$ 
16
17     // collecting
18     forall  $\iota$  with  $\alpha.ms(\iota) = \text{U}$  do
19       if  $\mathcal{O}(\iota) = \alpha$  then
20          $\mathcal{C}.heap := \mathcal{C}.heap[\iota \mapsto \perp]$ 
21          $\alpha.rc := \alpha.rc[\iota \mapsto \perp]$ 
22       else
23          $\mathcal{O}(\iota).qu.push(\text{orca}(\iota; -\alpha.rc(\iota)))$ 
24          $\alpha.rc := \alpha.rc[\iota \mapsto \perp]$ 
25
26     if  $\alpha.frame = \emptyset$  then  $\alpha.st := \text{IDLE}$  else  $\alpha.st := \text{EXECUTE}$ 
27   }
```

Fig. 6. Pseudo-code for garbage collection.

Definition 8 (Tracing). We define the functions

$\text{trace_this} : \text{Config} \times \text{ActorAddr} \rightarrow \mathcal{P}(\text{Addr})$

$\text{trace_frame} : \text{Config} \times \text{ActorAddr} \times \text{Frame} \rightarrow \mathcal{P}(\text{Addr})$

as follows

$\text{trace_this}_C(\alpha) \equiv \{\iota \mid \exists \bar{f}. \mathcal{A}_C(\alpha, \text{this}, \bar{f}) = \iota\}$

$\text{trace_frame}_C(\alpha, \phi) \equiv \{\iota \mid \exists x \in \text{dom}(\phi), \bar{f}. \mathcal{A}_C(\alpha, x, \bar{f}) = \iota\}$

4.4 Garbage Collection

We describe garbage collection in Fig. 6. An idle, or an executing actor (pre-condition on line 2) may start collecting at any time. Then, it sets its state to COLLECT (line 5), and initialises the marks, ms , to empty (line 6).

The main idea of ORCA collection is that the requirement for global unreachability of owned objects can be weakened to the local requirement to local unreachability and a $LRC = 0$. Therefore, the actor marks all owned objects, and all addresses with a $RC > 0$ as U (line 9). After that, it traces the actor's fields, and also the actor's frame if it happens not to be empty (as we shall see later, idle actors have empty frames) and marks all accessible addresses as R (line 12). Then, the actor marks all owned objects with $RC > 0$ as R (line 15). Thus we expect that: (*) *Any ι with $ms(\iota) = \text{U}$ is locally unreachable, and if owned by the current actor, then its LRC is 0.* For each address with $ms(\iota) = \text{U}$, if the actor

owns ι , then it collects it (line 20)—this is sound because of \mathbf{I}_2 , \mathbf{I}_3 , \mathbf{I}_4 and $(*)$. If the actor does not own ι , then it asks ι 's owner to decrement its reference count by the current actor's reference count, and deletes its own reference count to it (thus becoming 0) (line 24)—this preserves \mathbf{I}_2 , \mathbf{I}_3 and \mathbf{I}_4 .

There is no need for special provision for cycles across actor boundaries. Rather, the corresponding objects will be collected by each actor separately, when it is the particular actor's turn to perform GC.

Example: Look at the cycle ω_5 – ω_6 , and assume that the message $\text{app}(b, \omega_5)$ had finished execution without any heap mutation, and that $\alpha_1.\text{rc}_C(\omega_5) = \alpha_1.\text{rc}_C(\omega_6) = 1 = \alpha_2.\text{rc}_C(\omega_5) = \alpha_2.\text{rc}_C(\omega_6)$ —this will be the outcome of the example in Sect. 4.5. Now, the objects ω_5 and ω_6 are globally unreachable. Assume that α_1 performs GC: it will *not* be able to collect any of these objects, but it will send a $\text{orca}(\omega_6 : -1)$ to α_2 . Some time later, α_2 will pop this message, and some time later it will enter a GC cycle: it will collect ω_6 , and send a $\text{orca}(\omega_5 : -1)$ to α_1 . When, later on, α_1 pops this message, and later enters a GC cycle, it will collect ω_5 .

At the end of the GC cycle, the actor sets its state back to what it was before (line 26). If the frame is empty, then the actor had been IDLE, otherwise it had been in state EXECUTE.

4.5 Receiving and Sending Messages

Through message send or receive, actors share addresses with other actors. This changes accessibility. Therefore, action is needed to re-establish \mathbf{I}_3 and \mathbf{I}_4 for all the objects accessible from the message's arguments.

Receiving application messages is described by **Receiving** in Fig. 7. It requires that the actor α is in the IDLE state and has an application message on top of its queue. The actor sets its state to RECEIVE (line 5), traces from the message arguments and stores all accessible addresses into ws (line 7). Since accessibility is not affected by other actors' actions, *c.f.*, *last paragraph in Sect. 4.6* it is legitimate to consider the calculation of trace_frame as one single step. It then pops the message from its queue (line 8), and thus the AMC for all the addresses in ws will decrease by 1. To preserve \mathbf{I}_4 , for each ι in its ws , the actor:

- if it is ι 's owner, then it *decrements* its reference count for ι by 1, thus decreasing $\text{LRC}_C(\iota)$ (line 12).
- if it is *not* ι 's owner, then it *increments* its reference count for ι by 1, thus increasing $\text{FRC}_C(\iota)$ (line 14).

After that, the actor sets its frame to that from the message (line 17), and goes to the EXECUTE state (line 18).

Example: Actor α_1 has an application message in its queue. Assuming that it is IDLE, it may execute **Receiving**: It will trace ω_5 and as a result store

```

1 Receiving( $\alpha$ ):
2    $\alpha.st = \text{IDLE} \wedge \alpha.qu.top() = \text{app}(\phi)$ 
3    $\rightarrow$ 
4   {
5      $\alpha.st := \text{RECEIVE}$ 
6
7      $\alpha.ws := \text{trace\_frame}(\alpha, \phi)$ 
8      $\text{pop}(\alpha.qu)$ 
9
10    foreach  $\iota \in \alpha.ws$  do
11      if  $\alpha = \mathcal{O}(\iota)$  then
12         $\alpha.rc(\iota) -= 1$ 
13      else
14         $\alpha.rc(\iota) += 1$ ;
15         $\alpha.ws := \alpha.ws \setminus \{\iota\}$ 
16
17     $\alpha.frame := \phi$ 
18     $\alpha.st := \text{EXECUTE}$ 
19  }

1 ReceiveORCA( $\alpha$ ):
2    $\alpha.state = \text{IDLE} \wedge \alpha.qu.top() = \text{ORCA}(\iota : z)$ 
3    $\rightarrow$ 
4   {
5      $\alpha.rc(\iota) += z$ 
6      $\alpha.qu.pop()$ 
7   }

```

Fig. 7. Receiving application and ORCA messages.

$\{\omega_5, \omega_6, \omega_8, \alpha_1, \alpha_2\}$ in its **ws**. It will then decrement its reference count for ω_5 and α_1 (the owned addresses) and increment it for the others. It will then pop the message from its queue, create the appropriate frame, and go to state **EXECUTE**.

Receiving ORCA messages is described in Fig. 7. An actor in the **IDLE** state with an **ORCA** message at the top, pops the message from its queue, and adds the value z to the reference count for ι , and stays in the **IDLE** state.

Sending application messages is described in Fig. 8. The actor must be in the **EXECUTE** state for some behaviour b and must have local variables which can be split into ψ and ψ' —the latter will form part of the message to be sent. As the **AMC** for all the addresses reachable through the message increases by 1, in order to preserve **I₄** for each address ι in **ws**, the actor:

- increments its reference count for ι by 1, if it owns it (line 14);
- decrements its reference count for ι if it does not own it (line 16). But special care is needed if the actor’s (foreign) reference count for ι is 1, because then a simple decrement would break **I₅**. Instead, the actor set its reference count for ι by 256 (line 18) and sends an **ORCA** message to ι ’s owner with 256 as argument.

After this, it removes ψ' from its frame (line 22), pushes the message $\text{app}(b', \psi')$ onto α' ’s queue, and transitions to the **EXECUTE** state.


```

1  Sending( $\alpha$ ):
2   $\alpha.st = EXECUTE \wedge \alpha.frame = (b, \psi \cdot \psi') \wedge$ 
3   $\forall x \in dom(\psi), x' \in dom(\psi'). \forall \kappa, \kappa'. \forall \bar{f}, \bar{f}'. [$ 
4     $[ \mathcal{A}_C(\alpha, x.\bar{f}) = (\iota, \kappa) \wedge \mathcal{A}_C(\alpha, x'.\bar{f}') = (\iota, \kappa') \longrightarrow \kappa' \sim \kappa ] \wedge$ 
5     $[ Class(\alpha) \vdash b.x'.\bar{f}' : \kappa' \longleftrightarrow Class(\alpha') \vdash b'.x'.\bar{f}' : \kappa' ]$ 
6   $\longrightarrow$ 
7  {
8     $\alpha.st := SEND$ 
9
10    $\alpha.ws := trace\_frame(\alpha, (b, \psi'))$ 
11
12   foreach  $\iota \in \alpha.ws$  do
13     if  $\alpha = \mathcal{O}(\iota)$  then
14        $\alpha.rc(\iota) += 1$ 
15     elseif  $\alpha.rc(\iota) > 1$  then
16        $\alpha.rc(\iota) -= 1$ 
17     else
18        $\alpha.rc(\iota) := 256$ 
19        $\mathcal{O}(\iota).qu.push(orca(\iota : 256))$ 
20        $\alpha.ws := \alpha.ws \setminus \{\iota\}$ 
21
22    $\alpha.frame := (b, \psi)$ 
23    $\alpha'.qu.push(app(b', \psi'))$ 
24
25    $\alpha.st := EXECUTE$ 
26 }
```

Fig. 8. Pseudo-code for message sending.

We now discuss the preconditions. These ensure that sending the message $app(b, \psi')$ will not introduce data races: Line 4 ensures that there are no data races between paths starting at ψ and paths starting at ψ' , while Line 5 ensures that the sender, α , and the receiver, α' see all the paths sent, *i.e.* those starting from (b', ψ') , at the same capability. We express our expectation that the source language compiler produces code only if it satisfies this property by adding this static requirement as a precondition. These static requirements imply that after the message has been sent, there will be no races between paths starting at the sender's frame and those starting at the last message in the receiver's queue. In more detail, after the sender's frame has been reduced to (b, ψ) , and $app(b', \psi')$ has been added to the receiver's queue (at location k), we will have a new configuration $\mathcal{C}' = \mathcal{C}[\alpha, frame \mapsto (b, \psi)][\alpha', queue \mapsto \alpha'.queue_{\mathcal{C}} :: (b', \psi')]$. In this new configuration lines 4 and 5 ensure that $\mathcal{A}_{\mathcal{C}'}(\alpha, x.\bar{f}) = (\iota, \kappa) \wedge \mathcal{A}_{\mathcal{C}'}(\alpha', k.x'.\bar{f}') = (\iota, \kappa') \longrightarrow \kappa' \sim \kappa$, which means that if there were no data races in \mathcal{C} , there will be no data races in \mathcal{C}' either. Formally: $\models \mathcal{C} \diamond \longrightarrow \models \mathcal{C}' \diamond$.

We can now complete Definition 3 for the receiving and the sending cases, to take into account paths that do not exist yet, but which will exist when the message receipt or message sending has been completed.

Definition 9 (accessibility—receiving and sending). *Completing Definition 3:* $\mathcal{A}_C(\alpha, -1.x.\bar{f}) = (\iota, \kappa)$ iff

$$\alpha.\text{st}_C = \text{Receiving} \wedge 9 \leq \alpha.\text{pc}_C < 18 \wedge \mathcal{C}(\psi(x).\bar{f}) = \iota \wedge \text{Class}(\alpha) \vdash b.x.\bar{f} : \kappa$$

where (b, ψ) is the frame popped at line 8,

or

$$\alpha.\text{st}_C = \text{Sending} \wedge \alpha.\text{pc}_C = 23 \wedge \mathcal{C}(\psi'(x).\bar{f}) = \iota \wedge \text{Class}(\alpha') \vdash b'.x.\bar{f} : \kappa$$

where α' is the actor to receive the app-message, and
 (b', ψ') is the frame to be sent in line 23.

Example: When actor α_1 executes **Receiving**, and its program counter is between 9 and 18, then $\mathcal{A}_{C_0}(\alpha_1, -1.x.f_5) = (\omega_6, \text{write})$, even though x is not yet on the stack frame. As soon as the frame is pushed on the stack, and we reach program counter 20, then $\mathcal{A}_{C_0}(\alpha_1, -1.x.f_5)$ is undefined, but $\mathcal{A}_{C_0}(\alpha_1, x.f_5) = (\omega_6, \text{write})$.

4.6 Actor Behaviour

As our model is parametric with the host language, we do not aim to describe any of the actions performed while executing behaviours, such as synchronous method calls and pushing frames onto stacks, conditionals, loops etc. Instead, we concentrate on how behaviour execution may affect GC; this happens only when the heap is mutated either by object creation or by mutation of objects' fields (since this affects accessibility). In particular, our model does not accommodate for recursive calls; we claim that the result from the current model would easily be extended to a model with recursion in synchronous behaviour, but would require a considerable notation overhead.

Figure 9 shows the actions of an actor α while in the EXECUTE state, *i.e.* while it executes behaviours synchronously. The description is nondeterministic: the procedures **Goldle**, or **Create**, or **MutateHeap**, may execute when the corresponding preconditions hold. Thus, we do not describe the execution of a given program, rather we describe all possible executions for any program. In **Goldle**, the actor α simply passes from the execution state to the idle state; the only condition is that its state is EXECUTE (line 2). It deletes the frame, and sets the actor's state to IDLE (line 4). **Create** creates a new object, initialises its fields to null, and stores its address into local variable x .

The most interesting procedure is field assignment, **MutateHeap**. line 8 modifies the object at address ι_1 , reachable through local path $lp1$, and stores in its field f the address ι_2 which was reachable through local path $lp2$. We require that the type system makes the following two guarantees: line 2, second conjunct, requires that $lp1$ should be writable, while line 3 requires that $lp2$ should be accessible. Line 4 and line 5 require that capabilities of objects do not increase through heap mutation: any address that is accessible with a capability κ after the field update was accessible with the same or more permissive capability κ' before the field update. This requirement guarantees preservation of data race freedom, *i.e.* that $\models \mathcal{C} \diamond$ implies $\models \mathcal{C}[\iota_1, f \mapsto \iota_2] \diamond$.

```

1 Goldle( $\alpha$ ):
2    $\alpha.st = EXECUTE$ 
3    $\rightarrow$ 
4   {  $\alpha.frame := \emptyset$ ;  $\alpha.st := IDLE$ ; }
5
6 Create( $\alpha$ ):
7    $\alpha.st = EXECUTE \wedge \text{fresh } \omega \wedge \mathcal{O}(\omega) = \alpha$ 
8    $\rightarrow$ 
9   {
10    heap :=
11    heap[ $\omega \mapsto (f_1 \mapsto \text{null}, \dots, f_n \mapsto \text{null})$ ]
12     $\alpha.frame := \alpha.frame[x \mapsto \omega]$ 
13  }

1 MutateHeap( $\alpha$ ):
2    $\alpha.st = EXECUTE \wedge \mathcal{A}_C(\alpha, lp1) = (\iota_1, \text{write})$ 
3    $\wedge \mathcal{A}_C(\alpha, lp2) = \iota_2$ 
4    $\wedge \forall \iota, \kappa, lp \ [ \mathcal{A}_{C[\iota_1, f \mapsto \iota_2]}(\alpha, lp) = (\iota, \kappa) \longrightarrow$ 
5      $(\exists \kappa', lp' \ \mathcal{A}_C(\alpha, lp') = (\iota, \kappa') \wedge \kappa' \leq \kappa ) ]$ 
6    $\rightarrow$ 
7   {
8     heap := heap[ $\iota_1, f \mapsto \iota_2$ ]
9   }

```

Fig. 9. Pseudo-code for synchronous operations.

Heap Mutation Does not Affect Accessibility in Other Actors. Heap mutation either creates new objects, which will not be accessible to other actors, or modifies objects to which the current actor has **write** access. By $\models \mathcal{C} \diamond$ all other actors have only **tag** access to the modified object. Therefore, because of *capabilities' degradation with growing paths (as in A1 and A2)*, no other actor will be able to access objects reachable through paths that go through the modified object.

5 Soundness and Completeness

In this section we show soundness and completeness of ORCA.

5.1 I_1 and I_2 Support Safe Local GC

As we said earlier, I_1 and I_2 support safe local GC. Namely, I_1 guarantees that as long as GC only traces objects to which the actor has **read** or **write** access, there will be no data races with other actors' behaviour or GC. And I_2 guarantees that collection can take place based on local information only:

Definition 10. For a configuration \mathcal{C} , and object address ω we say that

- ω is globally inaccessible in \mathcal{C} , iff $\forall \alpha, p. \mathcal{A}_C(\alpha, p) \neq \omega$
- ω is collectable, iff $LRC_C(\omega) = 0$, and $\forall lp. \mathcal{A}_C(\mathcal{O}(\omega), lp) \neq \omega$.

Lemma 2. If I_2 holds, then every collectable object is globally inaccessible.

5.2 Completeness

In [16] we show that globally inaccessible objects remain so, and that for any globally inaccessible object there exists a sequence of steps which will collect it.

Theorem 1 (Inaccessibility is monotonic). *For any configurations \mathcal{C} , and \mathcal{C}' , if \mathcal{C}' is the outcome of the execution of any single line of code from any of the procedures from Figs. 6, 7, 8 and 9, and ω is globally inaccessible in \mathcal{C} , then ω is globally inaccessible in \mathcal{C}' .*

Theorem 2 (Completeness of ORCA). *For any configuration \mathcal{C} , and object address ω which is globally inaccessible in \mathcal{C} , there exists a finite sequence of steps which lead to \mathcal{C}' in which $\omega \notin \text{dom}(\mathcal{C}')$.*

5.3 Dealing with Fine-Grained Concurrency

So far, we have discussed actions under an assumption atomicity. However, ORCA needs to work under fine-grained concurrency, whereby several actors may be executing concurrently, each of them executing a behaviour, or sending or receiving a message, or collecting garbage. With fine-grained concurrency, and with the preliminary definitions of AMC and OMC, the invariants are no longer preserved. In fact, they need never hold!

Example: Consider Fig. 4, and assume that actor α_1 was executing [Receiving](#). Then, at line 7 and before popping the message off the queue, we have $\text{LRC}(\omega_5) = 2$, $\text{FRC}(\omega_5) = 1$, $\text{AMC}^p(\omega_5) = 1$, where $\text{AMC}^p(-)$ stands for the preliminary definition of AMC; thus \mathbf{I}_4 holds. After popping and before updating the RC for ω_5 , *i.e.* between lines 9 and 11, we have $\text{AMC}^p(\omega_5) = 0$ —thus \mathbf{I}_4 is broken. At first sight, this might not seem a big problem, because the update of RC at line 12 will set $\text{LRC}(\omega_5) = 1$, and thus restore \mathbf{I}_4 . However, if there was another message containing ω_5 in α_2 's queue, and consider a snapshot where α_2 had just finished line 8 and α_1 had just finished line 12, then the update of α_1 's RC will *not* restore \mathbf{I}_4 .

The reason for this problem is, that with the preliminary definition $\text{AMC}^p(-)$, upon popping at line 8, the AMC is decremented in one atomic step for all objects accessible from the message, while the RC is updated later on (at line 12 or line 14), and one object at a time. In other words, the updates to AMC and LRC are not in sync. Instead, we give the full definition of AMC so, that AMC is in sync LRC; namely it is not affected by popping the message, and is reduced one object at a time once we reach program counter line 15. Similarly, because updating the RC's takes place in a separate step from the removal of the ORCA-message from its queue, we refine the definition of OMC:

Definition 11 (Auxiliary Counters for AMC, and OMC)

$$\begin{aligned} \text{AMC}_{\mathcal{C}}^{\text{cv}}(\iota) &\equiv \# \{ \alpha \mid \alpha.\text{st}_{\mathcal{C}} = \text{RECEIVE} \wedge 9 \leq \alpha.\text{pc}_{\mathcal{C}} \wedge \\ &\quad \iota \in \alpha.\text{ws} \setminus \text{CurrAddrRcv}_{\mathcal{C}}(\alpha) \} \\ \text{CurrAddrRcv}_{\mathcal{C}}(\alpha) &\equiv \begin{cases} \{ \iota_{10} \} & \text{if } \alpha.\text{pc}_{\mathcal{C}} = 15 \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

In the above $\alpha.\mathbf{ws}$ refers to the contents of the variable \mathbf{ws} while the actor α is executing the pseudocode from [Receiving](#), and ι_{10} refers to the contents of the variable ι arbitrarily chosen in line 10 of the code.

We define $\text{AMC}_{\mathcal{C}}^{\text{snd}}(\iota)$, $\text{OMC}_{\mathcal{C}}^{\text{rcv}}(\iota)$, and $\text{OMC}_{\mathcal{C}}^{\text{snd}}(\iota)$ similarly in [16].

The counters AMC^{rcv} and AMC^{snd} are zero except for actors which are in the process of receiving or sending application messages. Also, the counters OMC^{rcv} and AMC^{snd} are zero except for actors which are in the process of receiving or sending ORCA-messages. All these counters are always ≥ 0 . We can now complete the definition of AMC and OMC:

Definition 12 (AMC and OMC – full definition)

$$\text{OMC}_{\mathcal{C}}(\iota) \equiv \sum_j \begin{cases} z & \text{if } \mathcal{O}(\iota).\text{qu}_{\mathcal{C}}[j] = \text{orca}(\iota : z) \\ 0 & \text{otherwise} \end{cases} + \text{OMC}_{\mathcal{C}}^{\text{snd}}(\iota) - \text{OMC}_{\mathcal{C}}^{\text{rcv}}(\iota)$$

$$\text{AMC}_{\mathcal{C}}(\iota) \equiv \#\{ (\alpha, k) \mid k > 0 \wedge \exists x.\bar{f}.\mathcal{A}_{\mathcal{C}}(\alpha, k, x.\bar{f}) = \iota \} + \text{AMC}_{\mathcal{C}}^{\text{snd}}(\iota) + \text{AMC}_{\mathcal{C}}^{\text{rcv}}(\iota)$$

where $\#$ denotes cardinality.

Example: Let us again consider that α_1 was executing [Receiving](#). Then, at line 10 we have $\mathbf{ws} = \{\iota_5, \iota_6\}$ and $\text{AMC}(\omega_5) = 1 = \text{AMC}(\omega_6)$. Assume at the first iteration, at line 10 we chose ι_5 , then right before reaching line 15 we have $\text{AMC}(\omega_5) = 0$ and $\text{AMC}(\omega_6) = 1$. At the second iteration, at line 10 we will chose ι_6 , and then right before reaching 15 we have $\text{AMC}(\omega_6) = 0$.

5.4 Soundness

To complete the definition of well-formed configurations, we need to define what it means for an actor or a queue to be well-formed.

Well-Formed Queues - \mathbf{I}_7 . The owner’s reference count for any live address (*i.e.* any address reachable from a message path, or foreign actor, or in an ORCA message) should be greater than 0 at the current configuration, as well as, at all configurations which arise from receiving pending, but no new, messages from the owner’s queue. Thus, in order to ensure that ORCA decrement messages do not make the local reference count negative, \mathbf{I}_7 requires that the effect of any prefix of the message queue leaves the reference count for any object positive. To formulate \mathbf{I}_7 we use the concept of $\text{QueueEffect}_{\mathcal{C}}(\alpha, \iota, n)$, which describes the contents of LRC after the actor α has consumed and reacted to the first n messages in its queue—*i.e.* is about “looking into the future”. Thus, for actor α , address ι , and number n we define the effect of the n -prefix of the queue on the reference count as follows:

$$\text{QueueEffect}_{\mathcal{C}}(\alpha, \iota, n) \equiv \text{LRC}_{\mathcal{C}}(\iota) - z + \sum_{j=0}^n \text{Weight}_{\mathcal{C}}(\alpha, \iota, j)$$

where $z = k$, if α is in the process of executing [ReceiveORCA](#), and $\alpha.\text{pc}_{\mathcal{C}} = 6$, and $\alpha.\text{qu}.\text{top} = \text{orca}(\iota : k)$, and otherwise $z = 0$.

And where,

$$Weight_C(\alpha, \iota, j) \equiv \begin{cases} z' & \text{if } \alpha.\text{qu}_C[j] = \text{orca}(\iota : z') \\ -1 & \text{if } \exists x. \exists \bar{f}. \mathcal{A}_C(\alpha, k.x.\bar{f}) = \iota \wedge \mathcal{O}(\iota) = \alpha \\ 0 & \text{otherwise} \end{cases}$$

I₇ makes the following four guarantees: [a] The effect of any prefix of the message queue leaves the *LRC* non-negative. [b] If ι is accessible from the j -th message in its owner's queue, then the *LRC* for ι will remain >0 during execution of the current message queue up to, and including, the j -th message. [c] If ι is accessible from an ORCA-message, then the *LRC* will remain >0 during execution of the current message queue, up to and excluding execution of the ORCA-message itself. [d] If ι is globally accessible (*i.e.* reachable from a local path or from a message in a non-owning actor) then $LRC(\iota)$ is currently >0 , and will remain so after during popping of all the entries in the current queue.

Definition 13 (I₇). $\models_{Queues} C$, iff for all $j \in \mathbb{N}$, for all addresses ι , actors α , α' , where $\mathcal{O}(\iota) = \alpha \neq \alpha'$, the following conditions hold:

- a $\forall n. QueueEffect_C(\alpha, \iota, n) \geq 0$
- b $\exists x. \exists \bar{f}. \mathcal{A}_C(\alpha, j.x.\bar{f}) = \iota \longrightarrow \forall k \leq j. QueueEffect_C(\alpha, \iota, k) > 0.$
- c $\alpha.\text{qu}_C[j] = \text{orca}(\iota : z) \longrightarrow \forall k < j. QueueEffect_C(\alpha, \iota, k) > 0.$
- d $\exists p. \mathcal{A}_C(\alpha', p) = \iota \longrightarrow \forall k \in \mathbb{N}. QueueEffect_C(\alpha, \iota, k) > 0.$

For example, in a configuration with $LRC(\iota) = 2$, and a queue with $\text{orca}(\iota : -2) :: \text{orca}(\iota : -1) :: \text{orca}(\iota : 256)$ is illegal by **I₇**[a]. Similarly, in a configuration with $LRC(\iota) = 2$, and a queue with $\text{orca}(\iota : -2) :: \text{orca}(\iota : 256)$, the owning actor could collect ι before popping the message $\text{orca}(\iota : 256)$ from its queue. Such a configuration is also deemed illegal by **I₇**[c].

I₈-Well-Formed Actor. In [16] we define well-formedness of an actor α through the judgement $C, \alpha \vdash \text{st}$. This judgement depends on α 's current state **st**, and requires, among other things, that the contents of the local variables **ws**, **ms** are consistent with the contents of the **pc** and **RC**. Remember also, that because **Receiving** and **Sending** modify the **ws** or send ORCA-messages before updating the frame or sending the application message, in the definition of AMC and OMC we took into account the internal state of actors executing such procedures.

Well-Formed Configuration. The following completes Definition 6 from Sect. 4.2.

Definition 14 (Well-formed configurations—full). A configuration C is well-formed, $\models C$, iff **I₁**–**I₆** (Definition 6) for C , if its queues are well-formed ($\models_{Queues} C$, **I₇**), as well as, all its actors ($C, \alpha \vdash \alpha.\text{st}_C$, **I₈**).

In [16] we consider the execution of each line in the codes from Sect. 4, and prove:

Theorem 3 (Soundness of ORCA). For any configurations C and C' : If $\models C$, and C' is the outcome of the execution of any single line of code from any of the procedures from Figs. 6, 7, 8 and 9, then $\models C'$.

This theorem together with \mathbf{I}_6 implies that ORCA never leaves accessible paths dangling. Note that the theorem is stated so as to be applicable for a fine interleaving of the execution. Even though we expressed ORCA through procedures, in our proof we cater for an execution where one line of any of these procedures is executed interleaved with any other procedures in the other actors.

6 Related Work

The challenges faced when developing and debugging concurrent garbage collectors have motivated the development of formal models and proofs of correctness [6, 13, 19, 30, 35]. However, most work considers a global heap where mutator and collector threads *race* for objects and relies on synchronisation mechanisms (or atomic reduction steps), such as read or write barriers, in contrast to ORCA which considers many local heaps, no atomicity or synchronization, and relies on the properties of the type system. McCreight et al. [25] introduced a framework to reason about and build certified garbage collectors, verifying independently both mutator and collector threads. Their work focuses mainly on garbage collectors similar to those that run on Java programs, such as STW mark-and-sweep, STW copying and incremental copying. Vechev et al. [39] specified concurrent mark-and-sweep collectors with write barriers for synchronisation. The authors also present a parametric garbage collector from which other collectors can be derived. Hawblitzel and Petrank [22] mechanized proofs of two real-world collectors (copying and mark-and-sweep) and their respective allocators. The assembly code was instrumented with pre- and post-conditions, invariants and assertions, which were then verified using Z3 and Boogie. Ugawa et al. [38] extended a copying, on-the-fly, concurrent garbage collector to process reference types. The authors model-checked their algorithm using a model that limited the number of objects and threads. Gamie et al. [17] machine-checked a state-of-the-art, on-the-fly, concurrent, mark-and-sweep garbage collector [32]. They modelled one collector thread and many mutator threads. ORCA does not limit the number of actors running concurrently.

Local heaps have been used in the context of garbage collection to reduce the amount of synchronisation required before [1–3, 13, 15, 24, 31, 34], where different threads have their own heap and share a global heap. However, only two of these have been proved correct. Doligez and Gonthier [13] proved a collector [14] which splits the heap into many local heaps and one global heap, and uses mark-and-sweep for individual collection of local heaps. The algorithm imposes restrictions on the object graph, that is, a thread cannot access objects in other threads' local heaps. ORCA allows for references across heaps. Raghunathan et al. [34] proved correct a hierarchical model of local heaps for functional programming languages. The work restricted objects graphs and prevented mutation.

As for collectors that rely on message passing, Moreau et al. [26] revisited the Birrell's reference listing algorithm, which also uses message passing to update reference counts in a distributed system, and presented its formalisation and proofs of soundness and completeness. Moreover, Clebsch and Drossopoulou [10] proved correct MAC, a concurrent collector for actors.

7 Conclusions

We have shown the soundness and completeness of the ORCA actor memory reclamation protocol. The ORCA model is not tied to a particular programming language and is parametric in the host language. Instead it relies on a number of invariants and properties which can be met by a combination of language and static checks. The central property that is required is the absence of data races on objects shared between actors.

We developed a formal model of ORCA and identified requirements for the host language, its type system, or associated tooling. We described ORCA at a language-agnostic level and identified eight invariants that capture how global consistency is obtained in the absence of synchronisation. We proved that ORCA will not prematurely collect objects (soundness) and that all garbage will be identified as such (completeness).

Acknowledgements. We are deeply grateful to Tim Wood for extensive discussions and suggestions about effective communication of our ideas. We thank Rakhilya Mekhtieva for her contributions to the formal proofs, Sebastian Blessing and Andy McNeil for their contributions to the implementation, as well as the anonymous reviewers for their insightful comments. This work was initially funded by Causality Ltd, and has also received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement 695412) and the FP7 project UPSCALE, the Swedish Research council through the grant Structured Aliasing and the UPMARC Linneaus Centre of Excellence, the EPSRC (grant EP/K011715/1), the NSF (award 1544542) and ONR (award 503353).

References

1. Armstrong, J.: A history of Erlang. In: HOPL III (2007)
2. Auerbach, J., Bacon, D.F., Guerraoui, R., Spring, J.H., Vitek, J.: Flexible task graphs: a unified restricted thread programming model for Java. In: LCTES (2008)
3. Auhagen, S., Bergstrom, L., Fluet, M., Reppy, J.: Garbage collection for multicore NUMA machines. In: MSPC (2011). <https://doi.org/10.1145/1988915.1988929>
4. Boyland, J., Noble, J., Retert, W.: Capabilities for sharing: a generalisation of uniqueness and read-only. In: ECOOP (2001). https://doi.org/10.1007/3-540-45337-7_2
5. Brandauer, S., et al.: Parallel objects for multicores: a glimpse at the parallel language ENCORE. In: Bernardo, M., Johnsen, E.B. (eds.) SFM 2015. LNCS, vol. 9104, pp. 1–56. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18941-3_1
6. Cheng, P.S.D.: Scalable real-time parallel garbage collection for symmetric multiprocessors. Ph.D. thesis, Carnegie Mellon University (2001)
7. Clarke, D., Wrigstad, T., Östlund, J., Johnsen, E.B.: Minimal ownership for active objects. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 139–154. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_11
8. Clebsch, S.: Pony: co-designing a type system and a runtime. Ph.D. thesis, Imperial College London (2018, to be published)

9. Clebsch, S., Blessing, S., Franco, J., Drossopoulou, S.: Ownership and reference counting based garbage collection in the actor world. In: ICIOOLPS (2015)
10. Clebsch, S., Drossopoulou, S.: Fully concurrent garbage collection of actors on many-core machines. In: OOPSLA (2013). <https://doi.org/10.1145/2544173.2509557>
11. Clebsch, S., Drossopoulou, S., Blessing, S., McNeil, A.: Deny capabilities for safe, fast actors. In: AGERE! (2015). <https://doi.org/10.1145/2824815.2824816>
12. Clebsch, S., Franco, J., Drossopoulou, S., Yang, A., Wrigstad, T., Vitek, J.: Orca: GC and type system co-design for actor languages. In: OOPSLA (2017). <https://doi.org/10.1145/3133896>
13. Doligez, D., Gonthier, G.: Portable, unobtrusive garbage collection for multiprocessor systems. In: POPL (1994). <https://doi.org/10.1145/174675.174673>
14. Doligez, D., Leroy, X.: A concurrent, generational garbage collector for a multithreaded implementation of ML. In: POPL (1993). <https://doi.org/10.1145/158511.158611>
15. Domani, T., Goldshtein, G., Kolodner, E.K., Lewis, E., Petrank, E., Sheinwald, D.: Thread-local heaps for Java. In: ISMM (2002). <https://doi.org/10.1145/512429.512439>
16. Franco, J., Clebsch, S., Drossopoulou, S., Vitek, J., Wrigstad, T.: Soundness of a concurrent collector for actors (extended version). Technical report, Imperial College London (2018). <https://www.doc.ic.ac.uk/research/technicalreports/2018/>
17. Gamie, P., Hosking, A., Engelhard, K.: Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In: PLDI (2015). <https://doi.org/10.1145/2737924.2738006>
18. Gordon, C.S., Parkinson, M.J., Parsons, J., Bromfield, A., Duffy, J.: Uniqueness and reference immutability for safe parallelism. In: OOPSLA (2012). <https://doi.org/10.1145/2384616.2384619>
19. Gries, D.: An exercise in proving parallel programs correct. *Commun. ACM* **20**(12), 921–930 (1977). <https://doi.org/10.1145/359897.359903>
20. Haller, P., Loiko, A.: LaCasa: lightweight affinity and object capabilities in Scala. In: OOPSLA (2016). <https://doi.org/10.1145/2983990.2984042>
21. Haller, P., Odersky, M.: Capabilities for uniqueness and borrowing. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 354–378. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_17
22. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. In: POPL (2009). <https://doi.org/10.1145/1480881.1480935>
23. Kniesel, G., Theisen, D.: JAC-access right based encapsulation for Java. *Softw. Pract. Exp.* **31**(6), 555–576 (2001)
24. Marlow, S., Peyton Jones, S.: Multicore garbage collection with local heaps. In: ISMM (2011). <https://doi.org/10.1145/1993478.1993482>
25. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: PLDI (2007). <https://doi.org/10.1145/1250734.1250788>
26. Moreau, L., Dickman, P., Jones, R.: Birrell's distributed reference listing revisited. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **27**(6), 1344–1395 (2005). <https://doi.org/10.1145/1108970.1108976>
27. Moreau, L., Duprat, J.: A construction of distributed reference counting. *Acta Informatica* **37**(8), 563–595 (2001). <https://doi.org/10.1007/PL00013315>
28. Östlund, J.: Language constructs for safe parallel programming on multi-cores. Ph.D. thesis, Department of Information Technology, Uppsala University (2016)

29. Östlund, J., Wrigstad, T., Clarke, D., Åkerblom, B.: Ownership, uniqueness, and immutability. In: Paige, R.F., Meyer, B. (eds.) *TOOLS EUROPE 2008*. LNBIP, vol. 11, pp. 178–197. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69824-1_11
30. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. *Acta Informatica* **6**(4), 319–340 (1976)
31. Pizlo, F., Hosking, A.L., Vitek, J.: Hierarchical Real-Time Garbage Collection (2007). <https://doi.org/10.1145/1254766.1254784>
32. Pizlo, F., Ziarek, L., Maj, P., Hosking, A.L., Blanton, E., Vitek, J.: Schism: fragmentation-tolerant real-time garbage collection. In: *PLDI* (2010). <https://doi.org/10.1145/1806596.1806615>
33. Potanin, A., Östlund, J., Zibin, Y., Ernst, M.D.: Immutability. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. LNCS, vol. 7850, pp. 233–269. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36946-9_9
34. Raghunathan, R., Muller, S.K., Acar, U.A., Belloch, G.: Hierarchical memory management for parallel programs. In: *ICFP* (2016). <https://doi.org/10.1145/2951913.2951935>
35. Ramesh, S., Mehndiratta, S.: The liveness property of on-the-fly garbage collector - a proof. *Inf. Process. Lett.* **17**(4), 189–195 (1983)
36. Srinivasan, S., Mycroft, A.: Kilim: isolation-typed actors for Java. In: Vitek, J. (ed.) *ECOOP 2008*. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70592-5_6
37. Tschantz, M.S., Ernst, M.D.: Javari: adding reference immutability to Java. In: *OOPSLA* (2005). <https://doi.org/10.1145/1094811.1094828>
38. Ugawa, T., Jones, R.E., Ritson, C.G.: Reference object processing in on-the-fly garbage collection. In: *ISMM* (2014). <https://doi.org/10.1145/2602988.2602991>
39. Vechev, M.T., Yahav, E., Bacon, D.F.: Correctness-preserving derivation of concurrent garbage collection algorithms. In: *PLDI* (2006). <https://doi.org/10.1145/1133981.1134022>
40. Yang, A.M., Wrigstad, T.: Type-assisted automatic garbage collection for lock-free data structures. In: *ISMM* (2017). <https://doi.org/10.1145/3092255.3092274>
41. Zibin, Y., Potanin, A., Li, P., Ali, M., Ernst, M.D.: Ownership and immutability in generic Java. In: *OOPSLA* (2010). <https://doi.org/10.1145/1932682.1869509>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Paxos Consensus, Deconstructed and Abstracted

Álvaro García-Pérez¹(✉) , Alexey Gotsman¹, Yuri Meshman¹,
and Ilya Sergey²

¹ IMDEA Software Institute, Madrid, Spain

{alvaro.garcia.perez,alexey.gotsman,yuri.meshman}@imdea.org

² University College London, London, UK

i.sergey@ucl.ac.uk

Abstract. Lamport’s Paxos algorithm is a classic consensus protocol for state machine replication in environments that admit crash failures. Many versions of Paxos exploit the protocol’s intrinsic properties for the sake of gaining better run-time performance, thus widening the gap between the original description of the algorithm, which was proven correct, and its real-world implementations. In this work, we address the challenge of specifying and verifying complex Paxos-based systems by (a) devising composable specifications for implementations of Paxos’s single-decree version, and (b) engineering disciplines to reason about protocol-aware, semantics-preserving optimisations to single-decree Paxos. In a nutshell, our approach elaborates on the deconstruction of single-decree Paxos by Boichat et al. We provide novel non-deterministic specifications for each module in the deconstruction and prove that the implementations refine the corresponding specifications, such that the proofs of the modules that remain unchanged can be reused across different implementations. We further reuse this result and show how to obtain a verified implementation of Multi-Paxos from a verified implementation of single-decree Paxos, by a series of novel protocol-aware transformations of the network semantics, which we prove to be behaviour-preserving.

1 Introduction

Consensus algorithms are an essential component of the modern fault-tolerant deterministic services implemented as message-passing distributed systems. In such systems, each of the distributed nodes contains a replica of the system’s state (*e.g.*, a database to be accessed by the system’s clients), and certain nodes may propose values for the next state of the system (*e.g.*, requesting an update in the database). Since any node can crash at any moment, all the replicas have to keep copies of the state that are consistent with each other. To achieve this, at each update to the system, all the non-crashed nodes run an instance of a *consensus protocol*, uniformly deciding on its outcome. The safety requirements for consensus can be thus stated as follows: “only a single value is decided uniformly by all non-crashed nodes, it never changes in the future, and the decided value has been proposed by some node participating in the protocol” [16].

© The Author(s) 2018

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 912–939, 2018.

https://doi.org/10.1007/978-3-319-89884-1_32

The Paxos algorithm [15, 16] is the classic consensus protocol, and its single-decree version (SD-Paxos for short) allows a set of distributed nodes to reach an agreement on the outcome of a *single* update. Optimisations and modifications to SD-Paxos are common. For instance, the multi-decree version, often called Multi-Paxos [15, 27], considers multiple slots (*i.e.*, multiple positioned updates) and decides upon a result for *each* slot, by running a slot-specific instance of an SD-Paxos. Even though it is customary to think of Multi-Paxos as of a series of independent SD-Paxos instances, in reality the implementation features multiple protocol-aware optimisations, exploiting intrinsic dependencies between separate single-decree consensus instances to achieve better throughput. To a great extent, these and other optimisations to the algorithm are pervasive, and verifying a modified version usually requires to devise a new protocol definition and a proof from scratch. New versions are constantly springing (*cf.* Sect. 5 of [27] for a comprehensive survey) widening the gap between the description of the algorithms and their real-world implementations.

We tackle the challenge of *specifying* and *verifying* these distributed algorithms by contributing two verification techniques for consensus protocols.

Our first contribution is a family of composable specifications for Paxos' core subroutines. Our starting point is the deconstruction of SD-Paxos by Boichat *et al.* [2, 3], allowing one to consider a distributed consensus instance as a *shared-memory concurrent program*. We introduce novel specifications for Boichat *et al.*'s modules, and let them be non-deterministic. This might seem as an unorthodox design choice, as it *weakens* the specification. To show that our specifications are still *strong enough*, we restore the top-level *deterministic* abstract specification of the consensus, which is convenient for client-side reasoning. The weakness introduced by the non-determinism in the specifications has been impelled by the need to prove that the implementations of Paxos' components *refine* the specifications we have ascribed [9]. We prove the refinements modularly via the Rely/Guarantee reasoning with prophecy variables and explicit linearisation points [11, 26]. On the other hand, this weakness becomes a virtue when better understanding the volatile nature of Boichat *et al.*'s abstractions and of the Paxos algorithm, which may lead to newer modifications and optimisations.

Our second contribution is a methodology for verifying composite consensus protocols by reusing the proofs of their constituents, targeting specifically Multi-Paxos. We distill protocol-aware system optimisations into a separate semantic layer and show how to obtain the realistic Multi-Paxos implementation from SD-Paxos by a *series of transformations* to the *network semantics* of the system, as long as these transformations preserve the behaviour observed by clients. We then provide a family of such transformations along with the formal conditions allowing one to compose them in a behaviour-preserving way.

We validate our approach for construction of modularly verified consensus protocols by providing an executable proof-of-concept implementation of Multi-Paxos with a high-level shared memory-like interface, obtained via a series of behaviour-preserving network transformations. The full proofs of lemmas and

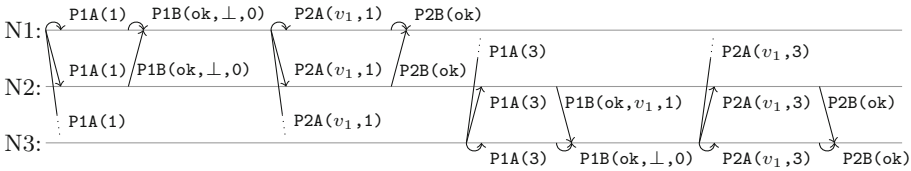


Fig. 1. A run of SD-Paxos.

theorems from our development, as well as some boilerplate definitions, are given in the appendices of the supplementary extended version of this paper.¹

2 The Single-Decree Paxos Algorithm

We start with explaining SD-Paxos through an intuitive scenario. In SD-Paxos, each node in the system can adopt the roles of *proposer* or *acceptor*, or both. A value is decided when a *quorum* (*i.e.*, a majority of acceptors) accepts the value proposed by some proposer. Now consider a system with three nodes N1, N2 and N3, where N1 and N3 are both proposers and acceptors, and N2 is an acceptor, and assume N1 and N3 propose values v_1 and v_3 , respectively.

The algorithm works in two phases. In Phase 1, a proposer polls every acceptor in the system and tries to convince a quorum to promise that they will later accept its value. If the proposer succeeds in Phase 1 then it moves to Phase 2, where it requests the acceptors to fulfil their promises in order to get its value decided. In our example, it would seem in principle possible that N1 and N3 could respectively convince two different quorums—one consisting of N1 and N2, and the other consisting of N2 and N3—to go through both phases and to respectively accept their values. This would happen if the communication between N1 and N3 gets lost and if N2 successively grants the promise and accepts the value of N1, and then does the same with N3. This scenario breaks the safety requirements for consensus because both v_1 and v_3 —which can be different—would get decided. However, this cannot happen. Let us explain why.

The way SD-Paxos enforces the safety requirements is by distinguishing each attempt to decide a value with a unique *round*, where the rounds are totally ordered. Each acceptor stores its current round, initially the least one, and only grants a promise to proposers with a round greater or equal than its current round, at which moment the acceptor switches to the proposer's round. Figure 1 depicts a possible run of the algorithm. Assume that rounds are natural numbers, that the acceptors' current rounds are initially 0, and that the nodes N1 and N3 attempt to decide their values with rounds 1 and 3 respectively. In Phase 1, N1 tries to convince a quorum to switch their current round to 1 (messages P1A(1)). The message to N3 gets lost and the quorum consisting of N1 and N2 switches round and promises to only accept values at a round greater or

¹ Find the extended version online at <https://arxiv.org/abs/1802.05969>.

<i>Paxos</i>	1 <code>val vP := undef;</code>
<i>Round-Based Consensus</i>	2 <code>proposeP(val v0) {</code>
<i>Round-Based Register</i>	3 <code> (assume(!(v0 = undef));</code>
	4 <code> if (vP = undef) {</code>
	5 <code> vP := v0;</code>
	6 <code> } return vP; } }</code>

Fig. 2. Deconstruction of SD-Paxos (left) and specification of module *Paxos* (right).

equal than 1. Each acceptor that switches to the proposer's round sends back to the proposer its stored value and the round at which this value was accepted, or an undefined value if the acceptor never accepted any value yet (messages $P1B(ok, \perp, 0)$, where \perp denotes a default undefined value). After Phase 1, N1 picks as a candidate value the one accepted at the greatest round from those returned by the acceptors in the quorum, or its proposed value if all acceptors returned an undefined value. In our case, N1 picks its value v_1 . In Phase 2, N1 requests the acceptors to accept the candidate value v_1 at round 1 (messages $P2A(v_1, 1)$). The message to N3 gets lost, and N1 and N2 accept value v_1 , which gets decided (messages $P2B(ok)$).

Now N3 goes through Phase 1 with round 3 (messages $P1A(3)$). Both N2 and N3 switch to round 3. N2 answers N3 with its stored value v_1 and with the round 1 at which v_1 was accepted (message $P1B(ok, v_1, 1)$), and N3 answers itself with an undefined value, as it has never accepted any value yet (message $P1B(ok, \perp, 0)$). This way, if some value has been already decided upon, *any* proposer that convinces a quorum to switch to its round would receive the decided value from some of the acceptors in the quorum (recall that two quorums have a non-empty intersection). That is, N3 picks the v_1 returned by N2 as the candidate value, and in Phase 2 it manages that the quorum N2 and N3 accepts v_1 at round 3 (messages $P2A(v_1, 3)$ and $P2B(ok)$). N3 succeeds in making a new decision, but the decided value remains the same, and, therefore, the safety requirements of a consensus protocol are satisfied.

3 The Faithful Deconstruction of SD-Paxos

We now recall the faithful deconstruction of SD-Paxos in [2,3], which we take as the reference architecture for the implementations that we aim to verify. We later show how each module of the deconstruction can be verified separately.

The deconstruction is depicted on the left of Fig. 2, which consists of modules *Paxos*, *Round-Based Consensus* and *Round-Based Register*. These modules correspond to the ones in Fig. 4 of [2], with the exception of *Weak Leader Election*. We assume that a correct process that is trusted by every other correct process always exists, and omit the details of the leader election. Leaders take the role of proposers and invoke the interface of *Paxos*. Each module uses the interface provided by the module below it.

```

1  read(int k) {
2      int j; val v; int kW; val maxV;
3      int maxKW; set of int Q; msg m;
4      for (j := 1, j <= n, j++)
5      { send(j, [RE, k]); }
6      maxKW := 0; maxV := undef; Q := {};
7      do { (j, m) := receive();
8          switch (m) {
9              case [ackRE, @k, v, kW]:
10                 Q := Q ∪ {j};
11                 if (kW >= maxKW)
12                 { maxKW := kW; maxV := v; }
13                 case [nackRE, @k]:
14                     return (false, _);
15             } if (|Q| = ⌈(n+1)/2⌉)
16                 { return (true, maxV); } }
17     while (true); }

18 write(int k, val vW) {
19     int j; set of int Q; msg m;
20     for (j := 1, j <= n, j++)
21     { send(j, [WR, k, vW]); }
22     Q := {};
23     do { (j, m) := receive();
24         switch (m) {
25             case [ackWR, @k]:
26                 Q := Q ∪ {j};
27             case [nackWR, @k]:
28                 return false;
29         } if (|Q| = ⌈(n+1)/2⌉)
30             { return true; } }
31     while (true); }

```

Fig. 3. Implementation of *Round-Based Register* (read and write).

The entry module *Paxos* implements SD-Paxos. Its specification (right of Fig. 2) keeps a variable vP that stores the decided value (initially undefined) and provides the operation **proposeP** that takes a proposed value $v0$ and returns vP if some value was already decided, or otherwise it returns $v0$. The code of the operation runs *atomically*, which we emphasise via angle brackets $\langle \dots \rangle$. We define this specification so it meets the safety requirements of a consensus, therefore, any implementation whose entry point refines this specification will have to meet the same safety requirements.

In this work we present both specifications and implementations in pseudo-code for an imperative WHILE-like language with basic arithmetic and primitive types, where **val** is some user-defined type for the values decided by Paxos, and **undef** is a literal that denotes an undefined value. The pseudo-code is self-explanatory and we restraint ourselves from giving formal semantics to it, which could be done in standard fashion if so wished [30]. At any rate, the pseudo-code is ultimately a vehicle for illustration and we stick to this informal presentation.

The implementation of the modules is depicted in Figs. 3, 4 and 5. We describe the modules following a bottom-up approach, which better fits the purpose of conveying the connection between the deconstruction and SD-Paxos. We start with module *Round-Based Register*, which offers operations **read** and **write** (Fig. 3) and implements the replicated processes that adopt the role of acceptors (Fig. 4). We adapt the wait-free, crash-stop implementation of *Round-Based Register* in Fig. 5 of [2] by adding loops for the explicit reception of each individual message and by counting acknowledgement messages one by one. Processes are identified by integers from 1 to n , where n is the number of processes in the system. Proposers and acceptors exchange read and write requests, and their corresponding acknowledgements and non/acknowledgements. We assume a type **msg** for messages and let the message vocabulary to be as follows.

```

1 process Acceptor(int j) {
2   val v := undef; int r := 0; int w := 0;
3   start() {
4     int i; msg m; int k;
5     do { (i, m) := receive();
6       switch (m) {
7         case [RE, k]:
8           if (k < r) { send(i, [nackRE, k]); }
9           else { < r := k; send(i, [ackRE, k, v, w]); } }
10        case [WR, k, vW]:
11          if (k < r) { send(i, [nackWR, k]); }
12          else { < r := k; w := k; v := vW; send(i, [ackWR, k]); } }
13      } }
14   while (true); } }

```

Fig. 4. Implementation of *Round-Based Register* (acceptor).

Read requests $[RE, k]$ carry the proposer's round k . Write requests $[WR, k, v]$ carry the proposer's round k and the proposed value v . Read acknowledgements $[ackRE, k, v, k']$ carry the proposer's round k , the acceptor's value v , and the round k' at which v was accepted. Read non-acknowledgements $[nackRE, k]$ carry the proposer's round k , and so do carry write acknowledgements $[ackWR, k]$ and write non/acknowledgements $[nackWR, K]$.

In the pseudo-code, we use $_$ for a wildcard that could take any literal value. In the pattern-matching primitives, the literals specify the pattern against which an expression is being matched, and operator $@$ turns a variable into a literal with the variable's value. Compare the case $[ackRE, @k, v, kW]$ in Fig. 3, where the value of k specifies the pattern and v and kW get some values assigned, with the case $[RE, k]$ in Fig. 4, where k gets some value assigned.

We assume the network ensures that messages are neither created, modified, deleted, nor duplicated, and that they are always delivered but with an arbitrarily large transmission delay.² Primitive **send** takes the destination j and the message m , and its effect is to send m from the current process to the process j . Primitive **receive** takes no arguments, and its effect is to receive at the current process a message m from origin i , after which it delivers the pair (i, m) of identifier and message. We assume that **send** is non-blocking and that **receive** blocks and suspends the process until a message is available, in which case the process awakens and resumes execution.

Each acceptor (Fig. 4) keeps a value v , a current round r (called the *read round*), and the round w at which the acceptor's value was last accepted (called the *write round*). Initially, v is **undef** and both r and w are 0.

Phase 1 of SD-Paxos is implemented by operation **read** on the left of Fig. 3. When a proposer issues a **read**, the operation requests each acceptor's promise to only accept values at a round greater or equal than k by sending $[RE, k]$

² We allow creation and duplication of $[RE, k]$ messages in Sect. 5, where we obtain Multi-Paxos from SD-Paxos by a series of transformations of the network semantics.

<pre> 1 proposeRC(int k, val v0) { 2 bool res; val v; 3 (res, v) := read(k); 4 if (res) { 5 if (v = undef) { v := v0; } 6 res := write(k, v); 7 if (res) { return (true, v); } } 8 return (false, _); } </pre>	<pre> 1 proposeP(val v0) { 2 int k; bool res; val v; 3 k := pid(); 4 do { (res, v) := 5 proposeRC(k, v0); 6 k := k + n; 7 } while (!res); 8 return v; } </pre>
--	--

Fig. 5. Implementation of *Round-Based Consensus* (left) and *Paxos* (right)

(lines 4–5). When an acceptor receives a `[RE, k]` (lines 5–7 of Fig. 4) it acknowledges the promise depending on its read round. If `k` is strictly less than `r` then the acceptor has already made a promise to another proposer with greater round and it sends `[nackRE, k]` back (line 8). Otherwise, the acceptor updates `r` to `k` and acknowledges by sending `[ackRE, k, v, w]` (line 9). When the proposer receives an acknowledgement (lines 8–10 of Fig. 3) it counts acknowledgements up (line 10) and calculates the greatest write round at which the acceptors acknowledging so far accepted a value, and stores this value in `maxV` (lines 11–12). If a majority of acceptors acknowledged, the operation succeeds and returns `(true, maxV)` (lines 15–16). Otherwise, if the proposer received some `[nackRE, k]` the operation fails, returning `(false, _)` (lines 13–14).

Phase 2 of SD-Paxos is implemented by operation `write` on the right of Fig. 3. After having collected promises from a majority of acceptors, the proposer picks the candidate value `vW` and issues a `write`. The operation requests each acceptor to accept the candidate value by sending `[WR, k, vW]` (lines 20–21). When an acceptor receives `[WR, k, vW]` (line 10 of Fig. 4) it accepts the value depending on its read round. If `k` is strictly less than `r`, then the acceptor never promised to accept at such round and it sends `[nackWR, k]` back (line 11). Otherwise, the acceptor fulfils its promise and updates both `w` and `r` to `k` and assigns `vW` to its value `v`, and acknowledges by sending `[ackWR, k]` (line 12). Finally, when the proposer receives an acknowledgement (lines 23–25 of Fig. 3) it counts acknowledgements up (line 26) and checks whether a majority of acceptors acknowledged, in which case `vW` is decided and the operation succeeds and returns `true` (lines 29–30). Otherwise, if the proposer received some `[nackWR, k]` the operation fails and returns `false` (lines 27–28).³

Next, we describe module *Round-Based Consensus* on the left of Fig. 5. The module offers an operation `proposeRC` that takes a round `k` and a proposed value `v0`, and returns a pair `(res, v)` of Boolean and value, where `res` informs of the success of the operation and `v` is the decided value in case `res` is `true`. We have taken the implementation from Fig. 6 in [2] but adapted to our pseudo-code conventions. *Round-Based Consensus* carries out Phase 1 and Phase 2 of

³ For the implementation to be correct with our shared-memory-concurrency approach, the update of the data in acceptors must happen atomically with the sending of acknowledgements in lines 9 and 12 of Fig. 4.

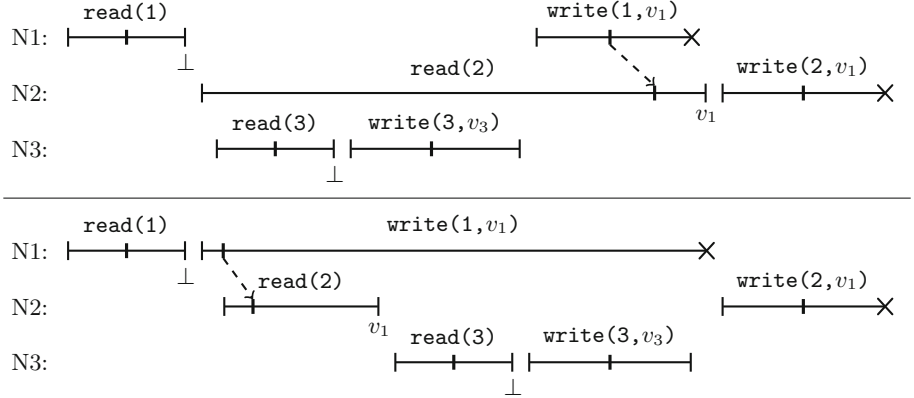


Fig. 6. Two histories in which a failing `write` contaminates some acceptor.

SD-Paxos as explained in Sect. 2. The operation `proposeRC` calls `read` (line 3) and if it succeeds then chooses a candidate value between the proposed value `v0` or the value `v` returned by `read` (line 5). Then, the operation calls `write` with the candidate value and returns `(true, v)` if `write` succeeds, or fails and returns `(false, _)` (line 8) if either the `read` or the `write` fails.

Finally, the entry module *Paxos* on the right of Fig. 5 offers an operation `proposeP` that takes a proposed value `v0` and returns the decided value. We assume that the system primitive `pid()` returns the process identifier of the current process. We have come up with this straightforward implementation of operation `proposeP`, which calls `proposeRC` with increasing round until the call succeeds, starting at a round equal to the process identifier `pid()` and increasing it by the number of processes n in each iteration. This guarantees that the round used in each invocation to `proposeRC` is unique.

The Challenge of Verifying the Deconstruction of Paxos. Verifying each module of the deconstruction separately is cumbersome because of the distributed character of the algorithm and the nature of a linearisation proof. A process may not be aware of the information that will flow from itself to other processes, but this future information flow may dictate whether some operation has to be linearised at the present. Figure 6 illustrates this challenge.

Let N1, N2 and N3 adopt both the roles of acceptors and proposers, which propose values v_1 , v_2 and v_3 with rounds 1, 2 and 3 respectively. Consider the history on the top of the figure. N2 issues a read with round 2 and gets acknowledgements from all but one acceptors in a quorum. (Let us call this one acceptor A.) None of these acceptors have accepted anything yet and they all return \perp as the last accepted value at round 0. In parallel, N3 issues a read with round 3 (third line in the figure) and gets acknowledgements from a quorum in which A does not occur. This read succeeds as well and returns `(true, undef)`.

```

1 (bool × val) ptp[1..n] := undef;
2 val abs_vP := undef; single bool abs_resP[1..n] := undef;
3 proposeP(val v0) {
4   int k; bool res; val v; assume(!(v0 = undef));
5   k := pid(); ptp[pid()] := (true, v0);
6   do { (res, v) := proposeRC(k, v0);
7     if (res) {
8       for (i := 1, i <= n, i++) {
9         if (ptp[i] = (true, v)) { lin(i); ptp[i] := (false, v); } }
10        if (!(v = v0)) { lin(pid()); ptp[pid()] := (false, v0); } } }
11   k := k + n; }
12   while (!res); return v; }

```

Fig. 7. Instrumented implementation of *Paxos*.

Then N3 issues a write with round 3 and value v_3 . Again, it gets acknowledgements from a quorum in which A does not occur, and the write succeeds deciding value v_3 and returns **true**. Later on, and in real time order with the write by N3 but in parallel with the read by N2, node N1 issues a write with round 1 and value v_1 (first line in the figure). This write is to fail because the value v_3 was already decided with round 3. However, the write manages to “contaminate” acceptor A with value v_1 , which now acknowledges N2 and sends v_1 as its last accepted value at round 1. Now N2 has gotten acknowledgements from a quorum, and since the other acceptors in the quorum returned 0 as the round of their last accepted value, the read will catch value v_1 accepted at round 1, and the operation succeeds and returns (\mathbf{true}, v_1) . This history linearises by moving N2’s read after N1’s write, and by respecting the real time order for the rest of the operations. (The linearisation ought to respect the information flow order between N1 and N2 as well, *i.e.*, N1 contaminates A with value v_1 , which is read by N2.)

In the figure, a segment ending in an \times indicates that the operation fails. The value returned by a successful read operation is depicted below the end of the segment. The linearisation points are depicted with a thick vertical line, and the dashed arrow indicates that two operations are in the information flow order.

The variation of this scenario on the bottom of Fig. 6 is also possible, where N1’s write and N2’s read happen concurrently, but where N2’s read is shifted backwards to happen before in real time order with N3’s read and write. Since N1’s write happens before N2’s read in the information flow order, then N1’s write has to inexorably linearise before N3’s operations, which are the ones that will “steal” N1’s valid round.

These examples give us three important hints for designing the specifications of the modules. First, after a decision is committed it is *not enough* to store only the decided value, since a posterior write may contaminate some acceptor with a value different from the decided one. Second, a read operation *may succeed* with some round even if by that time other operation has already succeeded with a higher round. And third, a write with a valid round *may fail* if its round will be “stolen” by a concurrent operation. The non-deterministic specifications that we introduce next allow one to model execution histories as the ones in Fig. 6.

4 Modularly Verifying SD-Paxos

In this section, we provide non-deterministic specifications for *Round-Based Consensus* and *Round-Based Register* and show that each implementation refines its specification [9]. To do so, we instrument the implementations of all the modules with *linearisation-point* annotations and use Rely/Guarantee reasoning [26].

This time we follow a top-down order and start with the entry module *Paxos*.

Module *Paxos*. In order to prove that the implementation on the right of Fig. 5 refines its specification on the right of Fig. 2, we introduce the instrumented implementation in Fig. 7, which uses the helping mechanism for external linearisation points of [18]. We assume that each proposer invokes `proposeP` with a unique proposed value. The auxiliary pending thread pool `ptp[n]` is an array of pairs of Booleans and values of length n , where n is the number of processes in the system. A cell `ptp[i]` containing a pair `(true, v)` signals that the process i proposed value v and the invocation `proposeP(v)` by process i awaits to be linearised. Once this invocation is linearised, the cell `ptp[i]` is updated to the pair `(false, v)`. A cell `ptp[i]` containing `undef` signals that the process i never proposed any value yet. The array `abs_resP[n]` of Boolean single-assignment variables stores the abstract result of each proposer’s invocation. A linearisation-point annotation `lin(i)` takes a process identifier i and performs atomically the abstract operation invoked by proposer i and assigns its result to `abs_resP[i]`. The abstract state is modelled by variable `abs_vP`, which corresponds to variable `vP` in the specification on the right of Fig. 2. One invocation of `proposeP` may help linearise other invocations as follows. The linearisation point is together with the invocation to `proposeRC` (line 6). If `proposeRC` committed with some value v , the instrumented implementation traverses `ptp` and linearises all the proposers which were proposing value v (the proposer may linearise itself in this traversal) (lines 8–9). Then, the current proposer linearises itself if its proposed value v_0 is different from v (line 10), and the operation returns v (line 12). All the annotations and code in lines 6–10 are executed inside an atomic block, together with the invocation to `proposeRC(k, v0)`.

Theorem 1. *The implementation of Paxos on the right of Fig. 5 linearises with respect to its specification on the right of Fig. 2.*

Module *Round-Based Consensus*. The top of Fig. 8 shows the non-deterministic module’s specification. Global variable `vRC` is the decided value, initially `undef`. Global variable `roundRC` is the highest round at which some value was decided, initially 0; a global set of values `valsRC` (initially empty) contains values that may have been proposed by proposers. The specification is non-deterministic in that local value `vD` and Boolean `b` are unspecified, which we model by assigning random values to them. We assume that the current process identifier is $((k - 1) \bmod n) + 1$, which is consistent with how rounds are assigned to each process and incremented in the code of `proposeP` on the right of Fig. 5. If the unspecified value `vD` is neither in the set `valsRC` nor equal to `v0` then the operation returns `(false, _)` (line 11). This models that the operation fails

```

1  val vRC := undef; int roundRC := 0; set of val valsRC := {};
2  proposeRC(int k, val v0) {
3    < val vD := random(); bool b := random();
4      assume(!(v0 = undef)); assume(pid() = ((k - 1) mod n) + 1);
5      if (vD ∈ (valsRC ∪ {v0})) {
6        valsRC := valsRC ∪ {vD};
7        if (b && (k >= roundRC)) { roundRC := k;
8                                if (vRC = undef) { vRC := vD; }
9                                return (true, vRC); }
10       else { return (false, _); } }
11     else { return (false, _); } } }

1  val abs_vRC := undef; int abs_roundRC := 0;
2  set of val abs_valsRC := {};
3  proposeRC(int k, val v0) {
4    single (bool × val) abs_resRC := undef; bool res; val v;
5    assume(!(v0 = undef)); assume(pid() = ((k - 1) mod n) + 1);
6    ⌈ (res, v) := read(k); if (res = false) { linRC(undef, _); } ⌋
7    if (res) { if (v = undef) { v := v0; }
8              ⌈ res := write(k, v); if (res) { linRC(v, true); }
9              else { linRC(v, false); } ⌋
10           if (res) { return (true, v); } } }
11   return (false, _); }

```

Fig. 8. Specification (top) and instrumented implementation (bottom) of *Round-Based Consensus*.

without contaminating any acceptor. Otherwise, the operation may contaminate some acceptor and the value vD is added to the set $valsRC$ (line 6). Now, if the unspecified Boolean b is false, then the operation returns $(false, _)$ (lines 7 and 10), which models that the round will be stolen by a posterior operation. Finally, the operation succeeds if k is greater or equal than $roundRC$ (line 7), and $roundRC$ and vRC are updated and the operation returns $(true, vRC)$ (lines 7–9).

In order to prove that the implementation in Fig. 5 linearises with respect to the specification on the top of Fig. 8, we use the instrumented implementation on the bottom of the same figure, where the abstract state is modelled by variables abs_vRC , $abs_roundRC$ and abs_valsRC in lines 1–2, the local single-assignment variable abs_resRC stores the result of the abstract operation, and the linearisation-point annotations $linRC(vD, b)$ take a value and a Boolean parameters and invoke the non-deterministic abstract operation and disambiguate it by assigning the parameters to the unspecified vD and b of the specification. There are two linearisation points together with the invocations of `read` (line 6) and `write` (line 8). If `read` fails, then we linearise forcing the unspecified vD to be `undef` (line 6), which ensures that the abstract operation fails without adding any value to abs_valsRC nor updating the round $abs_roundRC$. Otherwise, if `write` succeeds with value v , then we linearise forcing the unspecified value vD and Boolean b to be v and `true` respectively (line 8). This ensures that

```

1  read(int k) {
2    ( val vD := random();
3      bool b := random(); val v;
4      assume(vD ∈ valsRR);
5      assume(pid() =
6        ((k - 1) mod n) + 1);
7      if (b) {
8        if (k >= roundRR) {
9          roundRR := k;
10         if (!(vRR = undef)) {
11           v := vRR; }
12         else { v := vD; } }
13       else { v := vD; }
14       return (true, v); }
15     else { return (false, _); } } }

16  val vRR := undef;
17  int roundRR := 0;
18  set of val valsRR := {undef};
19
20  write(int k, val vW) {
21    ( bool b := random();
22      assume(!(vW = undef));
23      assume(pid() =
24        ((k - 1) mod n) + 1);
25      valsRR := valsRR ∪ {vW};
26      if (b && (k >= roundRR)) {
27        roundRR := k;
28        vRR := vW;
29        return true; }
30      else { return false; } } }

```

Fig. 9. Specification of *Round-Based Register*.

the abstract operation succeeds and updates the round `abs_roundRC` to `k` and assigns `v` to the decided value `abs_vRC`. If `write` fails then we linearise forcing the unspecified `vD` and `b` to be `v` and `false` respectively (line 9). This ensures that the abstract operation fails.

Theorem 2. *The implementation of Round-Based Consensus in Fig. 5 linearises with respect to its specification on the top of Fig. 8.*

Module *Round-Based Register*. Figure 9 shows the module’s non-deterministic specification. Global variable `vRR` represents the decided value, initially `undef`. Global variable `roundRR` represents the current round, initially 0, and global set of values `valsRR`, initially containing `undef`, stores values that may have been proposed by some proposer. The specification is non-deterministic in that method `read` has unspecified local Boolean `b` and local value `vD` (we assume that `vD` is `valsRR`), and method `write` has unspecified local Boolean `b`. We assume the current process identifier is $((k - 1) \bmod n) + 1$.

Let us explain the specification of the `read` operation. The operation can succeed regardless of the proposer’s round `k`, depending on the value of the unspecified Boolean `b`. If `b` is `true` and the proposer’s round `k` is valid (line 8), then the read round is updated to `k` (line 9) and the operation returns `(true, v)` (line 14), where `v` is the read value, which coincides with the decided value if some decision was committed already or with `vD` otherwise. Now to the specification of operation `write`. The value `vW` is always added to the set `valsRR` (line 25). If the unspecified Boolean `b` is false (the round will be stolen by a posterior operation) or if the round `k` is non-valid, then the operation returns `false` (lines 26 and 30). Otherwise, the current round is updated to `k`, and the decided value `vRR` is updated to `vW` and the operation returns `true` (lines 27–29).

In order to prove that the implementation in Figs. 3 and 4 linearises with respect to the specification in Fig. 9, we use the instrumented implementation in

Figs. 10 and 11, which uses prophecy variables [1,26] that “guess” whether the execution of the method will reach a particular program location or not. The instrumented implementation also uses external linearisation points. In particular, the code of the acceptors may help to linearise some of the invocations to **read** and **write**, based on the prophecies and on auxiliary variables that count the number of acknowledgements sent by acceptors after each invocation of a **read** or a **write**. The next paragraphs elaborate on our use of prophecy variables and on our helping mechanism.

Variables **abs_vRR**, **abs_roundRR** and **abs_valsRR** in Fig. 10 model the abstract state. They are initially set to **undef**, 0 and the set containing **undef** respectively. Variable **abs_res_r[k]** is an infinite array of single-assignment pairs of Boolean and value that model the abstract results of the invocations to **read**. (Think of an infinite array as a map from integers to some type; we use the array notation for convenience.) Similarly, variable **abs_res_w[k]** is an infinite array of single-assignment Booleans that models the abstract results of the invocations to **write**. All the cells in both arrays are initially **undef** (e.g. the initial maps are empty). Variables **count_r[k]** and **count_w[k]** are infinite arrays of integers that model the number of acknowledgements sent (but not necessarily received yet) from acceptors in response to respectively read or write requests. All cells in both arrays are initially 0. The variable **proph_r[k]** is an infinite array of single-assignment pairs $\text{bool} \times \text{val}$, modelling the prophecy for the invocations of **read**, and variable **proph_w[k]** is an infinite array of single-assignment Booleans modelling the prophecy for the invocations of **write**.

The linearisation-point annotations **linRE(k, vD, b)** for **read** take the proposer’s round **k**, a value **vD** and a Boolean **b**, and they invoke the abstract operation and disambiguate it by assigning the parameters to the unspecified **vD** and **b** of the specification on the left of Fig. 9. At the beginning of a **read(k)** (lines 11–14 of Fig. 10), the prophecy **proph_r[k]** is set to **(true, v)** if the invocation reaches **PL: RE_SUCC** in line 26. The *v* is defined to coincide with **maxV** at the time when that location is reached. That is, *v* is the value accepted at the greatest round by the acceptors acknowledging so far, or undefined if no acceptor ever accepted any value. If the operation reaches **PL: RE_FAIL** in line 24 instead, the prophecy is set to **(false, _)**. (If the method never returns, the prophecy is left **undef** since it will never linearise.) A successful **read(k)** linearises in the code of the acceptor in Fig. 11, when the $\lceil (n+1)/2 \rceil$ th acceptor sends **[ackRE, k, v, w]**, and only if the prophecy is **(true, v)** and the operation was not linearised before (lines 10–14). We force the unspecified **vD** and **b** to be *v* and **true** respectively, which ensures that the abstract operation succeeds and returns **(true, v)**. A failing **read(k)** linearises at the **return** in the code of **read** (lines 23–24 of Fig. 10), after the reception of **[nackRE, k]** from one acceptor. We force the unspecified **vD** and **b** to be **undef** and **false** respectively, which ensures that the abstract operation fails.

The linearisation-point annotations **linWR(k, vW, b)** for **write** take the proposer’s round **k** and value **vW**, and a Boolean **b**, and they invoke the abstract operation and disambiguate it by assigning the parameter to the unspecified **b**

```

1  val abs_vRR := undef; int abs_roundRR := 0;
2  set of val abs_valsRR := {undef};
3  single val abs_res_r[1..∞] := undef;
4  single val abs_res_w[1..∞] := undef;
5  int count_r[1..∞] := 0; int count_w[1..∞] := 0;
6  single (bool × val) proph_r[1..∞] := undef;
7  single bool proph_w[1..∞] := undef;
8  read(int k) {
9    int j; val v; set of int Q; int maxKW; val maxV; msg m;
10   assume(pid() = ((k - 1) mod n) + 1);
11   { if (operation reaches PL: RE_SUCC and define v = maxV at that time) {
12     proph_r[k] := (true, v); }
13   else { if (operation reaches PL: RE_FAIL) {
14     proph_r[k] := (false, _); } } }
15   for (j := 1, j <= n, j++) { send(j, [RE, k]); }
16   maxKW := 0; maxV := undef; Q := {};
17   do { (j, m) := receive();
18     switch (m) {
19       case [ackRE, @k, v, kW]:
20         Q := Q ∪ {j};
21         if (kW >= maxKW) { maxKW := kW; maxV := v; }
22       case [nackRE, @k]:
23         { linRE(k, undef, false); proph_r[k] := undef;
24           return (false, _); } // PL: RE_FAIL
25     } if (|Q| = ⌈(n+1)/2⌉) {
26       return (true, maxV); } } // PL: RE_SUCC
27   while (true); }
28  write(int k, val vW) {
29    int j; set of int Q; msg m;
30    assume(!vW = undef); assume(pid() = ((k - 1) mod n) + 1);
31    { if (operation reaches PL: WR_SUCC) { proph_w[k] := true; }
32    else { if (operation reaches PL: WR_FAIL) {
33      proph_w[k] := false; } } }
34    for (j := 1, j <= n, j++) { send(j, [WR, k, vW]); }
35    Q := {};
36    do { (j, m) := receive();
37      switch (m) {
38        case [ackWR, @k]:
39          Q := Q ∪ {j};
40        case [nackWR, @k]:
41          { if (count_w[k] = 0) {
42            linWR(k, vW, false); proph_w[k] := undef; }
43            return false; } // PL: WR_FAIL
44          } if (|Q| = ⌈(n+1)/2⌉) {
45            return true; } } // PL: WR_SUCC
46    while (true); }

```

Fig. 10. Instrumented implementation of read and write methods.


```

1  process Acceptor(int j) {
2    val v := undef; int r := 0; int w := 0;
3    start() {
4      int i; msg m; int k;
5      do { (i, m) := receive();
6        switch (m) {
7          case [RE, k]:
8            if (k < r) { send(i, [nackRE, k]); }
9            else {  $\langle$  r := k;
10              if (abs_res_r[k] = undef) {
11                if (proph_r[k] = (true, v)) {
12                  if (count_r[k] =  $\lceil (n+1)/2 \rceil - 1$ ) {
13                    linRE(k, v, true); } } }
14              count_r[k]++; send(i, [ackRE, k, v, w]); } }
15          case [WR, k, vW]:
16            if (k < r) { send(j, i, [nackWR, k]); }
17            else {  $\langle$  r := k; w := k; v := vW;
18              if (abs_res_w[k] = undef) {
19                if (!(proph_w[k] = undef)) {
20                  if (proph_w[k]) {
21                    if (count_w[k] =  $\lceil (n+1)/2 \rceil - 1$ ) {
22                      linWR(k, vW, true); } }
23                  else { linWR(k, vW, false); } } }
24              count_w[k]++; send(j, i, [ackWR, k]); } }
25            } }
26    while (true); } }

```

Fig. 11. Instrumented implementation of acceptor processes.

of the specification on the right of Fig. 9. At the beginning of a `write(k, vW)` (lines 31–33 of Fig. 10), the prophecy `proph_r[k]` is set to `true` if the invocation reaches `PL:WR_SUCC` in line 45, or to `false` if it reaches `PL:WR_FAIL` in line 43 (or it is left `undef` if the method never returns). A successfully `write(k, vW)` linearises in the code of the acceptor in Fig. 11, when the $\lceil (n+1)/2 \rceil$ th acceptor sends `[ackWR, k]`, and only if the prophecy is `true` and the operation was not linearised before (lines 17–24). We force the unspecified `b` to be `true`, which ensures that the abstract operation succeeds deciding value `vW` and updates `roundRR` to `k`. A failing `write(k, vW)` may linearise either at the `return` in its own code (lines 41–43 of Fig. 10) if the proposer received one `[nackWR, k]` and no acceptor sent any `[ackWR, k]` yet, or at the code of the acceptor, when the first acceptor sends `[ackWR, k]`, and only if the prophecy is `false` and the operation was not linearised before. In both cases, we force the unspecified `b` to be `false`, which ensures that the abstract operation fails.

Theorem 3. *The implementation of Round-Based Register in Figs. 10 and 11 linearises with respect to its specification in Fig. 9.*

5 Multi-Paxos via Network Transformations

We now turn to more complicated distributed protocols that build upon the idea of Paxos consensus. Our ultimate goal is to reuse the verification result from the Sects. 3 and 4, as well as the high-level round-based register interface. In this section, we will demonstrate how to reason about an implementation of Multi-Paxos as of an array of *independent* instances of the *Paxos* module defined previously, despite the subtle dependencies between its sub-components, as present in Multi-Paxos’s “canonical” implementations [5, 15, 27]. While an abstraction of Multi-Paxos to an array of independent shared “single-shot” registers is almost folklore, what appears to be inherently difficult is to verify a Multi-Paxos-based consensus (*wrt.* to the array-based abstraction) by means of *reusing* the proof of a SD-Paxos. All proofs of Multi-Paxos we are aware of are, thus, *non-modular* with respect to underlying SD-Paxos instances [5, 22, 24], *i.e.*, they require one to redesign the invariants of the *entire* consensus protocol.

This proof modularity challenge stems from the optimised nature of a classical Multi-Paxos protocol, as well as its real-world implementations [6]. In this part of our work is to distil such protocol-aware optimisations into a separate *network semantics layer*, and show that each of them refines the semantics of a Cartesian product-based view, *i.e.*, exhibits the very same client-observable behaviours. To do so, we will establishing the refinement between the optimised implementations of Multi-Paxos and a simple Cartesian product abstraction, which will allow to extend the register-based abstraction, explored before in this paper, to what is considered to be a canonical amortised Multi-Paxos implementation.

5.1 Abstract Distributed Protocols

We start by presenting the formal definitions of encoding distributed protocols (including Paxos), their message vocabularies, protocol-based network semantics, and the notion of an observable behaviours.

Protocols and Messages. Figure 12 provides basic definitions of the distributed protocols and their components. Each protocol p is a tuple $\langle \Delta, \mathcal{M}, \mathcal{S}_{\text{int}}, \mathcal{S}_{\text{rcv}}, \mathcal{S}_{\text{snd}} \rangle$. Δ is a set of local states, which can be assigned to each of the participating nodes, also determining the node’s role via an additional tag,⁴ if necessary (*e.g.*, an acceptor and a proposer states in Paxos are different). \mathcal{M} is a “message vocabulary”, determining the set of messages that can be used for communication between the nodes.

Protocols	$\mathcal{P} \ni p \triangleq \langle \Delta, \mathcal{M}, \mathcal{S} \rangle$
Configurations	$\Sigma \ni \sigma \triangleq \text{Nodes} \rightarrow \Delta$
Internal steps	$\mathcal{S}_{\text{int}} \in \Delta \times \Delta$
Receive-steps	$\mathcal{S}_{\text{rcv}} \in \Delta \times \mathcal{M} \times \Delta$
Send-steps	$\mathcal{S}_{\text{snd}} \in \Delta \times \Delta \times \wp(\mathcal{M})$

Fig. 12. States and transitions.

⁴ We leave out implicit the consistency laws for the state, that are protocol-specific.

$$\begin{array}{c}
\text{STEPINT} \\
\frac{n \in \text{dom}(\sigma) \quad \delta = \sigma(n) \quad \langle \delta, \delta' \rangle \in p.\mathcal{S}_{\text{int}} \quad \sigma' = \sigma[n \mapsto \delta']}{\langle \sigma, M \rangle \xrightarrow[p_{\text{int}}]{p} \langle \sigma', M \rangle} \\
\\
\text{STEPSEND} \\
\frac{n \in \text{dom}(\sigma) \quad \delta = \sigma(n) \quad \langle \delta, \delta', \text{ms} \rangle \in p.\mathcal{S}_{\text{snd}} \quad \sigma' = \sigma[n \mapsto \delta'] \quad M' = M \cup \text{ms}}{\langle \sigma, M \rangle \xrightarrow[p_{\text{snd}}]{p} \langle \sigma', M' \rangle} \\
\\
\text{STEPRCEIVE} \\
\frac{m \in M \quad m.\text{active} \quad m.\text{to} \in \text{dom}(\sigma) \quad \delta = \sigma(m.\text{to}) \quad \langle \delta, m, \delta' \rangle \in p.\mathcal{S}_{\text{rcv}} \quad m' = m[\text{active} \mapsto \text{False}] \quad \sigma' = \sigma[n \mapsto \delta'] \quad M' = M \setminus \{m\} \cup \{m'\}}{\langle \sigma, M \rangle \xrightarrow[p_{\text{rcv}}]{p} \langle \sigma', M' \rangle}
\end{array}$$

Fig. 13. Transition rules of the simple protocol-aware network semantics

Messages can be thought of as JavaScript-like dictionaries, pairing unique fields (isomorphic to strings) with their values. For the sake of a uniform treatment, we assume that each message $m \in \mathcal{M}$ has at least two fields, *from* and *to* that point to the source and the destination node of a message, correspondingly. In addition to that, for simplicity we will assume that each message carries a Boolean field *active*, which is set to **True** when the message is sent and is set to **False** when the message is received by its destination node. This flag is required to keep history information about messages sent in the past, which is customary in frameworks for reasoning about distributed protocols [10, 23, 28]. We assume that a “message soup” M is a multiset of messages (*i.e.* a set with zero or more copies of each message) and we consider that each copy of the same message in the multiset has its own “identity”, and we write $m \neq m'$ to represent that m and m' are not the same copy of a particular message.

Finally, $\mathcal{S}_{\{\text{int}, \text{rcv}, \text{snd}\}}$ are step-relations that correspond to the internal changes in the local state of a node (\mathcal{S}_{int}), as well as changes associated with sending (\mathcal{S}_{snd}) and receiving (\mathcal{S}_{rcv}) messages by a node, as allowed by the protocol. Specifically, \mathcal{S}_{int} relates a local node state before and after the allowed internal change; \mathcal{S}_{rcv} relates the initial state and an incoming message $m \in \mathcal{M}$ with the resulting state; \mathcal{S}_{snd} relates the internal state, the output state and the set of atomically sent messages. For simplicity we will assume that $\text{id} \subseteq \mathcal{S}_{\text{int}}$.

In addition, we consider $\Delta_0 \subseteq \Delta$ —the set of the allowed *initial* states, in which the system can be present at the very beginning of its execution. The global state of the network $\sigma \in \Sigma$ is a map from node identifiers ($n \in \text{Nodes}$) to local states from the set of states Δ , defined by the protocol.

Simple Network Semantics. The simple initial operational semantics of the network ($\xRightarrow{p} \subseteq (\Sigma \times \wp(\mathcal{M})) \times (\Sigma \times \wp(\mathcal{M}))$) is parametrised by a protocol p and relates the initial *configuration* (*i.e.*, the global state and the set of messages) with the resulting configuration. It is defined via as a reflexive closure of the union of three relations $\xrightarrow[p_{\text{int}}]{p} \cup \xrightarrow[p_{\text{rcv}}]{p} \cup \xrightarrow[p_{\text{snd}}]{p}$, their rules are given in Fig. 13.

The rule **STEPINT** corresponds to a node n picked non-deterministically from the domain of a global state σ , executing an internal transition, thus changing its local state from δ to δ' . The rule **STEPRECEIVE** non-deterministically picks a m message from a message soup $M \subseteq \mathcal{M}$, changes the state using the protocol's receive-step relation $p.\mathcal{S}_{\text{rcv}}$ at the corresponding host node to , and updates its local state accordingly in the common mapping $(\sigma[to \mapsto \delta'])$. Finally, the rule **STEPSSEND**, non-deterministically picks a node n , executes a send-step, which results in updating its local state emission of a set of messages \mathbf{ms} , which is added to the resulting soup. In order to “bootstrap” the execution, the initial states from the set $\Delta_0 \subseteq \Delta$ are assigned to the nodes.

We next define the observable protocol behaviours *wrt.* the simple network semantics as the prefix-closed set of all system's configuration traces.

Definition 1. (Protocol behaviours)

$$\mathcal{B}_p = \bigcup_{m \in \mathbb{N}} \left\{ \langle \langle \sigma_0, M_0 \rangle, \dots, \langle \sigma_m, M_m \rangle \rangle \mid \begin{array}{l} \exists \delta_0^{n \in N} \in \Delta_0, \sigma_0 = \biguplus_{n \in N} [n \mapsto \delta_0^n] \wedge \\ \langle \sigma_0, M_0 \rangle \xRightarrow{p} \dots \xRightarrow{p} \langle \sigma_m, M_m \rangle \end{array} \right\}$$

That is, the set of behaviours captures all possible configurations of initial states for a fixed set of nodes $N \subseteq \text{Nodes}$. In this case, the set of nodes N is an implicit parameter of the definition, which we fix in the remainder of this section.

Example 1 (Encoding SD-Paxos). An abstract distributed protocol for SD-Paxos can be extracted from the pseudo-code of Sect. 3 by providing a suitable small-step operational semantics à la Winskel [30]. We restraint ourselves from giving such formal semantics, but in Appendix D of the extended version of the paper we outline how the distributed protocol would be obtained from the given operational semantics and from the code in Figs. 3, 4 and 5.

5.2 Out-of-Thin-Air Semantics

We now introduce an intermediate version of a simple protocol-aware semantics that generates messages “out of thin air” according to a certain predicate $\mathcal{P} \subseteq \Delta \times \mathcal{M}$, which determines whether the network generates a certain message without exercising the corresponding send-transition. The rule is as follows:

$$\frac{\text{OTASend} \quad n \in \text{dom}(\sigma) \quad \delta = \sigma(n) \quad \mathcal{P}(\delta, m) \quad M' = M \cup \{m\}}{\langle \sigma, M \rangle \xRightarrow[p, \mathcal{P}]{\text{ota}} \langle \sigma, M' \rangle}$$

That is, a random message m can be sent at any moment in the semantics described by $\xRightarrow{p} \cup \xRightarrow[p, \mathcal{P}]{\text{ota}}$, given that the node n , “on behalf of which” the message is sent is in a state δ , such that $\mathcal{P}(\delta, m)$ holds.

Example 2. In the context of Single-Decree Paxos, we can define \mathcal{P} as follows:

$$\mathcal{P}(\delta, m) \triangleq m.\text{content} = [\text{RE}, k] \wedge \delta.\text{pid} = n \wedge \delta.\text{role} = \text{Proposer} \wedge k \leq \delta.\text{kP}$$

In other words, if a node n is a *Proposer* currently operating with a round $\delta.\text{kP}$, the network semantics can always send another request “on its behalf”, thus generating the message “out-of-thin-air”. Importantly, the last conjunct in the definition of \mathcal{P} is in terms of \leq , rather than equality. This means that the predicate is intentionally loose, allowing for sending even “stale” messages, with expired rounds that are smaller than what n currently holds (no harm in that!).

By definition of single-decree Paxos protocol, the following lemma holds:

Lemma 1 (OTA refinement). $\mathcal{B} \xrightarrow[p \mapsto \cup]{p, \mathcal{P}} \subseteq \mathcal{B}_p$, where p is an instance of the module *Paxos*, as defined in Sect. 3 and in Example 1.

5.3 Slot-Replicating Network Semantics

With the basic definitions at hand, we now proceed to describing alternative network behaviours that make use of a specific protocol $p = \langle \Delta, \mathcal{M}, \mathcal{S}_{\text{int}}, \mathcal{S}_{\text{rcv}}, \mathcal{S}_{\text{snd}} \rangle$, which we will consider to be fixed for the remainder of this section, so we will be at times referring to its components (*e.g.*, \mathcal{S}_{int} , \mathcal{S}_{rcv} , *etc.*) without a qualifier.

$\begin{array}{c} \text{SRSTEPINT} \\ i \in I \quad n \in \text{dom}(\sigma) \\ \delta = \sigma(n)[i] \quad \langle \delta, \delta' \rangle \in p.\mathcal{S}_{\text{int}} \\ \sigma' = \sigma[n[i] \mapsto \delta'] \\ \hline \langle \sigma, M \rangle \xrightarrow[\text{int}]{x} \langle \sigma', M \rangle \end{array}$	$\begin{array}{c} \text{SRSTEPSEND} \\ i \in I \quad n \in \text{dom}(\sigma) \\ \delta = \sigma(n)[i] \quad \langle \delta, \delta', \text{ms} \rangle \in p.\mathcal{S}_{\text{snd}} \\ \sigma' = \sigma[n[i] \mapsto \delta'] \quad M' = M \cup \text{ms}[slot \mapsto i] \\ \hline \langle \sigma, M \rangle \xrightarrow[\text{snd}]{x} \langle \sigma', M' \rangle \end{array}$
$\begin{array}{c} \text{SRSTEPRECEIVE} \\ m \in M \quad m.\text{active} \quad m.\text{to} \in \text{dom}(\sigma) \quad \delta = \sigma(m.\text{to})[m.\text{slot}] \quad \langle \delta, m, \delta' \rangle \in p.\mathcal{S}_{\text{rcv}} \\ m' = m[\text{active} \mapsto \text{False}] \quad \sigma' = \sigma(n)[m.\text{slot} \mapsto \delta'] \quad M' = M \setminus \{m\} \cup \{m'\} \\ \hline \langle \sigma, M \rangle \xrightarrow[\text{rcv}]{x} \langle \sigma', M' \rangle \end{array}$	

Fig. 14. Transition rules of the slot-replicating network semantics.

Figure 14 describes a semantics of a *slot-replicating* (SR) network that exercises multiple copies of the *same* protocol instance p_i for $i \in I$, some, possibly infinite, set of indices, to which we will be also referring as *slots*. Multiple copies of the protocol are incorporated by enhancing the messages from p ’s vocabulary \mathcal{M} with the corresponding indices, and implementing the on-site dispatch of the indexed messages to corresponding protocol instances at each node. The local protocol state of each node is, thus, no longer a single element being updated,

but rather an *array*, mapping $i \in I$ into δ_i —the corresponding local state component. The small-step relation for SR semantics is denoted by $\xRightarrow{\times}$. The rule SRSTEPINT is similar to STEPINT of the simple semantics, with the difference that it picks not only a node but also an index i , thus referring to a specific component $\sigma(n)[i]$ as δ and updating it correspondingly ($\sigma(n)[i] \mapsto \delta'$). For the remaining transitions, we postulate that the messages from p 's vocabulary $p.\mathcal{M}$ are enhanced to have a dedicated field *slot*, which indicates a protocol copy at a node, to which the message is directed. The receive-rule SRSTEPRECEIVE is similar to STEPRECEIVE but takes into the account the value of $m.slot$ in the received message m , thus redirecting it to the corresponding protocol instance and updating the local state appropriately. Finally, the rule SRSTEPSEND can be now executed for any slot $i \in I$, reusing most of the logic of the initial protocol and otherwise mimicking its simple network semantic counterpart STEPSSEND.

Importantly, in this semantics, for two different slots i, j , such that $i \neq j$, the corresponding “projections” of the state behave *independently* from each other. Therefore, transitions and messages in the protocol instances indexed by i at different nodes *do not interfere* with those indexed by j . This observation can be stated formally. In order to do so we first defined the behaviours of slot-replicating networks and their projections as follows:

Definition 2 (Slot-replicating protocol behaviours).

$$\mathcal{B}_{\times} = \bigcup_{m \in \mathbb{N}} \left\{ \langle \langle \sigma_0, M_0 \rangle, \dots, \langle \sigma_m, M_m \rangle \rangle \left| \begin{array}{l} \exists \delta_0^{n \in N} \in \Delta_0, \\ \sigma_0 = \biguplus_{n \in N} [n \mapsto \{i \mapsto \delta_0^n \mid i \in I\}] \wedge \\ \langle \sigma_0, M_0 \rangle \xRightarrow{p} \dots \xRightarrow{p} \langle \sigma_m, M_m \rangle \end{array} \right. \right\}$$

That is, the slot-replicated behaviours are merely behaviours with respect to networks, whose nodes hold *multiple instances* of the same protocol, indexed by slots $i \in I$. For a slot $i \in I$, we define *projection* $\mathcal{B}_{\times}|_i$ as a set of global state traces, where each node's local states is restricted only to its i th component. The following simulation lemma holds naturally, connecting the state-replicating network semantics and simple network semantics.

Lemma 2 (Slot-replicating simulation). *For all $I, i \in I$, $\mathcal{B}_{\times}|_i = \mathcal{B}_p$.*

Example 3 (Slot-replicating semantics and Paxos). Given our representation of Paxos using roles (acceptors/proposers) encoded via the corresponding parts of the local state δ , we can construct a “naïve” version of Multi-Paxos by using the SR semantics for the protocol. In such, every slot will correspond to a SD-Paxos instance, not interacting with any other slots. From the practical perspective, such an implementation is rather non-optimal, as it does not exploit dependencies between rounds accepted at different slots.

5.4 Widening Network Semantics

We next consider a version of the SR semantics, extended with a new rule for handling received messages. In the new semantics, dubbed *widening*, a node, upon receiving a message $m \in T$, where $T \subseteq p.\mathcal{M}$, for a slot i , replicates it for all slots from the index set I , for the very same node. The new rule is as follows:

$$\frac{\text{WSTEPRECEIVET} \quad \begin{array}{l} m \in M \quad m.\text{active} \quad m.\text{to} \in \text{dom}(\sigma) \quad \delta = \sigma(m.\text{to})[m.\text{slot}] \\ \langle \delta, m, \delta' \rangle \in p.\mathcal{S}_{\text{rev}} \quad m' = m[\text{active} \mapsto \text{False}] \quad \sigma' = \sigma(n)[m.\text{slot} \mapsto \delta'] \\ \text{ms} = \text{if } (m \in T) \text{ then } \{m' \mid m' = m[\text{slot} \mapsto j], j \in I\} \text{ else } \emptyset \end{array}}{\langle \sigma, M \rangle \xrightarrow[\text{rcv}]{\nabla} \langle \sigma', (M \setminus \{m\}) \cup \{m'\} \cup \text{ms} \rangle}$$

At first, this semantics seems rather unreasonable: it might create more messages than the system can “consume”. However, it is possible to prove that, under certain conditions on the protocol p , the set of behaviours observed under this semantics (*i.e.*, with SRSTEPRECEIVE replaced by WSTEPRECEIVET) is *not larger* than \mathcal{B}_\times as given by Definition 2. To state this formally we first relate the set of “triggering” messages T from WSTEPRECEIVET to a specific predicate \mathcal{P} .

Definition 3 (OTA-compliant message sets). The set of messages $T \subseteq p.\mathcal{M}$ is OTA-compliant with the predicate \mathcal{P} iff for any $b \in \mathcal{B}_p$ and $\langle \sigma, M \rangle \in b$, if $m \in M$, then $\mathcal{P}(\sigma(m.\text{from}), m)$.

In other words, the protocol p is relaxed enough to “justify” the presence of m in the soup at *any* execution, by providing the predicate \mathcal{P} , relating the message to the corresponding sender’s state. Next, we use this definition to slot-replicating and widening semantics via the following definition.

Definition 4 (\mathcal{P} -monotone protocols). A protocol p is \mathcal{P} -monotone iff for any, $b \in \mathcal{B}_\times$, $\langle \sigma, M \rangle \in b$, $m, i = m.\text{slot}$, and $j \neq i$, if $\mathcal{P}(\sigma(m.\text{from})[i], \mathfrak{h}m)$ then we have that $\mathcal{P}(\sigma(m.\text{from})[j], \mathfrak{h}m)$, where $\mathfrak{h}m$ “removes” the *slot* field from m .

Less formally, Definition 4 ensures that in a slot-replicated product \times of a protocol p , different components cannot perform “out of sync” *wrt.* \mathcal{P} . Specifically, if a node in i th projection is related to a certain message $\mathfrak{h}m$ via \mathcal{P} , then any other projection j of the same node will be \mathcal{P} -related to this message, as well.

Example 4. This is a “non-example”. A version of slot-replicated SD-Paxos, where we allow for arbitrary increments of the round *per-slot* at a same proposer node (*i.e.*, out of sync), would not be monotone *wrt.* \mathcal{P} from Example 2. In contrast, a slot-replicated product of SD-Paxos instances with fixed rounds is monotone *wrt.* the same \mathcal{P} .

Lemma 3. If T from WSTEPRECEIVET is OTA-compliant with predicate \mathcal{P} , such that $\mathcal{B} \xrightarrow[p]{p, \mathcal{P}} \subseteq \mathcal{B} \xrightarrow[p]{p}$ and p is \mathcal{P} -monotone, then $\mathcal{B} \xrightarrow[\text{ota}]{\nabla} \subseteq \mathcal{B} \xrightarrow[\times]{\nabla}$.

Example 5 (Widening semantics and Paxos). The SD-Paxos instance as described in Sect. 3 satisfies the refinement condition from Lemma 3. By taking $T = \{m \mid m = \{\text{content} = [\text{RE}, \mathbf{k}]; \dots\}\}$ and using Lemma 3, we obtain the refinement between widened semantics and SR semantics of Paxos.

5.5 Optimised Widening Semantics

Our next step towards a realistic implementation of Multi-Paxos out of SD-Paxos instances is enabled by an observation that in the widening semantics, the replicated messages are *always* targeting the same node, to which the initial message $m \in T$ was addressed. This means that we can optimise the receive-step, making it possible to execute multiple receive-transitions of the core protocol in batch. The following rule **OWSTEPRECEIVET** captures this intuition formally:

$$\frac{\text{OWSTEPRECEIVET} \quad m \in M \quad m.\text{active} \quad m.\text{to} \in \text{dom}(\sigma) \quad \langle \sigma', \text{ms} \rangle = \text{receiveAndAct}(\sigma, n, m)}{\langle \sigma, M \rangle \xrightarrow[\text{rcv}]{\nabla^*} \langle \sigma', M \setminus \{m\} \cup \{m[\text{active} \mapsto \text{False}]\} \cup \text{ms} \rangle}$$

where $\text{receiveAndAct}(\sigma, n, m) \triangleq \langle \sigma', \text{ms} \rangle$, such that $\text{ms} = \bigcup_j \{m[\text{slot} \mapsto j] \mid m \in \text{ms}_j\}$, $\forall j \in I, \delta = \sigma(m.\text{to})[j] \wedge \langle \delta_j, \sharp m, \delta_j^1 \rangle \in p.\mathcal{S}_{\text{rcv}} \wedge \langle \delta_j^1, \delta_j^2 \rangle \in p.\mathcal{S}_{\text{int}}^* \wedge \langle \delta_j^2, \delta_j^3, \text{ms}_j \rangle \in p.\mathcal{S}_{\text{snd}}$, $\forall j \in I, \sigma'(m.\text{to})[j] = \delta_j^3$.

In essence, the rule **OWSTEPRECEIVET** blends several steps of the widening semantics together for a single message: (a) it first receives the message and replicates it for all slots at a destination node; (b) performs receive-steps for the message's replicas at each slot; (c) takes a number of internal steps, allowed by the protocol's \mathcal{S}_{int} ; and (d) takes a send-transition, eventually sending all emitted message, instrumented with the corresponding slots.

Example 6. Continuing Example 5, with the same parameters, the optimising semantics will execute the transitions of an acceptor, *for all slots*, triggered by receiving a single **[RE, k]** message for a particular slot, sending back *all* the results for all the slots, which might either agree to accept the value or reject it.

The following lemma relates the optimising and the widening semantics.

Lemma 4 (Refinement for OW semantics). *For any $b \in \mathcal{B}_{\nabla^*}$ there exists $b' \in \mathcal{B}_{\nabla}$, such that b can be obtained from b' by replacing sequences of configurations $[\langle \sigma_k, M_k \rangle, \dots, \langle \sigma_{k+m}, M_{k+m} \rangle]$ that have just a single node n , whose local state is affected in $\sigma_k, \dots, \sigma_{k+m}$, by $[\langle \sigma_k, M_k \rangle, \langle \sigma_{k+m}, M_{k+m} \rangle]$.*

That is, behaviours in the optimised semantics are the same as in the widening semantics, modulo some sequences of locally taken steps that are being “compressed” to just the initial and the final configurations.

5.6 Bunching Semantics

As the last step towards Multi-Paxos, we introduce the final network semantics that optimises executions according to ∇^* described in previous section even further by making a simple addition to the message vocabulary of a slot-replicated SD-Paxos—*bunched messages*. A bunched message simply packages

$$\begin{array}{c}
\text{BSTEPRECVB} \\
\frac{
\begin{array}{l}
m \in M \quad m.\text{active} \quad m.\text{to} \in \text{dom}(\sigma) \\
\langle \sigma', \text{ms} \rangle = \text{receiveAndAct}(\sigma, n, m) \\
M' = M \setminus \{m\} \cup \{m[\text{active} \mapsto \text{False}]\} \\
m' = \text{bunch}(\text{ms}, m.\text{to}, m.\text{from})
\end{array}
}{
\langle \sigma, M \rangle \xrightarrow[\text{rcv}]{B} \langle \sigma', M' \cup \{m'\} \rangle
}
\end{array}
\qquad
\begin{array}{c}
\text{BSTEPRECVU} \\
\frac{
\begin{array}{l}
m \in M \quad m.\text{active} \quad m.\text{to} \in \text{dom}(\sigma) \\
m.\text{msgs} = \text{ms} \quad M' = M \setminus \{m\} \cup \text{ms}
\end{array}
}{
\langle \sigma, M \rangle \xrightarrow[\text{rcv}]{B} \langle \sigma, M' \rangle
}
\end{array}$$

where $\text{bunch}(\text{ms}, n_1, n_2) = \{\text{msgs} = \text{ms}; \text{from} = n_1; \text{to} = n_2; \text{active} = \text{True}\}$.

Fig. 15. Added rules of the Bunching Semantics

together several messages, obtained typically as a result of a “compressed” execution via the optimised semantics from Sect. 5.5. We define two new rules for packaging and “unpacking” certain messages in Fig. 15. The two new rules can be added to enhance either of the versions of the slot-replicating semantics shown before. In essence, the only effect they have is to combine the messages resulting in the execution of the corresponding steps of an optimised widening (via BSTEPRECVB), and to unpackage the messages ms from a bunching message, adding them back to the soup (BSTEPRECVU). The following natural refinement result holds:

Lemma 5. *For any $b \in \mathcal{B}_{\xrightarrow{B}}$ there exists $b' \in \mathcal{B}_{\xrightarrow{\nabla^*}}$, such that b' can be obtained from b by replacing all bunched messages in b by their msgs -component.*

The rule BSTEPRECVU enables effective local caching of the bunched messages, so they are processed *on demand* on the recipient side (*i.e.*, by the per-slot proposers), allowing the implementation to *skip* an entire round of Phase 1.

$$\begin{array}{ccccc}
(\xrightarrow{B}) & & (\xrightarrow[\text{ota}]{P}) & \text{via Lm 1 refines} & (\xrightarrow{P}) \\
\text{via Lm 5 refines} & & \text{sim. via Lm 2} & & \text{sim. via Lm 2} \\
(\xrightarrow{\nabla^*}) & \text{via Lm 4 refines} & (\xrightarrow{\nabla}) & \text{via Lm 3 refines} & (\xrightarrow{\times})
\end{array}$$

Fig. 16. Refinement between different network semantics.

```

1 proposeM(val^ v, val v0) {           5 val vM[1..∞] := undef;
2   { assume(!(v0 = undef));           6 getR(int s) { return &(vM[s]); }
3   if (*v = undef) { *v := v0; }      7 proposeM(getR(1), v);
4   return *v; }                       8 proposeM(getR(2), v);

```

Fig. 17. Specification of *Multi-Paxos* and interaction via a *register provider*.

5.7 The Big Picture

What exactly have we achieved by introducing the described above family of semantics? As illustrated in Fig. 16, all behaviours of the leftmost-topmost, bunching semantics, which corresponds precisely to an implementation of Multi-Paxos with an “amortised” Phase 1, can be transitively related to the corresponding behaviours in the rightmost, vanilla slot-replicated version of a simple semantics (via the correspondence from Lemma 1) by constructing the corresponding refinement mappings [1], delivered by the proofs of Lemmas 3–5.

From the perspective of Rely/Guarantee reasoning, which was employed in Sect. 4, the refinement result from Fig. 16 justifies the replacement of a semantics on the right of the diagram by one to the left of it, as all program-level assertions will remain substantiated by the corresponding system configurations, as long as they are *stable* (i.e., resilient *wrt.* transitions taken by nodes different from the one being verified), which they are in our case.

6 Putting It All Together

We culminate our story of faithfully deconstructing and abstracting Paxos via a round-based register, as well as recasting Multi-Paxos via a series of network transformations, by showing how to *implement* the register-based abstraction from Sect. 3 in tandem with the network semantics from Sect. 5 in order to deliver provably correct, yet efficient, implementation of Multi-Paxos.

The crux of the composition of the two results—a register-based abstraction of SD-Paxos and a family of semantics-preserving network transformations—is a convenient interface for the end client, so she could interact with a consensus instance via the `proposeM` method in lines 1–4 of Fig. 17, no matter with which particular slot of a Multi-Paxos implementation she is interacting. To do so, we propose to introduce a *register provider*—a service that would give a client a “reference” to the consensus object to interact with. Lines 6–7 of Fig. 17 illustrate the interaction with the service provider, where the client requests two specific slots, 1 and 2, of Multi-Paxos by invoking `getR` and providing a slot parameter. In both cases the client proposes the very same value v in the two instances that run the same machinery. (Notice that, except for the reference to the consensus object, `proposeM` is identical to the `proposeP` on the right of Fig. 2, which we have verified *wrt.* linearisability in Sect. 3.)

The implementation of Multi-Paxos that we have in mind resembles the one in Figs. 3, 4 and 5 of Sect. 3, but where all the global data is provided by the register provider and passed by reference. What differs in this implementation with respect to the one in Sect. 3 and is hidden from the client is the semantics of the network layer used by the bottom layer (*cf.* left part of Fig. 2) of the register-based implementation. The Multi-Paxos instances run (without changing the register’s code) over this network layer, which “overloads” the meaning of the `send/receive` primitives from Figs. 3 and 4 to follow the bunching network semantics, described in Sect. 5.6.

Theorem 4. *The implementation of Multi-Paxos that uses a register provider and bunching network semantics refines the specification in Fig. 17.*

We implemented the register/network semantics in a proof-of-concept prototype written in Scala/Akka.⁵ We relied on the abstraction mechanisms of Scala, allowing us to implement the register logic, verified in Sect. 4, separately from the network middle-ware, which has provided a family of Semantics from Sect. 5. Together, they provide a family of provably correct, modularly verified *distributed* implementations, coming with a simple *shared memory-like* interface.

7 Related Work

Proofs of Linearisability via Rely/Guarantee. Our work builds on the results of Boichat *et al.* [3], who were first to propose to a systematic deconstruction of Paxos into read/write operations of a *round-based register* abstraction. We extend and harness those abstractions, by intentionally introducing more non-determinism into them, which allows us to provide the first modular (*i.e.*, mutually independent) proofs of Proposer and Acceptor using Rely/Guarantee with linearisation points and prophecies. While several logics have been proposed recently to prove linearisability of concurrent implementations using Rely/Guarantee reasoning [14, 18, 19, 26], none of them considers message-passing distributed systems or consensus protocols.

Verification of Paxos-Family Algorithms. Formal verification of different versions of Paxos-family protocols *wrt.* inductive invariants and liveness has been a focus of multiple verification efforts in the past fifteen years. To name just a few, Lamport has specified and verified Fast Paxos [17] using TLA+ and its accompanying model checker [32]. Chand *et al.* used TLA+ to specify and verify Multi-Paxos implementation, similar to the one we considered in this work [5]. A version of SD-Paxos has been verified by Kellomaki using the PVS theorem prover [13]. Jaskelioff and Merz have verified Disk Paxos in Isabelle/HOL [12]. More recently, Rahli *et al.* formalised an executable version of Multi-Paxos in EventML [24], a dialect of NuPRL. Dragoi *et al.* [8] implemented and verified SD-Paxos in the PSYNC framework, which implements a partially synchronised model [7], supporting automated proofs of system invariants. Padon *et al.* have proved the system invariants and the consensus property of both simple Paxos and Multi-Paxos using the verification tool IVY [22, 23].

Unlike all those verification efforts that consider (Multi-/Disk/Fast/...)Paxos as a *single monolithic protocol*, our approach provides the first *modular* verification of single-decree Paxos using Rely/Guarantee framework, as well as the first verification of Multi-Paxos that directly reuses the proof of SD-Paxos.

⁵ The code is available at <https://github.com/certichain/protocol-combinators>.

Compositional Reasoning about Distributed Systems. Several recent works have partially addressed modular formal verification of distributed systems. The IronFleet framework by Hawblitzel *et al.* has been used to verify both safety and liveness of a real-world implementation of a Paxos-based replicated state machine library and a lease-based shared key-value store [10]. While the proof is structured in a modular way by composing specifications in a way similar to our decomposition in Sects. 3 and 4, that work does not address the linearisability and does not provide composition of proofs about complex protocols (*e.g.*, Multi-Paxos) from proofs about its subparts

The Verdi framework for deductive verification of distributed systems [29, 31] suggests the idea of *Verified System Transformers* (VSTs), as a way to provide *vertical composition* of distributed system implementation. While Verdi’s VSTs are similar in its purpose and idea to our network transformations, they *do not* exploit the properties of the protocol, which was crucial for us to verify Multi-Paxos’s implementation.

The DIESEL framework [25, 28] addresses the problem of *horizontal composition* of distributed protocols and their client applications. While we do not compose Paxos with any clients in this work, we believe its register-based specification could be directly employed for verifying applications that use Paxos as its sub-component, which is what is demonstrated by our prototype implementation.

8 Conclusion and Future Work

We have proposed and explored two complementary mechanisms for modular verification of Paxos-family consensus protocols [15]: (a) non-deterministic register-based specifications in the style of Boichat *et al.* [3], which allow one to decompose the proof of protocol’s linearisability into separate independent “layers”, and (b) a family of protocol-aware transformations of network semantics, making it possible to reuse the verification efforts. We believe that the applicability of these mechanisms spreads beyond reasoning about Paxos and its variants and that they can be used for verifying other consensus protocols, such as Raft [21] and PBFT [4]. We are also going to employ network transformations to verify implementations of Mencius [20], and accommodate more protocol-specific optimisations, such as implementation of master leases and epoch numbering [6].

Acknowledgements. We thank the ESOP 2018 reviewers for their feedback. This work by was supported by ERC Starting Grant H2020-EU 714729 and EPSRC First Grant EP/P009271/1.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. In: LICS, pp. 165–175. IEEE Computer Society (1988)
2. Boichat, R., Dutta, P., Frølund, S., Guerraoui, R.: Deconstructing Paxos (2001). OAIPMH server at infoscience.epfl.ch, record 52373. <http://infoscience.epfl.ch/record/52373>

3. Boichat, R., Dutta, P., Frølund, S., Guerraoui, R.: Deconstructing Paxos. *SIGACT News* **34**(1), 47–67 (2003)
4. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: OSDI, pp. 173–186. USENIX Association (1999)
5. 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
6. Chandra, T., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: PODC, pp. 398–407. ACM (2007)
7. Charron-Bost, B., Merz, S.: Formal verification of a consensus algorithm in the heard-of model. *Int. J. Softw. Inform.* **3**(2–3), 273–303 (2009)
8. Dragoi, C., Henzinger, T.A., Zufferey, D.: PSync: a partially synchronous language for fault-tolerant distributed algorithms. In: POPL, pp. 400–415. ACM (2016)
9. Filipovic, I., O’Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. *Theor. Comput. Sci.* **411**(51–52), 4379–4398 (2010)
10. 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, pp. 1–17. ACM (2015)
11. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **12**(3), 463–492 (1990)
12. Jaskelioff, M., Merz, S.: Proving the correctness of disk Paxos. *Archive of Formal Proofs* (2005)
13. Kellomäki, P.: An annotated specification of the consensus protocol of Paxos using superposition in PVS. Technical report 36, Tampere University of Technology, Institute of Software Systems (2004)
14. Khyzha, A., Gotsman, A., Parkinson, M.: A generic logic for proving linearizability. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 426–443. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_26
15. Lamport, L.: The part-time parliament. *ACM Trans. Comput. Syst.* **16**(2), 133–169 (1998)
16. Lamport, L.: Paxos made simple. *SIGACT News* **32**, 18–25 (2001)
17. Lamport, L.: Fast Paxos. *Distrib. Comput.* **19**(2), 79–103 (2006)
18. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: PLDI, pp. 459–470. ACM (2013)
19. Liang, H., Feng, X.: A program logic for concurrent objects under fair scheduling. In: POPL, pp. 385–399. ACM (2016)
20. Mao, Y., Junqueira, F.P., Marzullo, K.: Mencius: building efficient replicated state machine for WANs. In: OSDI, pp. 369–384. USENIX Association (2008)
21. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference, pp. 305–319 (2014)
22. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL* **1**(OOPSLA), 108:1–108:31 (2017)
23. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: PLDI, pp. 614–630. ACM (2016)
24. Rahli, V., Guaspari, D., Bickford, M., Constable, R.L.: Formal specification, verification, and implementation of fault-tolerant systems using EventML. In: AVOCs. EASST (2015)
25. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. *PACMPL* **2**(POPL), 28:1–28:30 (2018)

26. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge (2007)
27. van Renesse, R., Altinbuken, D.: Paxos made moderately complex. *ACM Comput. Surv.* **47**(3), 42:1–42:36 (2015)
28. Wilcox, J.R., Sergey, I., Tatlock, Z.: Programming language abstractions for modularly verified distributed systems. In: *SNAPL. LIPIcs*, vol. 71, pp. 19:1–19:12. Schloss Dagstuhl (2017)
29. 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*, pp. 357–368. ACM (2015)
30. Winskel, G.: *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge (1993)
31. 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*, pp. 154–165. ACM (2016)
32. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA^+ specifications. In: Pierre, L., Kropf, T. (eds.) *CHARME 1999. LNCS*, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





On Parallel Snapshot Isolation and Release/Acquire Consistency

Azalea Raad¹(✉), Ori Lahav², and Viktor Vafeiadis¹

¹ MPI-SWS, Kaiserslautern, Germany
{azalea,viktor}@mpi-sws.org

² Tel Aviv University, Tel Aviv, Israel
orilahav@tau.ac.il

Abstract. Parallel snapshot isolation (PSI) is a standard transactional consistency model used in databases and distributed systems. We argue that PSI is also a useful formal model for software transactional memory (STM) as it has certain advantages over other consistency models. However, the formal PSI definition is given declaratively by acyclicity axioms, which most programmers find hard to understand and reason about.

To address this, we develop a simple lock-based reference implementation for PSI built on top of the release-acquire memory model, a well-behaved subset of the C/C++11 memory model. We prove that our implementation is sound and complete against its higher-level declarative specification.

We further consider an extension of PSI allowing transactional and non-transactional code to interact, and provide a sound and complete reference implementation for the more general setting. Supporting this interaction is necessary for adopting a transactional model in programming languages.

1 Introduction

Following the widespread use of transactions in databases, *software transactional memory* (STM) [19,35] has been proposed as a programming language abstraction that can radically simplify the task of writing correct and efficient concurrent programs. It provides the illusion of blocks of code, called *transactions*, executing atomically and in isolation from any other such concurrent blocks.

In theory, STM is great for programmers as it allows them to concentrate on the high-level algorithmic steps of solving a problem and relieves them of such concerns as the low-level details of enforcing mutual exclusion. In practice, however, the situation is far from ideal as the semantics of transactions in the context of non-transactional code is not at all settled. Recent years have seen a plethora of different STM implementations [1–3,6,17,20], each providing a slightly different—and often unspecified—semantics to the programmer.

Simple models in the literature are lock-based, such as *global lock atomicity* (GLA) [28] (where a transaction must acquire a global lock prior to execution and

release it afterwards) and *disjoint lock atomicity* (DLA) [28] (where a transaction must acquire all locks associated with the locations it accesses prior to execution and release them afterwards), which provide *serialisable* transactions. That is, all transactions appear to have executed atomically one after another in some total order. The problem with these models is largely their implementation cost, as they impose too much synchronisation between transactions.

The database community has long recognised this performance problem and has developed weaker transactional models that do not guarantee serialisability. The most widely used such model is *snapshot isolation* (SI) [10], implemented by major databases, both centralised (e.g. Oracle and MS SQL Server) and distributed [16, 30, 33], as well as in STM [1, 11, 25, 26]. In this article, we focus on a closely related model, *parallel snapshot isolation* (PSI) [36], which is known to provide better scalability and availability in large-scale geo-replicated systems. SI and PSI allow conflicting transactions to execute concurrently and to commit successfully, so long as they do not have a write-write conflict. This in effect allows reads of SI/PSI transactions to read from an earlier memory snapshot than the one affected by their writes, and permits outcomes such as the following:

$$\begin{array}{c} \text{Initially, } x = y = 0 \\ \mathbf{T1:} \left[\begin{array}{l} x := 1; \\ a := y; \text{ //reads } 0 \end{array} \right] \parallel \mathbf{T2:} \left[\begin{array}{l} y := 1; \\ b := x; \text{ //reads } 0 \end{array} \right] \quad (\text{SB+txs}) \end{array}$$

The above is also known as the *write skew* anomaly in the database literature [14]. Such outcomes are analogous to those allowed by weak memory models, such as x86-TSO [29, 34] and C11 [9], for non-transactional programs. In this article, we consider—to the best of our knowledge for the first time—PSI as a possible model for STM, especially in the context of a concurrent language such as C/C++ with a weak memory model. In such contexts, programmers are already familiar with weak behaviours such as that exhibited by SB+txs above.

A key reason why PSI is more suitable for a programming language than SI (or other stronger models) is *performance*. This is analogous to why C/C++ adopted non-multi-copy-atomicity (allowing two different threads to observe a write by a third thread at different times) as part of their concurrency model. Consider the following “IRIW” (independent reads of independent writes) litmus test:

$$\begin{array}{c} \text{Initially, } x = y = 0 \\ \mathbf{T1:} \left[\begin{array}{l} x := 1; \end{array} \right] \parallel \mathbf{T2:} \left[\begin{array}{l} a := x; \text{ //reads } 0 \\ b := y; \text{ //reads } 0 \end{array} \right] \parallel \mathbf{T3:} \left[\begin{array}{l} c := y; \text{ //reads } 0 \\ d := x; \text{ //reads } 0 \end{array} \right] \parallel \mathbf{T4:} \left[\begin{array}{l} y := 1; \end{array} \right] \quad (\text{IRIW+txs}) \end{array}$$

In the annotated behaviour, transactions T2 and T3 disagree on the relative order of transactions T1 and T4. Under PSI, this behaviour (called the *long fork anomaly*) is allowed, as T1 and T4 are not ordered—they commit in parallel—but it is disallowed under SI. This intuitively means that SI must impose ordering guarantees even on transactions that do not access a common location, and can be rather costly in the context of a weakly consistent system.

A second reason why PSI is much more suitable than SI is that it has better properties. A key intuitive property a programmer might expect of transactions is *monotonicity*. Suppose, in the (SB+txs) program we split the two transactions into four smaller ones as follows:

$$\begin{array}{l} \text{Initially, } x = y = 0 \\ \text{T1: } [x := 1; \quad \quad \quad \parallel \quad \text{T2: } [y := 1; \quad \quad \quad \text{(SB+txs+chop)} \\ \text{T3: } [a := y; \text{ //reads } 0 \quad \parallel \quad \text{T4: } [b := x; \text{ //reads } 0 \end{array}$$

One might expect that if the annotated behaviour is allowed in (SB+txs), it should also be allowed in (SB+txs+chop). This indeed is the case for PSI, but not for SI! In fact, in the extreme case where every transaction contains a single access, SI provides serialisability. Nevertheless, PSI currently has two significant drawbacks, preventing its widespread adoption. We aim to address these here.

The first PSI drawback is that its formal semantics can be rather daunting for the uninitiated as it is defined declaratively in terms of acyclicity constraints. What is missing is perhaps a simple lock-based reference implementation of PSI, similar to the lock-based implementations of GLA and DLA, that the programmers can readily understand and reason about. As an added benefit, such an implementation can be viewed as an operational model, forming the basis for developing program logics for reasoning about PSI programs.

Although Cerone et al. [15] proved their declarative PSI specification equivalent to an implementation strategy of PSI in a distributed system with replicated storage over causal consistency, their implementation is not suitable for reasoning about *shared-memory* programs. In particular, it cannot help the programmers determine how transactional and non-transactional accesses may interact.

As our first contribution, in Sect. 4 we address this PSI drawback by providing a simple lock-based reference implementation that we prove equivalent to its declarative specification. Typically, one proves that an implementation is *sound* with respect to a declarative specification—i.e. every behaviour observable in the implementation is accounted for in the declarative specification. Here, we also want the other direction, known as *completeness*, namely that every behaviour allowed by the specification is actually possible in the implementation. Having a (simple) complete implementation is very useful for programmers, as it may be easier to understand and experiment with than the declarative specification.

Our reference implementation is built in the *release-acquire* fragment of the C/C++ memory model [8, 9, 21], using sequence locks [13, 18, 23, 32] to achieve the correct transactional semantics.

The second PSI drawback is that its study so far has not accounted for the subtle effects of non-transactional accesses and how they interact with transactional accesses. While this scenario does not arise in ‘closed world’ systems such as databases, it is crucially important in languages such as C/C++ and Java, where one cannot afford the implementation cost of making every access transactional so that it is “strongly isolated” from other concurrent transactions.

Therefore, as our second contribution, in Sect. 5 we extend our basic reference implementation to make it robust under uninstrumented non-transactional

accesses, and characterise declaratively the semantics we obtain. We call this extended model RPSI (for “robust PSI”) and show that it gives reasonable semantics even under scenarios where transactional and non-transactional accesses are mixed.

Outline. The remainder of this article is organised as follows. In Sect. 2 we present an overview of our contributions and the necessary background information. In Sect. 3 we provide the formal model of the C11 release/acquire fragment and describe how we extend it to specify the behaviour of STM programs. In Sect. 4 we present our PSI reference implementation (without non-transactional accesses), demonstrating its soundness and completeness against the declarative PSI specification. In Sect. 5 we formulate a declarative specification for RPSI as an extension of PSI accounting for non-transactional accesses. We then present our RPSI reference implementation, demonstrating its soundness and completeness against our proposed declarative specification. We conclude and discuss future work in Sect. 6.

2 Background and Main Ideas

One of the main differences between the specification of database transactions and those of STM is that STM specifications must additionally account for the interactions between *mixed-mode* (both transactional and non-transactional) accesses to the same locations. To characterise such interactions, Blundell et al. [12, 27] proposed the notions of *weak* and *strong atomicity*, often referred to as weak and strong isolation. Weak isolation guarantees isolation only amongst transactions: the intermediate state of a transaction cannot affect or be affected by other transactions, but no such isolation is guaranteed with respect to non-transactional code (e.g. the accesses of a transaction may be interleaved by those of non-transactional code.). By contrast, strong isolation additionally guarantees full isolation from non-transactional code. Informally, each non-transactional access is considered as a transaction with a single access. In what follows, we explore the design choices for implementing STMs under each isolation model (Sect. 2.1), provide an intuitive account of the PSI model (Sect. 2.2), and describe the key requirements for implementing PSI and how we meet them (Sect. 2.3).

2.1 Implementing Software Transactional Memory

Implementing STMs under either strong or weak isolation models comes with a number of challenges. Implementing strongly isolated STMs requires a conflict detection/avoidance mechanism between transactional and non-transactional code. That is, unless non-transactional accesses are instrumented to adhere to the same access policies, conflicts involving non-transactional code cannot be detected. For instance, in order to guarantee strong isolation under the GLA model [28] discussed earlier, non-transactional code must be modified to acquire the global lock prior to each shared access and release it afterwards.

Implementing weakly-isolated STMs requires a careful handling of aborting transactions as their intermediate state may be observed by non-transactional code. Ideally, the STM implementation must ensure that the intermediate state of aborting transactions is not leaked to non-transactional code. A transaction may abort either because it failed to commit (e.g. due to a conflict), or because it encountered an explicit abort instruction in the transactional code. In the former case, leaks to non-transactional code can be avoided by pessimistic concurrency control (e.g. locks), pre-empting conflicts. In the latter case, leaks can be prevented either by lazy version management (where transactional updates are stored locally and propagated to memory only upon committing), or by disallowing explicit abort instructions altogether – an approach taken by the (weakly isolated) relaxed transactions of the C++ memory model [6].

As mentioned earlier, our aim in this work is to build an STM with PSI guarantees in the RA fragment of C11. As such, instrumenting non-transactional accesses is not feasible and thus our STM guarantees weak isolation. For simplicity, throughout our development we make a few simplifying assumptions: (i) transactions are not nested; (ii) the transactional code is without explicit abort instructions (as with the weakly-isolated transactions of C++ [6]); and (iii) the locations accessed by a transaction can be statically determined. For the latter, of course, a static over-approximation of the locations accessed suffices for the soundness of our implementations.

2.2 Parallel Snapshot Isolation (PSI)

The initial model of PSI introduced in [36] is described informally in terms of a multi-version concurrent algorithm as follows. A transaction T at a replica r proceeds by taking an initial *snapshot* S of the shared objects in r . The execution of T is then carried out locally: read operations query S and write operations similarly update S . Once the execution of T is completed, it attempts to *commit* its changes to r and it succeeds *only if* it is not *write-conflicted*. Transaction T is write-conflicted if another *committed* transaction T' has written to a location in r also written to by T , since it recorded its snapshot S . If T fails the conflict check it aborts and may restart the transaction; otherwise, it commits its changes to r , at which point its changes become visible to all other transactions that take a snapshot of replica r thereafter. These committed changes are later propagated to other replicas asynchronously.

The main difference between SI and PSI is in the way the committed changes at a replica r are propagated to other sites in the system. Under the SI model, committed transactions are *globally* ordered and the changes at each replica are propagated to others in this global order. This ensures that all concurrent transactions are observed in the same order by all replicas. By contrast, PSI does not enforce a global order on committed transactions: transactional effects are propagated between replicas in *causal* order. This ensures that, if replica r_1 commits a message m which is later read at replica r_2 , and r_2 posts a response m' , no replica can see m' without having seen the original message m . However,

causal propagation allows two replicas to observe concurrent events as if occurring in different orders: if r_1 and r_2 concurrently commit messages m and m' , then replica r_3 may initially see m but not m' , and r_4 may see m' but not m . This is best illustrated by the (IRIW+txs) example in Sect. 1.

2.3 Towards a Lock-Based Reference Implementation for PSI

While the description of PSI above is suitable for understanding PSI, it is not very useful for integrating the PSI model in languages such as C, C++ or Java. From a programmer's perspective, in such languages the various threads directly access the shared memory; they do not access their own replicas, which are loosely related to the replicas of other threads. What we would therefore like is an equivalent description of PSI in terms of unreplicated accesses to shared memory and a synchronisation mechanism such as locks.

In effect, we want a definition similar in spirit to *global lock atomicity* (GLA) [28], which is arguably the simplest TM model, and models committed transactions as acquiring a global mutual exclusion lock, then accessing and updating the data in place, and finally releasing the global lock. Naturally, however, the implementation of PSI cannot be that simple.

A first observation is that PSI cannot be simply implemented over sequentially consistent (SC) shared memory.¹ To see this, consider the IRIW+txs program from the introduction. Although PSI allows the annotated behaviour, SC forbids it for the corresponding program without transactions. The point is that under SC, either the $x := 1$ or the $y := 1$ write first reaches memory. Suppose, without loss of generality, that $x := 1$ is written to memory before $y := 1$. Then, the possible atomic snapshots of memory are $x = y = 0$, $x = 1 \wedge y = 0$, and $x = y = 1$. In particular, the snapshot read by T3 is impossible.

To implement PSI we therefore resort to a weaker memory model. Among weak memory models, the “multi-copy-atomic” ones, such as x86-TSO [29, 34], SPARC PSO [37, 38] and ARMv8-Flat [31], also forbid the weak outcome of (IRIW+txs) in the same way as SC, and so are unsuitable for our purpose. We thus consider *release-acquire consistency* (RA) [8, 9, 21], a simple and well-behaved non-multi-copy-atomic model. It is readily available as a subset of the C/C++11 memory model [9] with verified compilation schemes to all major architectures.

RA provides a crucial property that is relied upon in the earlier description of PSI, namely *causality*. In terms of RA, this means that if thread A observes a write w of thread B, then it also observes all the previous writes of thread B as well as any other writes B observed before performing w .

A second observation is that using a single lock to enforce mutual exclusion does not work as we need to allow transactions that access disjoint sets of locations to complete in parallel. An obvious solution is to use multiple locks—one

¹ *Sequential consistency* (SC) [24] is the standard model for shared memory concurrency and defines the behaviours of a multi-threaded program as those arising by executing sequentially some interleaving of the accesses of its constituent threads.

per location—as in the *disjoint lock atomicity* (DLA) model [28]. The question remaining is how to implement taking a snapshot at the beginning of a transaction.

A naive attempt is to use reader/writer locks, which allow multiple readers (taking the snapshots) to run in parallel, as long as no writer has acquired the lock. In more detail, the idea is to acquire reader locks for all locations read by a transaction, read the locations and store their values locally, and then release the reader locks. However, as we describe shortly, this approach does not work. Consider the (IRIW+txs) example in Sect. 1. For T2 to get the annotated outcome, it must release its reader lock for y before T4 acquires it. Likewise, since T3 observes $y = 1$, it must acquire its reader lock for y after T4 releases it. By this point, however, it is transitively after the release of the y lock by T2, and so, because of causality, it must have observed all the writes observed by T2 by that point—namely, the $x := 1$ write. In essence, the problem is that reader-writer locks over-synchronise. When two threads acquire the same reader lock, they synchronise, whereas two read-only transactions should never synchronise in PSI.

To resolve this problem, we use *sequence locks* [13,18,23,32]. Under the sequence locking protocol, each location x is associated with a sequence (version) number vx , initialised to zero. Each write to x increments vx before and after its update, provided that vx is even upon the first increment. Each read from x checks vx before and after reading x . If both values are the same and even, then there cannot have been any concurrent increments, and the reader must have seen a consistent value. That is, $\text{read}(x) \triangleq \text{do}\{v:=vx; s:=x\} \text{ while}(\text{is-odd}(v) \mid vx \neq v)$. Under SC, sequence locks are equivalent to reader-writer locks; however, under RA, they are weaker exactly because readers do not synchronise.

Handling Non-transactional Accesses. Let us consider what happens if some of the data accessed by a transaction is modified concurrently by an atomic non-transactional write. Since non-transactional accesses do not acquire any locks, the snapshots taken can include values written by non-transactional accesses. The result of the snapshot then depends on the order in which the variables are read. Consider for example the following litmus test:

$$x := 1; \parallel \text{ T: } \begin{cases} a := y; \text{ //reads } 1 \\ b := x; \text{ //reads } 0 \end{cases}$$

In our implementation, if the transaction’s snapshot reads y before x , then the annotated weak behaviour is not possible, because the underlying model (RA) disallows the weak “message passing” behaviour. If, however, x is read before y by the snapshot, then the weak behaviour is possible. In essence, this means that the PSI implementation described so far is of little use, when there are races between transactional and non-transactional code.

Another problem is the lack of *monotonicity*. A programmer might expect that wrapping some code in a transaction block will never yield additional

behaviours not possible in the program without transactions. Yet, in this example, removing the T block and unwrapping its code gets rid of the annotated weak behaviour!

To get monotonicity, it seems that snapshots must read the variables in the same order they are accessed by the transactions. How can this be achieved for transactions that say read x , then y , and then x again? Or transactions that depending on some complex condition, access first x and then y or vice versa? The key to solving this conundrum is surprisingly simple: *read each variable twice*. In more detail, one takes two snapshots of the locations read by the transaction, and checks that both snapshots return the same values for each location. This ensures that every location is read both before and after every other location in the transaction, and hence all the high-level happens-before orderings in executions of the transactional program are also respected by its implementation.

There is however one caveat: since equality of values is used to determine whether the two snapshots are the same, we will miss cases where different non-transactional writes to a variable write the same value. In our formal development (see Sect. 5), we thus assume that if multiple non-transactional writes write the same value to the same location, they cannot race with the same transaction. This assumption is necessary for the soundness of our implementation and cannot be lifted without instrumenting non-transactional accesses.

3 The Release-Acquire Memory Model for STM

We present the notational conventions used in the remainder of this article and proceed with the declarative model of the *release-acquire* (RA) fragment [21] of the C11 memory model [9], in which we implement our STM. In Sect. 3.1 we describe how we extend this formal model to specify the behaviour of STM programs.

Notation. Given a relation r on a set A , we write $r^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closure of r , respectively. We write r^{-1} for the inverse of r ; $r|_A$ for $r \cap A^2$; $[A]$ for the identity relation on A , i.e. $\{(a, a) \mid a \in A\}$; $\text{irreflexive}(r)$ for $\neg \exists a. (a, a) \in r$; and $\text{acyclic}(r)$ for $\text{irreflexive}(r^+)$. Given two relations r_1 and r_2 , we write $r_1; r_2$ for their (left) relational composition, i.e. $\{(a, b) \mid \exists c. (a, c) \in r_1 \wedge (c, b) \in r_2\}$. Lastly, when r is a strict partial order, we write $r|_{\text{imm}}$ for the *immediate* edges in r : $\{(a, b) \in r \mid \neg \exists c. (a, c) \in r \wedge (c, b) \in r\}$.

The RA model is given by the fragment of the C11 memory model, where all read accesses are acquire (**acq**) reads, all writes are release (**rel**) writes, and all atomic updates (i.e. RMWs) are acquire-release (**acqrel**) updates. The semantics of a program under RA is defined as a set of *consistent executions*.

Definition 1 (Executions in RA). Assume a finite set of *locations* Loc ; a finite set of *values* Val ; and a finite set of *thread identifiers* TID . Let x, y, z range over locations, v over values and τ over thread identifiers. An *RA execution graph of an STM implementation*, G , is a tuple of the form $(E, \text{po}, \text{rf}, \text{mo})$ with its nodes given by E and its edges given by the po , rf and mo relations such that:

- $E \subset \mathbb{N}$ is a finite set of *events*, and is accompanied with the functions $\text{tid}(\cdot) : E \rightarrow \text{TID}$ and $\text{lab}(\cdot) : E \rightarrow \text{LABEL}$, returning the thread identifier and the label of an event, respectively. We typically use a , b , and e to range over events. The label of an event is a tuple of one of the following three forms: (i) $R(x, v)$ for *read* events; (ii) $W(x, v)$ for *write* events; or (iii) $U(x, v, v')$ for *update* events. The $\text{lab}(\cdot)$ function induces the functions $\text{typ}(\cdot)$, $\text{loc}(\cdot)$, $\text{val}_r(\cdot)$ and $\text{val}_w(\cdot)$ that respectively project the type (R , W or U), location, and read/written values of an event, where applicable. The set of *read events* is denoted by $\mathcal{R} \triangleq \{e \in E \mid \text{typ}(e) \in \{R, U\}\}$; similarly, the set of *write events* is denoted by $\mathcal{W} \triangleq \{e \in E \mid \text{typ}(e) \in \{W, U\}\}$ and the set of *update events* is denoted by $\mathcal{U} \triangleq \mathcal{R} \cap \mathcal{W}$.

We further assume that E always contains a set E_0 of initialisation events consisting of a write event with label $W(x, 0)$ for every $x \in \text{Loc}$.

- $\text{po} \subseteq E \times E$ denotes the ‘*program-order*’ relation, defined as a disjoint union of strict total orders, each orders the events of one thread, together with $E_0 \times (E \setminus E_0)$ that places the initialisation events before any other event.
- $\text{rf} \subseteq \mathcal{W} \times \mathcal{R}$ denotes the ‘*reads-from*’ relation, defined as a relation between write and read events of the same location and value; it is total and functional on reads, i.e. every read event is related to exactly one write event;
- $\text{mo} \subseteq \mathcal{W} \times \mathcal{W}$ denotes the ‘*modification-order*’ relation, defined as a disjoint union of strict orders, each of which totally orders the write events to one location.

We often use “ G .” as a prefix to project the various components of G (e.g. $G.E$). Given a relation $r \subseteq E \times E$, we write r_{loc} for $r \cap \{(a, b) \mid \text{loc}(a) = \text{loc}(b)\}$. Analogously, given a set $A \subseteq E$, we write A_x for $A \cap \{a \mid \text{loc}(a) = x\}$. Lastly, given the rf and mo relations, we define the ‘*reads-before*’ relation $\text{rb} \triangleq \text{rf}^{-1}; \text{mo} \setminus [E]$.

Executions of a given program represent traces of shared memory accesses generated by the program. We only consider “partitioned” programs of the form $\parallel_{\tau \in \text{TID}} c_\tau$, where \parallel denotes parallel composition, and each c_i is a sequential program. The set of executions associated with a given program is then defined by induction over the structure of sequential programs. We do not define this construction formally as it depends on the syntax of the implementation programming language. Each execution of a program P has a particular program *outcome*, prescribing the final values of local variables in each thread (see example in Fig. 1).

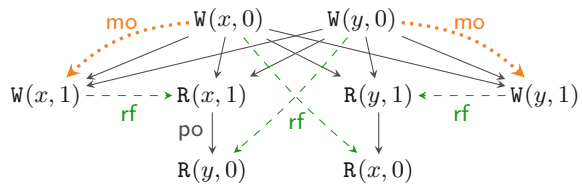


Fig. 1. An RA-consistent execution of a transaction-free variant of (IRIW+txs) in Sect. 1, with program outcome $a = c = 1$ and $b = d = 0$.

In this initial stage, the execution outcomes are unrestricted in that there are no constraints on the **rf** and **mo** relations. These restrictions and thus the permitted outcomes of a program are determined by the set of *consistent* executions:

Definition 2 (RA-consistency). A program execution G is *RA-consistent*, written $\text{RA-consistent}(G)$, if $\text{acyclic}(\text{hb}_{loc} \cup \text{mo} \cup \text{rb})$ holds, where $\text{hb} \triangleq (\text{po} \cup \text{rf})^+$ denotes the ‘RA-happens-before’ relation.

Among all executions of a given program P , only the *RA-consistent* ones define the allowed outcomes of P .

3.1 Software Transactional Memory in RA: Specification

Our goal in this section is to develop a declarative framework that allows us to specify the behaviour of mixed-mode STM programs under weak isolation guarantees. Whilst the behaviour of transactional code is dictated by the particular isolation model considered (e.g. PSI), the behaviour of non-transactional code and its interaction with transactions is guided by the underlying memory model. As we build our STM in the RA fragment of C11, we assume the behaviour of non-transactional code to conform to the RA memory model. More concretely, we build our specification of a program P such that (i) in the absence of transactional code, the behaviour of P is as defined by the RA model; (ii) in the absence of non-transactional code, the behaviour of P is as defined by the PSI model.

Definition 3 (Specification Executions). Assume a finite set of *transaction identifiers* TXID . An *execution graph* of an STM specification, Γ , is a tuple of the form $(E, \text{po}, \text{rf}, \text{mo}, T)$ where:

- $E \triangleq \mathcal{R} \cup \mathcal{W} \cup \mathcal{B} \cup \mathcal{E}$, denotes the set of *events* with \mathcal{R} and \mathcal{W} defined as the sets of read and write events as described above; and the \mathcal{B} and \mathcal{E} respectively denote the set of events marking the *beginning* and *end* of *transactions*. For each event $a \in \mathcal{B} \cup \mathcal{E}$, the $\text{lab}(\cdot)$ function is extended to return **B** when $a \in \mathcal{B}$, and **E** when $a \in \mathcal{E}$. The $\text{typ}(\cdot)$ function is accordingly extended to return a type in $\{\text{R}, \text{W}, \text{U}, \text{B}, \text{E}\}$, whilst the remaining functions are extended to return default (dummy) values for events in $\mathcal{B} \cup \mathcal{E}$.
- po , **rf** and **mo** denote the ‘program-order’, ‘reads-from’ and ‘modification-order’ relations as described above;
- $T \subseteq E$ denotes the set of *transactional events* with $\mathcal{B} \cup \mathcal{E} \subseteq T$. For transactional events in T , event labels are extended to carry an additional component, namely the associated transaction identifier. As such, a specification graph is additionally accompanied with the function $\text{tx}(\cdot) : T \rightarrow \text{TXID}$, returning the transaction identifier of transactional events. The derived ‘same-transaction’ relation, $\text{st} \in T \times T$, is the equivalence relation given by $\text{st} \triangleq \{(a, b) \in T \times T \mid \text{tx}(a) = \text{tx}(b)\}$.

We write T/st for the set of equivalence classes of T induced by st ; $[a]_{\text{st}}$ for the equivalence class that contains a ; and \mathcal{T}_{ξ} for the equivalence class of transaction

$\xi \in \text{TXID}$: $\mathcal{T}_\xi \triangleq \{a \mid \text{tx}(a) = \xi\}$. We write \mathcal{NT} for non-transactional events: $\mathcal{NT} \triangleq E \setminus \mathcal{T}$. We often use “ Γ .” as a prefix to project the Γ components.

Specification Consistency. The consistency of specification graphs is model-specific in that it is dictated by the guarantees provided by the underlying model. In the upcoming sections, we present two consistency definitions of PSI in terms of our specification graphs that lack cycles of certain shapes. In doing so, we often write r_Γ for lifting a relation $r \subseteq E \times E$ to transaction classes: $r_\Gamma \triangleq \text{st}; (r \setminus \text{st}); \text{st}$. Analogously, we write r_I to restrict r to the internal events of a transaction: $r \cap \text{st}$.

Comparison to Dependency Graphs. Adya et al. proposed *dependency graphs* for declarative specification of transactional consistency models [5, 7]. Dependency graphs are similar to our specification graphs in that they are constructed from a set of nodes and a set of edges (relations) capturing certain dependencies. However, unlike our specification graphs, the nodes in dependency graphs denote entire transactions and not individual events. In particular, Adya et al. propose three types of dependency edges: (i) a *read dependency* edge, $T_1 \xrightarrow{WR} T_2$, denotes that transaction T_2 reads a value written by T_1 ; (ii) a *write dependency* edge $T_1 \xrightarrow{WW} T_2$ denotes that T_2 overwrites a value written by T_1 ; and (iii) an *anti-dependency* edge $T_1 \xrightarrow{RW} T_2$ denotes that T_2 overwrites a value read by T_1 . Adya’s formalism does not allow for *non-transactional* accesses and it thus suffices to define the dependencies of an execution as edges between transactional classes. In our specification graphs however, we account for both transactional and non-transactional accesses and thus define our relational dependencies between individual events of an execution. However, when we need to relate an entire transaction to another with relation r , we use the transactional lift (r_Γ) defined above. In particular, Adya’s dependency edges correspond to ours as follows. Informally, the *WR* corresponds to our rf_Γ ; the *WW* corresponds to our mo_Γ ; and the *RW* corresponds to our rb_Γ . Adya’s dependency graphs have been used to develop declarative specifications of the PSI consistency model [14]. In Sect. 4, we revisit this model, redefine it as specification graphs in our setting, and develop a reference lock-based implementation that is sound and complete with respect to this abstract specification. The model in [14] does not account for non-transactional accesses. To remedy this, later in Sect. 5, we develop a declarative specification of PSI that allows for both transactional and non-transactional accesses. We then develop a reference lock-based implementation that is sound and complete with respect to our proposed model.

4 Parallel Snapshot Isolation (PSI)

We present a declarative specification of PSI (Sect. 4.1), and develop a lock-based reference implementation of PSI in the RA fragment (Sect. 4.2). We then demonstrate that our implementation is both sound (Sect. 4.3) and complete (Sect. 4.4) with respect to the PSI specification. Note that the PSI model in this section accounts for transactional code only; that is, throughout this section we assume that $\Gamma.E = \Gamma.\mathcal{T}$. We lift this assumption later in Sect. 5.

4.1 A Declarative Specification of PSI STMs in RA

In order to formally characterise the weak behaviour and anomalies admitted by PSI, Cerone and Gotsman [14, 15] formulated a declarative PSI specification. (In fact, they provide two equivalent specifications: one using dependency graphs proposed by Adya et al. [5, 7]; and the other using abstract executions.) As is standard, they characterise the set of executions admitted under PSI as graphs that lack certain cycles. We present an equivalent declarative formulation of PSI, adapted to use our notation as discussed in Sect. 3. It is straightforward to verify that our definition coincides with the dependency graph specification in [15]. As with [14, 15], throughout this section, we take PSI execution graphs to be those in which $E = \mathcal{T} \subseteq (\mathcal{R} \cup \mathcal{W}) \setminus \mathcal{U}$. That is, the PSI model handles transactional code only, consisting solely of read and write events (excluding updates).

PSI Consistency. A PSI execution graph $\Gamma = (E, \text{po}, \text{rf}, \text{mo}, \mathcal{T})$ is *consistent*, written $\text{psi-consistent}(\Gamma)$, if the following hold:

- $\text{rf}_I \cup \text{mo}_I \cup \text{rb}_I \subseteq \text{po}$ (INT)
- $\text{irreflexive}((\text{po}_T \cup \text{rf}_T \cup \text{mo}_T)^+; \text{rb}_T^?)$ (EXT)

Informally, INT ensures the consistency of each transaction internally, while EXT provides the synchronisation guarantees among transactions. In particular, we note that the two conditions together ensure that if two read events in the same transaction read from the same location x , and no write to x is *po*-between them, then they must read from the same write (known as ‘internal read consistency’).

Next, we provide an alternative formulation of PSI-consistency that is closer in form to RA-consistency. This formulation is the basis of our extension in Sect. 5 with non-transactional accesses.

Lemma 1. *A PSI execution graph $\Gamma = (E, \text{po}, \text{rf}, \text{mo}, \mathcal{T})$ is consistent if and only if $\text{acyclic}(\text{psi-hb}_{\text{loc}} \cup \text{mo} \cup \text{rb})$ holds, where psi-hb denotes the ‘PSI-happens-before’ relation, defined as $\text{psi-hb} \triangleq (\text{po} \cup \text{rf} \cup \text{rf}_T \cup \text{mo}_T)^+$.*

Proof. The full proof is provided in the technical appendix [4].

Note that this acyclicity condition is rather close to that of RA-consistency definition presented in Sect. 3, with the sole difference being the definition of ‘happens-before’ relation by replacing hb with psi-hb . The relation psi-hb is a strict extension of hb with $\text{rf}_T \cup \text{mo}_T$, which captures additional synchronisation guarantees resulting from transaction orderings, as described shortly. As in RA-consistency, the po and rf are included in the ‘PSI-happens-before’ relation psi-hb . Additionally, the rf_T and mo_T also contribute to psi-hb .

Intuitively, the rf_T corresponds to synchronisation due to causality between transactions. A transaction T_1 is causally-ordered before transaction T_2 , if T_1 writes to x and T_2 later (in ‘happens-before’ order) reads x . The inclusion of rf_T ensures that T_2 cannot read from T_1 without observing its entire effect. This in turn ensures that transactions exhibit an atomic ‘all-or-nothing’ behaviour. In particular, transactions cannot mix-and-match the values they read.

<pre> 0. for (x ∈ WS) lock vx; 1. for (x ∈ RS) { 2. a := vx; 3. if (is-odd(a) && x ∉ WS) continue; 4. if (x ∉ WS) v[x] := a; 5. s[x] := x; } 6. for (x ∈ RS) 7. if (¬valid(x)) goto line 1; 8. $\llbracket T \rrbracket$; 9. for (x ∈ WS) unlock vx; </pre>	<pre> lock vx \triangleq retry: v[x] := vx; if (is-odd(v[x])) goto retry; if (!CAS(vx, v[x], v[x]+1)) goto retry; unlock vx \triangleq vx := v[x] + 2 valid(x) \triangleq vx == v[x] valid_{RPSI}(x) \triangleq vx == v[x] && x == s[x] $\llbracket a := x \rrbracket \triangleq a := s[x]$ $\llbracket x := a \rrbracket \triangleq x := a; s[x] := a$ $\llbracket S_1; S_2 \rrbracket \triangleq \llbracket S_1 \rrbracket; \llbracket S_2 \rrbracket$ $\llbracket \text{while}(e) S \rrbracket \triangleq \text{while}(e) \llbracket S \rrbracket$... and so on ... </pre>
--	--

Fig. 2. PSI implementation of transaction T given RS , WS ; the RPSI implementation (Sect. 5) is obtained by replacing `valid` on line 7 with `validRPSI`.

For instance, if T_1 writes to both x and y , transaction T_2 may not read the value of x from T_1 but read the value of y from an earlier (in ‘happens-before’ order) transaction T_0 .

The mo_T corresponds to synchronisation due to conflicts between transactions. Its inclusion enforces the write-conflict-freedom of PSI transactions. In other words, if two transactions T_1 and T_2 both write to the same location x via events w_1 and w_2 such that $w_1 \xrightarrow{\text{mo}} w_2$, then T_1 must commit before T_2 , and thus the entire effect of T_1 must be visible to T_2 .

4.2 A Lock-Based PSI Implementation in RA

We present an operational model of PSI that is both sound and complete with respect to the declarative semantics in Sect. 4.1. To this end, in Fig. 2 we develop a pessimistic (lock-based) reference implementation of PSI using sequence locks [13, 18, 23, 32], referred to as *version locks* in our implementation. In order to avoid taking a snapshot of the *entire* memory and thus decrease the locking overhead, we assume that a transaction T is supplied with its *read set*, RS , containing those locations that are read by T . Similarly, we assume T to be supplied with its *write set*, WS , containing the locations updated by T .²

The implementation of T proceeds by exclusively acquiring the version locks on all locations in its write set (line 0). It then obtains a snapshot of the locations in its read set by inspecting their version locks, as described shortly, and subsequently recording their values in a thread-local array s (lines 1–7). Once a snapshot is recorded, the execution of T proceeds locally (via $\llbracket T \rrbracket$ on line 8) as

² A conservative estimate of RS and WS can be obtained by simple syntactic analysis.

follows. Each read operation consults the local snapshot in \mathbf{s} ; each write operation updates the memory eagerly (in-place) and subsequently updates its local snapshot to ensure correct lookup for future reads. Once the execution of T is concluded, the version locks on the write set are released (line 9). Observe that as the writer locks are acquired pessimistically, we do not need to check for write-conflicts in the implementation.

To facilitate our locking implementation, we assume that each location \mathbf{x} is associated with a version lock at address $\mathbf{x}+1$, written \mathbf{vx} . The value held by a version lock \mathbf{vx} may be in one of two categories: (i) an even number, denoting that the lock is free; or (ii) an odd number, denoting that the lock is exclusively held by a writer. For a transaction to write to a location \mathbf{x} in its write set \mathbf{WS} , the \mathbf{x} version lock (\mathbf{vx}) must be acquired exclusively by calling `lock vx`. Each call to `lock vx` reads the value of \mathbf{vx} and stores it in $\mathbf{v}[\mathbf{x}]$, where \mathbf{v} is a thread-local array. It then checks if the value read is even (\mathbf{vx} is free) and if so it atomically increments it by 1 (with a ‘compare-and-swap’ operation), thus changing the value of \mathbf{vx} to an odd number and acquiring it exclusively; otherwise it repeats this process until the version lock is successfully acquired. Conversely, each call to `unlock vx` updates the value of \mathbf{vx} to $\mathbf{v}[\mathbf{x}]+2$, restoring the value of \mathbf{vx} to an even number and thus releasing it. Note that deadlocks can be avoided by imposing an ordering on locks and ensuring their in-order acquisition by all transactions. For simplicity however, we have elided this step as we are not concerned with progress or performance issues here and our main objective is a reference implementation of PSI in RA.

Analogously, for a transaction to read from the locations in its read set \mathbf{RS} , it must record a snapshot of their values (lines 1–7). To obtain a snapshot of location \mathbf{x} , the transaction must ensure that \mathbf{x} is not currently being written to by another transaction. It thus proceeds by reading the value of \mathbf{vx} and recording it in $\mathbf{v}[\mathbf{x}]$. If \mathbf{vx} is free (the value read is even) or \mathbf{x} is in its write set \mathbf{WS} , the value of \mathbf{x} can be freely read and tentatively stored in $\mathbf{s}[\mathbf{x}]$. In the latter case, the transaction has already acquired the exclusive lock on \mathbf{vx} and is thus safe in the knowledge that no other transaction is currently updating \mathbf{x} . Once a *tentative* snapshot of all locations is obtained (lines 1–5), the transaction must *validate* it by ensuring that it reflects the values of the read set at a single point in time (lines 6–7). To do this, it revisits the version locks, inspecting whether their values have changed (by checking them against \mathbf{v}) since it recorded its snapshot. If so, then an intermediate update has intervened, potentially invalidating the obtained snapshot; the transaction thus restarts the snapshot process. Otherwise, the snapshot is successfully validated and returned in \mathbf{s} .

4.3 Implementation Soundness

The PSI implementation in Fig. 2 is *sound*: for each RA-consistent implementation graph G , a corresponding specification graph Γ can be constructed such that $\text{psi-consistent}(\Gamma)$ holds. In what follows we state our soundness theorem and briefly describe our construction of consistent specification graphs. We refer the reader to the technical appendix [4] for the full soundness proof.

Theorem 1 (Soundness). *For all RA-consistent implementation graphs G of the implementation in Fig. 2, there exists a PSI-consistent specification graph Γ of the corresponding transactional program that has the same program outcome.*

Constructing Consistent Specification Graphs. Observe that given an execution of our implementation with t transactions, the trace of each transaction $i \in \{1 \dots t\}$ is of the form $\theta_i = Ls_i \xrightarrow{po} FS_i \xrightarrow{po} S_i \xrightarrow{po} Ts_i \xrightarrow{po} Us_i$, where Ls_i , FS_i , S_i , Ts_i and Us_i respectively denote the sequence of events acquiring the version locks, attempting but failing to obtain a valid snapshot, recording a valid snapshot, performing the transactional operations, and releasing the version locks. For each transactional trace θ_i of our implementation, we thus construct a corresponding trace of the specification as $\theta'_i = B_i \xrightarrow{po} Ts'_i \xrightarrow{po} E_i$, where B_i and E_i denote the transaction begin and end events ($\text{lab}(B_i)=B$ and $\text{lab}(E_i)=E$). When Ts_i is of the form $t_1 \xrightarrow{po} \dots \xrightarrow{po} t_n$, we construct Ts'_i as $t'_1 \xrightarrow{po} \dots \xrightarrow{po} t'_n$ with each t'_j defined either as $t'_j \triangleq R(x, v)$ when $t_j = R(s[x], v)$ (i.e. the corresponding implementation event is a read event); or as $t'_j \triangleq W(x, v)$ when $t_j = W(x, v) \xrightarrow{po} W(s[x], v)$.

For each specification trace θ'_i we construct the ‘reads-from’ relation as:

$$RF_i \triangleq \left\{ (w, t'_j) \left| \begin{array}{l} t'_j \in Ts'_i \wedge \exists x, v. t'_j = R(x, v) \wedge w = W(x, v) \\ \wedge (w \in Ts'_i \Rightarrow w \xrightarrow{po} t'_j \wedge \\ \quad (\forall e \in Ts'_i. w \xrightarrow{po} e \xrightarrow{po} t'_j \Rightarrow (\text{loc}(e) \neq x \vee e \notin \mathcal{W}))) \\ \wedge (w \notin Ts'_i \Rightarrow (\forall e \in Ts'_i. (e \xrightarrow{po} t'_j \Rightarrow (\text{loc}(e) \neq x \vee e \notin \mathcal{W})) \\ \quad \wedge \exists r' \in S_i. \text{loc}(r') = x \wedge (w, r') \in G.\text{rf})) \end{array} \right. \right\}$$

That is, we construct our graph such that each read event t'_j from location x in Ts'_i either (i) is preceded by a write event w to x in Ts'_i without an intermediate write in between them and thus ‘reads-from’ w (lines two and three); or (ii) is not preceded by a write event in Ts'_i and thus ‘reads-from’ the write event w from which the initial snapshot read r' in S_i obtained the value of x (last two lines).

Given a consistent implementation graph $G = (E, po, \text{rf}, \text{mo})$, we construct a consistent specification graph $\Gamma = (E, po, \text{rf}, \text{mo}, T)$ such that:

- $\Gamma.E \triangleq \bigcup_{i \in \{1 \dots t\}} \theta'_i$ – the events of $\Gamma.E$ is the union of events in each transaction trace θ'_i of the specification constructed as above;
- $\Gamma.po \triangleq G.po|_{\Gamma.E}$ – the $\Gamma.po$ is that of $G.po$ limited to the events in $\Gamma.E$;
- $\Gamma.\text{rf} \triangleq \bigcup_{i \in \{1 \dots t\}} RF_i$ – the $\Gamma.\text{rf}$ is the union of RF_i relations defined above;
- $\Gamma.\text{mo} \triangleq G.\text{mo}|_{\Gamma.E}$ – the $\Gamma.\text{mo}$ is that of $G.\text{mo}$ limited to the events in $\Gamma.E$;
- $\Gamma.T \triangleq \Gamma.E$, where for each $e \in \Gamma.T$, we define $\text{tx}(e) = i$ when $e \in \theta'_i$.

4.4 Implementation Completeness

The PSI implementation in Fig. 2 is *complete*: for each consistent specification graph Γ a corresponding implementation graph G can be constructed such that $\text{RA-consistent}(G)$ holds. We next state our completeness theorem and describe

our construction of consistent implementation graphs. We refer the reader to the technical appendix [4] for the full completeness proof.

Theorem 2 (Completeness). *For all PSI-consistent specification graphs Γ of a transactional program, there exists an RA-consistent execution graph G of the implementation in Fig. 2 that has the same program outcome.*

Constructing Consistent Implementation Graphs. In order to construct an execution graph of the implementation G from the specification Γ , we follow similar steps as those in the soundness construction, in reverse order. More concretely, given each trace θ'_i of the specification, we construct an analogous trace of the implementation by inserting the appropriate events for acquiring and inspecting the version locks, as well as obtaining a snapshot. For each transaction class $\mathcal{T}_i \in \mathcal{T}/\text{st}$, we must first determine its read and write sets and subsequently decide the order in which the version locks are acquired (for locations in the write set) and inspected (for locations in the read set). This then enables us to construct the ‘reads-from’ and ‘modification-order’ relations for the events associated with version locks.

Given a consistent execution graph of the specification $\Gamma = (E, \text{po}, \text{rf}, \text{mo}, \mathcal{T})$, and a transaction class $\mathcal{T}_i \in \Gamma.\mathcal{T}/\text{st}$, we write $\text{WS}_{\mathcal{T}_i}$ for the set of locations written to by \mathcal{T}_i . That is, $\text{WS}_{\mathcal{T}_i} \triangleq \bigcup_{e \in \mathcal{T}_i \cap W} \text{loc}(e)$. Similarly, we write $\text{RS}_{\mathcal{T}_i}$ for the set of locations read from by \mathcal{T}_i , *prior to* being written to by \mathcal{T}_i . For each location x read from by \mathcal{T}_i , we additionally record the first read event in \mathcal{T}_i that retrieved the value of x . That is,

$$\text{RS}_{\mathcal{T}_i} \triangleq \left\{ (x, r) \mid r \in \mathcal{T}_i \cap \mathcal{R}_x \wedge \neg \exists e \in \mathcal{T}_i \cap E_x. e \xrightarrow{\text{po}} r \right\}$$

Note that transaction \mathcal{T}_i may contain several read events reading from x , prior to subsequently updating it. However, the internal-read-consistency property ensures that all such read events read from the same write event. As such, as part of the read set of \mathcal{T}_i we record the first such read event (in program-order).

Determining the ordering of lock events hinges on the following observation. Given a consistent execution graph of the specification $\Gamma = (E, \text{po}, \text{rf}, \text{mo}, \mathcal{T})$, let for each location x the total order mo be given as: $w_1 \xrightarrow{\text{mo}|\text{imm}} \dots \xrightarrow{\text{mo}|\text{imm}} w_{n_x}$. Observe that this order can be broken into adjacent segments where the events of each segment belong to the *same* transaction. That is, given the transaction classes $\Gamma.\mathcal{T}/\text{st}$, the order above is of the following form where $\mathcal{T}_1, \dots, \mathcal{T}_m \in \Gamma.\mathcal{T}/\text{st}$ and for each such \mathcal{T}_i we have $x \in \text{WS}_{\mathcal{T}_i}$ and $w_{(i,1)} \dots w_{(i,n_i)} \in \mathcal{T}_i$:

$$\underbrace{w_{(1,1)} \xrightarrow{\text{mo}|\text{imm}} \dots \xrightarrow{\text{mo}|\text{imm}} w_{(1,n_1)}}_{\mathcal{T}_1} \xrightarrow{\text{mo}|\text{imm}} \dots \xrightarrow{\text{mo}|\text{imm}} \underbrace{w_{(m,1)} \xrightarrow{\text{mo}|\text{imm}} \dots \xrightarrow{\text{mo}|\text{imm}} w_{(m,n_m)}}_{\mathcal{T}_m}$$

Were this not the case and we had $w_1 \xrightarrow{\text{mo}} w \xrightarrow{\text{mo}} w_2$ such that $w_1, w_2 \in \mathcal{T}_i$ and $w \in \mathcal{T}_j \neq \mathcal{T}_i$, we would consequently have $w_1 \xrightarrow{\text{mo}_\top} w \xrightarrow{\text{mo}_\top} w_1$, contradicting the assumption that Γ is consistent. Given the above order, let us then define

$\Gamma.\text{MO}_x = [\mathcal{T}_1 \cdots \mathcal{T}_m]$. We write $\Gamma.\text{MO}_x|_i$ for the i^{th} item of $\Gamma.\text{MO}_x$. As we describe shortly, we use $\Gamma.\text{MO}_x$ to determine the order of lock events.

Note that the execution trace for each transaction $\mathcal{T}_i \in \Gamma.\mathcal{T}/\text{st}$ is of the form $\theta'_i = B_i \xrightarrow{\text{po}} Ts'_i \xrightarrow{\text{po}} E_i$, where B_i is a transaction-begin (B) event, E_i is a transaction-end (E) event, and $Ts'_i = t'_1 \xrightarrow{\text{po}} \cdots \xrightarrow{\text{po}} t'_n$ for some n , where each t'_j is either a read or a write event. As such, we have $\Gamma.E = \Gamma.\mathcal{T} = \bigcup_{\mathcal{T}_i \in \Gamma.\mathcal{T}/\text{st}} \mathcal{T}_i = \theta'_i.E$.

For each trace θ'_i of the specification, we construct a corresponding trace of our implementation θ_i as follows. Let $\text{RS}_{\mathcal{T}_i} = \{(\mathbf{x}_1, r_1) \cdots (\mathbf{x}_p, r_p)\}$ and $\text{WS}_{\mathcal{T}_i} = \{\mathbf{y}_1 \cdots \mathbf{y}_q\}$. We then construct $\theta_i = Ls_i \xrightarrow{\text{po}} S_i \xrightarrow{\text{po}} Ts_i \xrightarrow{\text{po}} Us_i$, where

- $Ls_i = L_i^{y_1} \xrightarrow{\text{po}} \cdots \xrightarrow{\text{po}} L_i^{y_q}$ and $Us_i = U_i^{y_1} \xrightarrow{\text{po}} \cdots \xrightarrow{\text{po}} U_i^{y_q}$ denote the sequence of events acquiring and releasing the version locks, respectively. Each $L_i^{y_j}$ and $U_i^{y_j}$ are defined as follows, the first event $L_i^{y_1}$ has the same identifier as that of B_i , the last event $U_i^{y_q}$ has the same identifier as that of E_i , and the identifiers of the remaining events are picked fresh:

$$L_i^{y_j} = \text{U}(\text{vy}_j, 2a, 2a+1) \quad U_i^{y_j} = \text{W}(\text{vy}_j, 2a+2) \quad \text{where } \text{MO}_{y_j}|_a = \mathcal{T}_i$$

We then define the **mo** relation for version locks such that if transaction \mathcal{T}_i writes to y immediately after \mathcal{T}_j (i.e. \mathcal{T}_i is MO_y -ordered immediately after \mathcal{T}_j), then \mathcal{T}_i acquires the vy version lock immediately after \mathcal{T}_j has released it. On the other hand, if \mathcal{T}_i is the first transaction to write to y , then it acquires vy immediately after the event initialising the value of vy , written init_{vy} . Moreover, each vy release event of \mathcal{T}_i is **mo**-ordered immediately after the corresponding vy acquisition event in \mathcal{T}_i :

$$\text{IMO}_i \triangleq \bigcup_{y \in \text{WS}_{\mathcal{T}_i}} \left\{ \begin{array}{l} (L_i^y, U_i^y), \\ (w, L_i^y) \end{array} \middle| \begin{array}{l} (\Gamma.\text{MO}_x|_0 = \mathcal{T}_i \Rightarrow w = \text{init}_{\text{vy}}) \wedge \\ (\exists \mathcal{T}_j, a > 0. \Gamma.\text{MO}_y|_a = \mathcal{T}_i \wedge \Gamma.\text{MO}_y|_{a-1} = \mathcal{T}_j) \\ \Rightarrow w = U_j^y \end{array} \right\}$$

This partial **mo** order on lock events of \mathcal{T}_i also determines the **rf** relation for its lock acquisition events: $\text{IRF}_i^1 \triangleq \bigcup_{y \in \text{WS}_{\mathcal{T}_i}} \{(w, L_i^y) \mid (w, L_i^y) \in \text{IMO}_i\}$.

- $S_i = tr_i^{x_1} \xrightarrow{\text{po}} \cdots \xrightarrow{\text{po}} tr_i^{x_p} \xrightarrow{\text{po}} vr_i^{x_1} \xrightarrow{\text{po}} \cdots \xrightarrow{\text{po}} vr_i^{x_p}$ denotes the sequence of events obtaining a tentative snapshot ($tr_i^{x_j}$) and subsequently validating it ($vr_i^{x_j}$). Each $tr_i^{x_j}$ sequence is defined as $ir_i^{x_j} \xrightarrow{\text{po}} r_i^{x_j} \xrightarrow{\text{po}} s_i^{x_j}$ (reading the version lock vx_j , reading \mathbf{x}_j and recoding it in \mathbf{s}), with $ir_i^{x_j}$, $r_i^{x_j}$, $s_i^{x_j}$ and $vr_i^{x_j}$ events defined as follows (with fresh identifiers). We then define the **rf** relation for each of these read events in S_i . For each $(\mathbf{x}, r) \in \text{RS}_{\mathcal{T}_i}$, when r (i.e. the read event in the specification class \mathcal{T}_i that reads the value of \mathbf{x}) reads from an event w in the specification graph ($(w, r) \in \Gamma.\text{rf}$), we add (w, r_i^x) to the **rf** relation of G (the first line of IRF_i^2 below). For version locks, if transaction \mathcal{T}_i also writes to \mathbf{x}_j , then $ir_i^{x_j}$ and $vr_i^{x_j}$ events (reading and validating the value of version lock vx_j), read from the lock event in \mathcal{T}_i that acquired vx_j , namely $L_i^{x_j}$. On the other hand, if transaction \mathcal{T}_i does

not write to \mathbf{x}_j and it reads the value of \mathbf{x}_j written by \mathcal{T}_j , then $ir_i^{\mathbf{x}_j}$ and $vr_i^{\mathbf{x}_j}$ read the value written to \mathbf{vx}_j by \mathcal{T}_j when releasing it ($U_j^{\mathbf{x}}$). Lastly, if \mathcal{T}_i does not write to \mathbf{x}_j and it reads the value of \mathbf{x}_j written by the initial write, $init_{\mathbf{x}}$, then $ir_i^{\mathbf{x}_j}$ and $vr_i^{\mathbf{x}_j}$ read the value written to \mathbf{vx}_j by the initial write to \mathbf{vx} , $init_{\mathbf{vx}}$.

$$\text{IRF}_i^2 \triangleq \bigcup_{(\mathbf{x}, r) \in \text{RS}_{\mathcal{T}_i}} \left\{ \begin{array}{l} (w, r_i^{\mathbf{x}}), \\ (w', ir_i^{\mathbf{x}}), \\ (w', vr_i^{\mathbf{x}}) \end{array} \middle| \begin{array}{l} (w, r) \in \Gamma.\text{rf} \\ \wedge (\mathbf{x} \in \text{WS}_{\mathcal{T}_i} \Rightarrow w' = L_i^{\mathbf{x}}) \\ \wedge (\mathbf{x} \notin \text{WS}_{\mathcal{T}_i} \wedge \exists \mathcal{T}_j. w \in \mathcal{T}_j \Rightarrow w' = U_j^{\mathbf{x}}) \\ \wedge (\mathbf{x} \notin \text{WS}_{\mathcal{T}_i} \wedge w = init_{\mathbf{x}} \Rightarrow w' = init_{\mathbf{vx}}) \end{array} \right\}$$

$$r_i^{\mathbf{x}_j} = R(\mathbf{x}_j, v) \quad s_i^{\mathbf{x}_j} = W(s[\mathbf{x}_j], v) \quad \text{s.t. } \exists w. (w, r_i^{\mathbf{x}_j}) \in \text{IRF}_i^2 \wedge \text{val}_w(w) = v$$

$$ir_i^{\mathbf{x}_j} = vr_i^{\mathbf{x}_j} = R(\mathbf{vx}_j, v) \quad \text{s.t. } \exists w. (w, ir_i^{\mathbf{x}_j}) \in \text{IRF}_i^2 \wedge \text{val}_w(w) = v$$

- $Ts_i = t_1 \xrightarrow{\text{po}} \dots \xrightarrow{\text{po}} t_n$ (when $Ts'_i = t'_1 \xrightarrow{\text{po}} \dots \xrightarrow{\text{po}} t'_n$), with t_j defined as follows:

$$t_j = R(s[\mathbf{x}], v) \text{ when } t'_j = R(\mathbf{x}, v)$$

$$t_j = W(\mathbf{x}, v) \xrightarrow{\text{po}|\text{imm}} W(s[\mathbf{x}], v) \text{ when } t'_j = W(\mathbf{x}, v)$$

When t'_j is a read event, the t_j has the same identifier as that of t'_j . When t'_j is a write event, the first event in t_j has the same identifier as that of t'_j and the identifier of the second event is picked fresh.

We are now in a position to construct our implementation graph. Given a consistent execution graph Γ of the specification, we construct an execution graph $G = (E, \text{po}, \text{rf}, \text{mo})$ of the implementation as follows.

- $G.E = \bigcup_{\mathcal{T}_i \in \Gamma.\mathcal{T}/\text{st}} \theta_i.E$ – note that $G.E$ is an extension of $\Gamma.E$: $\Gamma.E \subseteq G.E$.
- $G.\text{po}$ is defined as $\Gamma.\text{po}$ extended by the po for the additional events of G , given by the θ_i traces defined above.
- $G.\text{rf} = \bigcup_{\mathcal{T}_i \in \Gamma.\mathcal{T}/\text{st}} (\text{IRF}_i^1 \cup \text{IRF}_i^2)$
- $G.\text{mo} = \Gamma.\text{mo} \cup \left(\bigcup_{\mathcal{T}_i \in \Gamma.\mathcal{T}/\text{st}} \text{IMO}_i \right)^+$

5 Robust Parallel Snapshot Isolation (RPSI)

In the previous section we adapted the PSI semantics in [14] to STM settings, in the *absence* of non-transactional code. However, a reasonable STM should account for mixed-mode code where shared data is accessed by both transactional and non-transactional code. To remedy this, we explore the semantics of PSI STMs in the presence of non-transactional code with *weak isolation* guarantees (see Sect. 2.1). We refer to the weakly isolated behaviour of such PSI STMs as *robust parallel snapshot isolation* (RPSI), due to its ability to provide PSI guarantees between transactions even in the presence of non-transactional code.

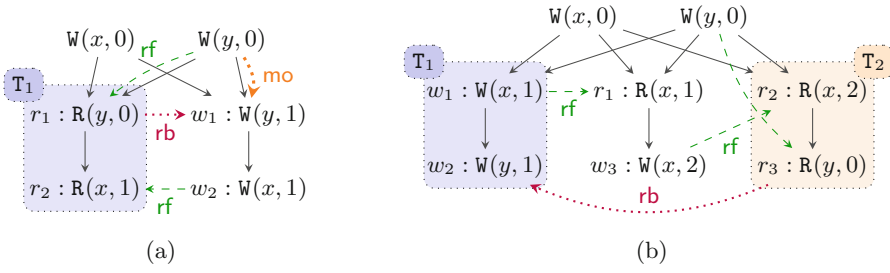


Fig. 3. RPSI-inconsistent executions due to **NT-RF** (a); and **T-RF** (b)

In Sect. 5.1 we propose the first declarative specification of RPSI STM programs. Later in Sect. 5.2 we develop a lock-based reference implementation of our RPSI specification in the RA fragment. We then demonstrate that our implementation is both sound (Sect. 5.3) and complete (Sect. 5.4) with respect to our proposed specification.

5.1 A Declarative Specification of RPSI STMs in RA

We formulate a declarative specification of RPSI semantics by adapting the PSI semantics presented in Sect. 4.1 to account for non-transactional accesses. As with the PSI specification in Sect. 4.1, throughout this section, we take RPSI execution graphs to be those in which $\mathcal{T} \subseteq (\mathcal{R} \cup \mathcal{W}) \setminus \mathcal{U}$. That is, RPSI transactions consist solely of read and write events (excluding updates). As before, we characterise the set of executions admitted by RPSI as graphs that lack cycles of certain shapes. More concretely, as with the PSI specification, we consider an RPSI execution graph to be *consistent* if $\text{acyclic}(\text{rpsi-hb}_{\text{loc}} \cup \text{mo} \cup \text{rb})$ holds, where **rpsi-hb** denotes the ‘*RPSI-happens-before*’ relation, extended from that of PSI **psi-hb**.

Definition 4 (RPSI consistency). An RPSI execution graph $\Gamma = (E, \text{po}, \text{rf}, \text{mo}, \mathcal{T})$ is consistent, written $\text{rpsi-consistent}(\Gamma)$, if $\text{acyclic}(\text{rpsi-hb}_{\text{loc}} \cup \text{mo} \cup \text{rb})$ holds, where **rpsi-hb** denotes the ‘*RPSI-happens-before*’ relation, defined as the smallest relation that satisfies the following conditions:

$$\begin{aligned}
 \text{rpsi-hb}; \text{rpsi-hb} &\subseteq \text{rpsi-hb} && (\text{TRANS}) \\
 \text{po} \cup \text{rf} \cup \text{mo}_{\mathcal{T}} &\subseteq \text{rpsi-hb} && (\text{PSI-HB}) \\
 [E \setminus \mathcal{T}]; \text{rf}; \text{st} &\subseteq \text{rpsi-hb} && (\text{NT-RF}) \\
 \text{st}; ([\mathcal{W}]; \text{st}; (\text{rpsi-hb} \setminus \text{st}); \text{st}; [\mathcal{R}])_{\text{loc}}; \text{st} &\subseteq \text{rpsi-hb} && (\text{T-RF})
 \end{aligned}$$

The **TRANS** and **PSI-HB** ensure that **rpsi-hb** is transitive and that it includes po , **rf** and $\text{mo}_{\mathcal{T}}$ as with its PSI counterpart. The **NT-RF** ensures that if a value written by a non-transactional write w is observed (read from) by a read event r in a transaction T , then its effect is observed by *all* events in T . That is, the w *happens-before* all events in T and not just r . This allows us to rule out executions such as the one depicted in Fig. 3a, which we argue must be disallowed by RPSI.

Consider the execution graph of Fig. 3a, where transaction T_1 is denoted by the dashed box labelled T_1 , comprising the read events r_1 and r_2 . Note that as r_1 and r_2 are transactional reads without prior writes by the transaction, they constitute a *snapshot* of the memory at the time T_1 started. That is, the values read by r_1 and r_2 must reflect a valid snapshot of the memory at the time it was taken. As such, since we have $(w_2, r_2) \in \text{rf}$, any event preceding w_2 by the ‘happens-before’ relation must also be observed by (synchronise with) T_1 . In particular, as w_1 happens-before w_2 ($(w_1, w_2) \in \text{po}$), the w_1 write must also be observed by T_1 . The **NT-RF** thus ensures that a non-transactional write read from by a transaction (i.e. a snapshot read) synchronises with the entire transaction.

Recall from Sect. 4.1 that the PSI **psi-hb** relation includes rf_T which has not yet been included in **rpsi-hb** through the first three conditions described. As we describe shortly, the **T-RF** is indeed a strengthening of rf_T to account for the presence of non-transactional events. In particular, note that rf_T is included in the left-hand side of **T-RF**: when **rpsi-hb** in $([W]; \text{st}; (\text{rpsi-hb} \setminus \text{st}); \text{st}; [\mathcal{R}])$ is replaced with $\text{rf} \subseteq \text{rpsi-hb}$, the left-hand side yields rf_T . As such, in the absence of non-transactional events, the definitions of **psi-hb** and **rpsi-hb** coincide.

Recall that inclusion of rf_T in **psi-hb** ensured transactional synchronisation due to causal ordering: if T_1 writes to x and T_2 later (in **psi-hb** order) reads x , then T_1 must synchronise with T_2 . This was achieved in PSI because either (i) T_2 reads x directly from T_1 in which case T_1 synchronises with T_2 via rf_T ; or (ii) T_2 reads x from another later (**mo**-ordered) transactional write in T_3 , in which case T_1 synchronises with T_3 via mo_T , T_3 synchronises with T_2 via rf_T , and thus T_1 synchronises with T_2 via $\text{mo}_T; \text{rf}_T$. How are we then to extend **rpsi-hb** to guarantee transactional synchronisation due to causal ordering in the presence of non-transactional events?

To justify **T-RF**, we present an execution graph that does not guarantee synchronisation between causally ordered transactions and is nonetheless deemed RPSI-consistent *without* the **T-RF** condition on **rpsi-hb**. We thus argue that this execution must be precluded by RPSI, justifying the need for **T-RF**. Consider the execution in Fig. 3b. Observe that as transaction T_1 writes to x via w_1 , transaction T_2 reads x via r_2 , and $(w_1, r_2) \in \text{rpsi-hb}$ ($w_1 \xrightarrow{\text{rf}} r_1 \xrightarrow{\text{po}} w_3 \xrightarrow{\text{rf}} r_2$), T_1 is causally ordered before T_2 and hence T_1 must synchronise with T_2 . As such, the r_3 in T_2 must observe w_2 in T_1 : we must have $(w_2, r_3) \in \text{rpsi-hb}$, rendering the above execution RPSI-inconsistent. To enforce the **rpsi-hb** relation between such causally ordered transactions with intermediate non-transactional events, **T-RF** stipulates that if a transaction T_1 writes to a location (e.g. x via w_1 above), another transaction T_2 reads from the same location (r_2), and the two events are related by ‘RPSI-happens-before’ ($(w_1, r_2) \in \text{rpsi-hb}$), then T_1 must synchronise with T_2 . That is, all events in T_1 must ‘RPSI-happen-before’ those in T_2 . Effectively, this allows us to transitively close the causal ordering between transactions, spanning transactional and non-transactional events in between.

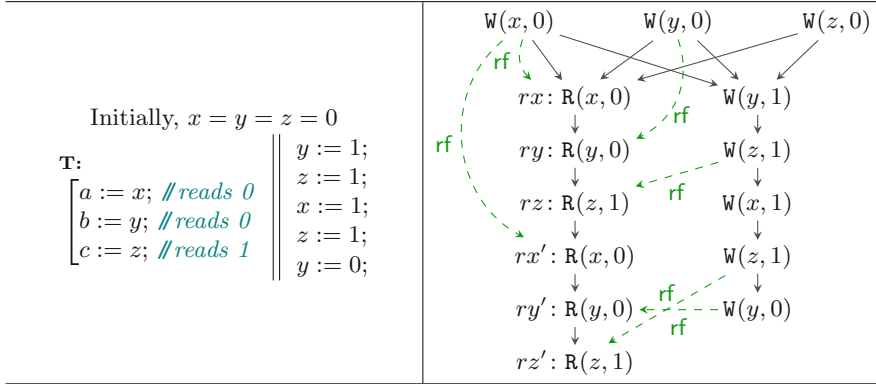


Fig. 4. A mixed-mode program with its annotated behaviour disallowed by RPSI (left); an RA-consistent execution graph of its RPSI implementation (right)

5.2 A Lock-Based RPSI Implementation in RA

We present a lock-based reference implementation of RPSI in the RA fragment (Fig. 2) by using sequence locks [13, 18, 23, 32]. Our implementation is both sound and complete with respect to our declarative RPSI specification in Sect. 5.1.

The RPSI implementation in Fig. 2 is rather similar to its PSI counterpart. The main difference between the two is in how they *validate* the tentative snapshot recorded in s . As before, in order to ensure that no intermediate *transactional* writes have intervened since s was recorded, for each location x in RS , the validation phase revisits vx , inspecting whether its value has changed from that recorded in $v[x]$. If this is the case, the snapshot is deemed invalid and the process is restarted. However, checking against intermediate transactional writes alone is not sufficient as it does not preclude the intervention of *non-transactional* writes. This is because unlike transactional writes, non-transactional writes do not update the version locks and as such their updates may go unnoticed. In order to rule out the possibility of intermediate non-transactional writes, for each location x the implementation checks the value of x against that recorded in $s[x]$. If the values do not agree, an intermediate non-transactional write has been detected: the snapshot fails validation and the process is restarted. Otherwise, the snapshot is successfully validated and returned in s . Observe that checking the value of x against $s[x]$ does not entirely preclude the presence of non-transactional writes, in cases where the same value is written (non-transactionally) to x twice.

To understand this, consider the mixed-mode program on the left of Fig. 4 comprising a transaction in the left-hand thread and a non-transactional program in the right-hand thread writing the same value (1) to z twice. Note that the annotated behaviour is disallowed under RPSI: all execution graphs of the program with the annotated behaviour yield RPSI-inconsistent execution graphs. Intuitively, this is because the values read by the transaction ($x : 0, y : 0, z : 1$)

do not constitute a valid *snapshot*: at *no* point during the execution of this program, are the values of x , y and z as annotated.

Nevertheless, it is possible to find an RA-consistent execution of the RPSI implementation in Fig. 2 that reads the annotated values as its snapshot. Consider the execution graph on the right-hand side of Fig. 4, depicting a particular execution of the RPSI implementation (Fig. 2) of the program on the left. The rx , ry and rz denote the events reading the initial snapshot of x , y and z and recording them in \mathbf{s} (line 5), respectively. Similarly, the rx' , ry' and rz' denote the events validating the snapshots recorded in \mathbf{s} (line 7). As T is the only transaction in the program, the version numbers \mathbf{vx} , \mathbf{vy} and \mathbf{vz} remain unchanged throughout the execution and we have thus omitted the events reading (line 2) and validating (line 7) their values from the execution graph. Note that this execution graph is RA-consistent even though we cannot find a corresponding RPSI-consistent execution with the same outcome. To ensure the soundness of our implementation, we must thus rule out such scenarios.

To do this, we assume that if multiple non-transactional writes write the same value to the same location, they cannot race with the same transaction. More concretely, we assume that *every* RPSI-consistent execution graph of a given program satisfies the following condition:

$$\begin{aligned} \forall \mathbf{x}. \forall r \in \mathcal{T} \cap \mathcal{R}_{\mathbf{x}}. \forall w, w' \in \mathcal{NT} \cap \mathcal{W}_{\mathbf{x}}. \\ w \neq w' \wedge \mathbf{val}_{\mathbf{w}}(w) = \mathbf{val}_{\mathbf{w}}(w') \wedge (r, w) \notin \text{rpsi-hb} \wedge (r, w') \notin \text{rpsi-hb} \quad (*) \\ \Rightarrow (w, r) \in \text{rpsi-hb} \wedge (w', r) \in \text{rpsi-hb} \end{aligned}$$

That is, given a transactional read r from location \mathbf{x} , and any two distinct non-transactional writes w, w' of the same value to \mathbf{x} , either (i) at least one of the writes RPSI-happen-after r ; or (ii) they both RPSI-happen-before r .

Observe that this does not hold of the program in Fig. 2. Note that this stipulation does not prevent two *transactions* to write the same value to a location \mathbf{x} . As such, in the absence of non-transactional writes, our RPSI implementation is equivalent to that of PSI in Sect. 4.2.

5.3 Implementation Soundness

The RPSI implementation in Fig. 2 is *sound*: for each consistent implementation graph G , a corresponding specification graph Γ can be constructed such that $\text{rpsi-consistent}(\Gamma)$ holds. In what follows we state our soundness theorem and briefly describe our construction of consistent specification graphs. We refer the reader to the technical appendix [4] for the full soundness proof.

Theorem 3 (Soundness). *Let P be a program that possibly mixes transactional and non-transactional code. If every RPSI-consistent execution graph of P satisfies the condition in (*), then for all RA-consistent implementation graphs G of the implementation in Fig. 2, there exists an RPSI-consistent specification graph Γ of the corresponding transactional program with the same program outcome.*

Constructing Consistent Specification Graphs. Constructing an RPSI-consistent specification graph from the implementation graph is similar to the corresponding PSI construction described in Sect. 4.3. More concretely, the events associated with non-transactional events remain unchanged and are simply added to the specification graph. On the other hand, the events associated with transactional events are adapted in a similar way to those of PSI in Sect. 4.3. In particular, observe that given an execution of the RPSI implementation with t transactions, as with the PSI implementation, the trace of each transaction $i \in \{1 \dots t\}$ is of the form $\theta_i = Ls_i \xrightarrow{po} FS_i \xrightarrow{po} S_i \xrightarrow{po} Ts_i \xrightarrow{po} Us_i$, with Ls_i , FS_i , S_i , Ts_i and Us_i denoting analogous sequences of events to those of PSI. The difference between an RPSI trace θ_i and a PSI one is in the FS_i and S_i sequences, obtaining the snapshot. In particular, the validation phases of FS_i and S_i in RPSI include an additional read for each location to rule out intermediate non-transactional writes. As in the PSI construction, for each transactional trace θ_i of our implementation, we construct a corresponding trace of the specification as $\theta'_i = B_i \xrightarrow{po} Ts'_i \xrightarrow{po} E_i$, with B_i , E_i and Ts'_i as defined in Sect. 4.3.

Given a consistent RPSI implementation graph $G = (E, po, rf, mo)$, let $G.\mathcal{NT} \triangleq G.E \setminus \bigcup_{i \in \{1 \dots t\}} \theta_i.E$ denote the non-transactional events of G . We construct a consistent RPSI specification graph $\Gamma = (E, po, rf, mo, T)$ such that:

- $\Gamma.E \triangleq G.\mathcal{NT} \cup \bigcup_{i \in \{1 \dots t\}} \theta'_i.E$ – the $\Gamma.E$ events comprise the non-transactional events in G and the events in each transactional trace θ'_i of the specification;
- $\Gamma.po \triangleq G.po|_{\Gamma.E}$ – the $\Gamma.po$ is that of $G.po$ restricted to the events in $\Gamma.E$;
- $\Gamma.rf \triangleq \bigcup_{i \in \{1 \dots t\}} RF_i \cup G.rf; [G.\mathcal{NT}]$ – the $\Gamma.rf$ is the union of RF_i relations for transactional reads as defined in Sect. 4.3, together with the $G.rf$ relation for non-transactional reads;
- $\Gamma.mo \triangleq G.mo|_{\Gamma.E}$ – the $\Gamma.mo$ is that of $G.mo$ restricted to the events in $\Gamma.E$;
- $\Gamma.T \triangleq \bigcup_{i \in \{1 \dots t\}} \theta'_i.E$, where for each $e \in \theta'_i.E$, we define $tx(e) = i$.

We refer the reader to the technical appendix [4] for the full proof demonstrating that the above construction of Γ yields a consistent specification graph.

5.4 Implementation Completeness

The RPSI implementation in Fig. 2 is *complete*: for each consistent specification graph Γ a corresponding implementation graph G can be constructed such that $RA\text{-consistent}(G)$ holds. We next state our completeness theorem and describe our construction of consistent implementation graphs. We refer the reader to the technical appendix [4] for the full completeness proof.

Theorem 4 (Completeness). *For all RPSI-consistent specification graphs Γ of a program, there exists an RA-consistent execution graph G of the implementation in Fig. 2 that has the same program outcome.*

Constructing Consistent Implementation Graphs. In order to construct an execution graph of the implementation G from the specification Γ , we follow similar steps as those in the corresponding PSI construction in Sect. 4.4. More concretely, the events associated with non-transactional events are unchanged and simply added to the implementation graph. For transactional events, given each trace θ'_i of a transaction in the specification, as before we construct an analogous trace of the implementation by inserting the appropriate events for acquiring and inspecting the version locks, as well as obtaining a snapshot. For each transaction class $\mathcal{T}_i \in \mathcal{T}/\text{st}$, we first determine its read and write sets as before and subsequently decide the order in which the version locks are acquired and inspected. This then enables us to construct the ‘reads-from’ and ‘modification-order’ relations for the events associated with version locks.

Given a consistent execution graph of the specification $\Gamma = (E, \text{po}, \text{rf}, \text{mo}, \mathcal{T})$, and a transaction class $\mathcal{T}_i \in \Gamma.\mathcal{T}/\text{st}$, we define $\text{WS}_{\mathcal{T}_i}$ and $\text{RS}_{\mathcal{T}_i}$ as described in Sect. 4.4. Determining the ordering of lock events hinges on a similar observation as that in the PSI construction. Given a consistent execution graph of the specification $\Gamma = (E, \text{po}, \text{rf}, \text{mo}, \mathcal{T})$, let for each location x the total order mo be given as: $w_1 \xrightarrow{\text{mo|imm}} \dots \xrightarrow{\text{mo|imm}} w_{n_x}$. This order can be broken into adjacent segments where the events of each segment are *either* non-transactional writes *or* belong to the *same* transaction. That is, given the transaction classes $\Gamma.\mathcal{T}/\text{st}$, the order above is of the following form where $\mathcal{T}_1, \dots, \mathcal{T}_m \in \Gamma.\mathcal{T}/\text{st}$ and for each such \mathcal{T}_i we have $x \in \text{WS}_{\mathcal{T}_i}$ and $w_{(i,1)} \dots w_{(i,n_i)} \in \mathcal{T}_i$:

$$\underbrace{w_{(1,1)} \xrightarrow{\text{mo|imm}} \dots \xrightarrow{\text{mo|imm}} w_{(1,n_1)}}_{\Gamma.\mathcal{NT} \cup \mathcal{T}_1} \xrightarrow{\text{mo|imm}} \dots \xrightarrow{\text{mo|imm}} \underbrace{w_{(m,1)} \xrightarrow{\text{mo|imm}} \dots \xrightarrow{\text{mo|imm}} w_{(m,n_m)}}_{\Gamma.\mathcal{NT} \cup \mathcal{T}_m}$$

Were this not the case and we had $w_1 \xrightarrow{\text{mo}} w \xrightarrow{\text{mo}} w_2$ such that $w_1, w_2 \in \mathcal{T}_i$ and $w \in \mathcal{T}_j \neq \mathcal{T}_i$, we would consequently have $w_1 \xrightarrow{\text{mo}^\top} w \xrightarrow{\text{mo}^\top} w_1$, contradicting the assumption that Γ is consistent. We thus define $\Gamma.\text{MO}_x = [\mathcal{T}_1 \dots \mathcal{T}_m]$.

Note that each transactional execution trace of the specification is of the form $\theta'_i = B_i \xrightarrow{\text{po}} Ts'_i \xrightarrow{\text{po}} E_i$, with B_i , E_i and Ts'_i as described in Sect. 4.4. For each such θ'_i , we construct a corresponding trace of our implementation as $\theta_i = Ls_i \xrightarrow{\text{po}} S_i \xrightarrow{\text{po}} Ts_i \xrightarrow{\text{po}} Us_i$, where Ls_i , Ts_i and Us_i are as defined in Sect. 4.4, and $S_i = tr_i^{x_1} \xrightarrow{\text{po}} \dots \xrightarrow{\text{po}} tr_i^{x_p} \xrightarrow{\text{po}} vr_i^{x_1} \xrightarrow{\text{po}} \dots \xrightarrow{\text{po}} vr_i^{x_p}$ denotes the sequence of events obtaining a tentative snapshot ($tr_i^{x_j}$) and subsequently validating it ($vr_i^{x_j}$). Each $tr_i^{x_j}$ sequence is of the form $ivr_i^{x_j} \xrightarrow{\text{po}} ir_i^{x_j} \xrightarrow{\text{po}} s_i^{x_j}$, with $ivr_i^{x_j}$, $ir_i^{x_j}$ and $s_i^{x_j}$ defined below (with fresh identifiers). Similarly, each $vr_i^{x_j}$ sequence is of the form $fr_i^{x_j} \xrightarrow{\text{po}} fvr_i^{x_j}$, with $fr_i^{x_j}$ and $fvr_i^{x_j}$ defined as follows (with fresh identifiers). We then define the rf relation for each of these read events in S_i in a similar way.

For each $(x, r) \in \text{RS}_{\mathcal{T}_i}$, when r (the event in the specification class \mathcal{T}_i that reads the value of x) reads from w in the specification graph ($(w, r) \in \Gamma.\text{rf}$), we add $(w, ir_i^{x_j})$ and $(w, fr_i^{x_j})$ to the rf of G (the first line of IRF_i^2 below). For version locks, as before if transaction \mathcal{T}_i also writes to x_j , then $ivr_i^{x_j}$ and $fvr_i^{x_j}$ events (reading and validating vx_j), read from the lock event in \mathcal{T}_i that acquired vx_j , namely $L_i^{x_j}$. Similarly, if \mathcal{T}_i does not write to x_j and it reads the value of x_j

written by the initial write, $init_x$, then $ivr_i^{x_j}$ and $fvr_i^{x_j}$ read the value written to vx_j by the initial write to vx , $init_{vx}$. Lastly, if transaction T_i does not write to x_j and it reads x_j from a write other than $init_x$, then $ir_i^{x_j}$ and $vr_i^{x_j}$ read from the unlock event of a transaction T_j (i.e. U_j^x), who has x in its write set and whose write to x , w_x , maximally ‘RPSI-happens-before’ r . That is, for all other such writes that ‘RPSI-happen-before’ r , then w_x ‘RPSI-happens-after’ them.

$$IRF_i^2 \triangleq \bigcup_{(x,r) \in RS_{T_i}} \left\{ \begin{array}{l} (w, ir_i^x), \\ (w, fr_i^x), \\ (w', ivr_i^x), \\ (w', fvr_i^x) \end{array} \middle| \begin{array}{l} (w, r) \in \Gamma.rf \wedge (x \in WS_{T_i} \Rightarrow w' = L_i^x) \\ \wedge (x \notin WS_{T_i} \wedge w = init_x \Rightarrow w' = init_{vx}) \\ \wedge (x \notin WS_{T_i} \wedge w \neq init_x \Rightarrow \\ \exists w_x, T_j. w_x \in T_j \cap \mathcal{W}_x \wedge w_x \xrightarrow{\text{rpsi-hb}} r \wedge w' = U_j^x \\ \wedge [\forall w'_x, T_k. w'_x \in T_k \cap \mathcal{W}_x \wedge w'_x \xrightarrow{\text{rpsi-hb}} r \Rightarrow w'_x \xrightarrow{\text{rpsi-hb}} w_x]) \end{array} \right\}$$

$$ir_i^{x_j} = fr_i^{x_j} = R(x_j, v) \quad s_i^{x_j} = W(s[x_j], v) \quad \text{s.t. } \exists w. (w, ir_i^{x_j}) \in IRF_i^2 \wedge val_w(w) = v$$

$$ivr_i^{x_j} = fvr_i^{x_j} = R(vx_j, v) \quad \text{s.t. } \exists w. (w, ivr_i^{x_j}) \in IRF_i^2 \wedge val_w(w) = v$$

We are now in a position to construct our implementation graph. Given a consistent execution graph Γ of the specification, we construct an execution graph of the implementation, $G = (E, po, rf, mo)$, such that:

- $G.E = \bigcup_{T_i \in \Gamma.T/st} \theta_i.E \cup \Gamma.NT$;
- $G.po$ is defined as $\Gamma.po$ extended by the po for the additional events of G , given by the θ_i traces defined above;
- $G.rf = \bigcup_{T_i \in \Gamma.T/st} (IRF_i^1 \cup IRF_i^2)$, with IRF_i^1 as in Sect. 4.4 and IRF_i^2 defined above;
- $G.mo = \Gamma.mo \cup \left(\bigcup_{T_i \in \Gamma.T/st} IMO_i \right)^+$, with IMO_i as defined in Sect. 4.4.

6 Conclusions and Future Work

We studied PSI, for the first time to our knowledge, as a consistency model for STMs as it has several advantages over other consistency models, thanks to its performance and monotonic behaviour. We addressed two significant drawbacks of PSI which prevent its widespread adoption. First, the absence of a simple lock-based reference implementation to allow the programmers to readily understand and reason about PSI programs. To address this, we developed a lock-based reference implementation of PSI in the RA fragment of C11 (using sequence locks), that is both sound and complete with respect to its declarative specification. Second, the absence of a formal PSI model in the presence of mixed-mode accesses. To this end, we formulated a declarative specification of RPSI (robust PSI) accounting for both transactional and non-transactional accesses. Our RPSI specification is an extension of PSI in that in the absence of non-transactional accesses it coincides with PSI. To provide a more intuitive account of RPSI, we developed a simple lock-based RPSI reference implementation by adjusting our PSI implementation. We established the soundness and completeness of our RPSI implementation against its declarative specification.

As directions of future work, we plan to build on top of the work presented here in three ways. First, we plan to explore possible lock-based reference implementations for PSI and RPSI in the context of other weak memory models, such as the full C11 memory models [9]. Second, we plan to study other weak transactional consistency models, such as SI [10], ALA (asymmetric lock atomicity), ELA (encounter-time lock atomicity) [28], and those of ANSI SQL, including RU (read-uncommitted), RC (read-committed) and RR (repeatable reads), in the STM context. We aim to investigate possible lock-based reference implementations for these models that would allow the programmers to understand and reason about STM programs with such weak guarantees. Third, taking advantage of the operational models provided by our simple lock-based reference implementations (those presented in this article as well as those in future work), we plan to develop reasoning techniques that would allow us to verify properties of STM programs. This can be achieved by either extending existing program logics for weak memory, or developing new program logics for currently unsupported models. In particular, we can reason about the PSI models presented here by developing custom proof rules in the existing program logics for RA such as [22, 39].

Acknowledgments. We thank the ESOP 2018 reviewers for their constructive feedback. This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project “RustBelt”, under the European Union’s Horizon 2020 Framework Programme (grant agreement no. 683289). The second author was additionally partly supported by Len Blavatnik and the Blavatnik Family foundation.

References

1. The Clojure Language: Refs and Transactions. <http://clojure.org/refs>
2. Haskell STM. <http://hackage.haskell.org/package/stm-2.2.0.1/docs/Control-Concurrent-STM.html>
3. Software transactional memory (Scala). <https://doc.akka.io/docs/akka/1.2/scala/stm.html>
4. Technical appendix for this paper. <http://plv.mpi-sws.org/transactions/>
5. Generalized isolation level definitions. In: Proceedings of the 16th International Conference on Data Engineering (2000)
6. Technical specification for C++ extensions for transactional memory (2015). <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf>
7. Adya, A.: Weak consistency: a generalized theory and optimistic implementations for distributed transactions. Ph.D. thesis, MIT (1999)
8. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* **36**(2), 7:1–7:74 (2014)
9. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 55–66 (2011)
10. Berenson, H., Bernstein, P., Gray, J., Melton, J., O’Neil, E., O’Neil, P.: A critique of ANSI SQL isolation levels. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, pp. 1–10 (1995)

11. Bieniusa, A., Fuhrmann, T.: Consistency in hindsight: a fully decentralized STM algorithm. In: Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, pp. 1–12 (2010)
12. Blundell, C., Lewis, E.C., Martin, M.M.K.: Deconstructing transactions: the subtleties of atomicity. In: 4th Annual Workshop on Duplicating, Deconstructing, and Debunking (2005)
13. Boehm, H.J.: Can seqlocks get along with programming language memory models? In: Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, pp. 12–20 (2012)
14. Cerone, A., Gotsman, A.: Analysing snapshot isolation. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, pp. 55–64 (2016)
15. Cerone, A., Gotsman, A., Yang, H.: Transaction chopping for parallel snapshot isolation. In: Moses, Y. (ed.) DISC 2015. LNCS, vol. 9363, pp. 388–404. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48653-5_26
16. Daudjee, K., Salem, K.: Lazy database replication with snapshot isolation. In: Proceedings of the 32nd International Conference on Very Large Data Bases, pp. 715–726 (2006)
17. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 48–60 (2005)
18. Hemminger, S.: Fast reader/writer lock for gettimeofday 2.5.30. <http://lwn.net/Articles/7388/>
19. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 289–300 (1993)
20. Hickey, R.: The Clojure programming language. In: Proceedings of the 2008 Symposium on Dynamic Languages, p. 1:1 (2008)
21. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 649–662 (2016)
22. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015, Part II. LNCS, vol. 9135, pp. 311–323. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_25
23. Lameter, C.: Effective synchronization on Linux/NUMA systems (2005). <http://www.lameter.com/gelato2005.pdf>
24. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* **28**(9), 690–691 (1979)
25. Litz, H., Cheriton, D., Firoozshahian, A., Azizi, O., Stevenson, J.P.: SI-TM: reducing transactional memory abort rates through snapshot isolation. *SIGPLAN Not.* **42**(1), 383–398 (2014)
26. Litz, H., Dias, R.J., Cheriton, D.R.: Efficient correction of anomalies in snapshot isolation transactions. *ACM Trans. Archit. Code Optim.* **11**(4), 65:1–65:24 (2015)
27. Martin, M., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.* **5**(2), 17 (2006)
28. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R.L., Saha, B., Welc, A.: Single global lock semantics in a weakly atomic STM. *SIGPLAN Not.* **43**(5), 15–26 (2008)
29. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, pp. 391–407 (2009)

30. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, pp. 251–264 (2010)
31. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* **2**(POPL), 19:1–19:29 (2017). <http://doi.acm.org/10.1145/3158107>
32. Rajwar, R., Goodman, J.R.: Speculative lock elision: enabling highly concurrent multithreaded execution. In: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 294–305 (2001)
33. Serrano, D., Patino-Martinez, M., Jimenez-Peris, R., Kemme, B.: Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In: Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing, pp. 290–297 (2007)
34. Sewell, P., Sarkar, S., Owens, S., Zappa Nardelli, F., Myreen, M.O.: x86-TSO: a rigorous and usable programmer’s model for x86 multiprocessors. *Commun. ACM* **53**(7), 89–97 (2010)
35. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 204–213 (1995)
36. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 385–400 (2011)
37. CORPORATE SPARC International Inc.: The SPARC Architecture Manual: Version 8 (1992)
38. CORPORATE SPARC International Inc.: The SPARC Architecture Manual (Version 9) (1994)
39. Vafeiadis, V., Narayan, C.: Relaxed separation logic: a program logic for c11 concurrency. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, pp. 867–884 (2013)


Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Eventual Consistency for CRDTs

Radha Jagadeesan and James Riely (✉) 

DePaul University, Chicago, USA
{rjagadeesan,jriely}@cs.depaul.edu

Abstract. We address the problem of *validity* in eventually consistent (EC) systems: In what sense does an EC data structure satisfy the sequential specification of that data structure? Because EC is a very weak criterion, our definition does not describe every EC system; however it is expressive enough to describe any Convergent or Commutative Replicated Data Type (CRDT).

1 Introduction

In a replicated implementation of a data structure, there are two impediments to requiring that all replicas achieve consensus on a global total order of the operations performed on the data structure (Lamport 1978): (a) the associated serialization bottleneck negatively affects performance and scalability (*e.g.* see (Ellis and Gibbs 1989)), and (b) the CAP theorem imposes a tradeoff between consistency and partition-tolerance (Gilbert and Lynch 2002).

In systems based on *optimistic replication* (Vogels 2009; Saito and Shapiro 2005), a replica may execute an operation without synchronizing with other replicas. If the operation is a mutator, the other replicas are updated asynchronously. Due to the vagaries of the network, the replicas could receive and apply the updates in possibly different orders.

For sequential systems, the correctness problem is typically divided into two tasks: proving *termination* and proving *partial correctness*. Termination requires that the program eventually halt on all inputs, whereas partial correctness requires that the program only returns results that are allowed by the specification.

For replicated systems, the analogous goals are *convergence* and *validity*. Convergence requires that all replicas eventually agree. Validity requires that they agree on something sensible. In a replicated list, for example, if the only value put into the list is 1, then convergence ensures that all replicas eventually see the same value for the head of the list; validity requires that the value be 1.

Convergence has been well-understood since the earliest work on replicated systems. Convergence is typically defined as *eventual consistency*, which requires that once all messages are delivered, all replicas have the same state. *Strong eventual consistency* (SEC) additionally requires convergence for all subsets of messages: replicas that have seen the same messages must have the same state.

Perhaps surprisingly, finding an appropriate definition of validity for replicated systems remains an open problem. There are solutions which use concurrent specifications, discussed below. But, as Shavit (2011) noted:

“It is infinitely easier and more intuitive for us humans to specify how abstract data structures behave in a sequential setting, where there are no interleavings. Thus, the standard approach to arguing the safety properties of a concurrent data structure is to specify the structure’s properties sequentially, and find a way to map its concurrent executions to these ‘correct’ sequential ones.”

In this paper we give the first definition of validity that is both (1) derived from standard sequential specifications and (2) validates the examples of interest.

We take the “examples of interest” to be *Convergent/Commutative Replicated Data Types* (CRDTs). These are replicated structures that obey certain monotonicity or commutativity properties. As an example of a CRDT, consider the *add-wins set*, also called an “observed remove” set in Shapiro et al. (2011a). The add-wins set behaves like a sequential set if add and remove operations on the same element are ordered. The concurrent execution of an add and remove result in the element being added to the set; thus the remove is ignored and the “add wins.” This concurrent specification is very simple, but as we will see in the next section, it is quite difficult to pin down the relationship between the CRDT and the sequential specification used in the CRDT’s definition. This paper is the first to successfully capture this relationship.

Many replicated data types are CRDTs, but not all (Shapiro et al. 2011a). Notably, Amazon’s Dynamo (DeCandia et al. 2007) is not a CRDT. Indeed, interest in CRDTs is motivated by a desire to avoid the well-know concurrency anomalies suffered by Dynamo and other ad hoc systems (Bieniusa et al. 2012).

Shapiro et al. (2011b) introduced the notion of CRDT and proved that every CRDT has an SEC implementation. Their definition of SEC includes convergence, but not validity.

The validity requirement can be broken into two components. We describe these below using the example of a list data type that supports only two operations: the mutator *put*, which adds an element to the end of the list, and the query *q*, which returns the state of the list. This structure can be specified as a set of strings such as “*put*(1); *put*(3); *q*=[1,3]” and “*put*(1); *put*(2); *put*(3); *q*=[1,2,3]”.

- *Linearization* requires that a response be consistent with some specification string. A state that received *put*(1) and *put*(3), may report *q*=[1,3] or *q*=[3,1], but not *q*=[2,1,3], since 2 has not been put into the list.
- *Monotonicity* requires that states evolve in a sensible way. We might permit the state *q*=[1,3] to evolve into *q*=[1,2,3], due to the arrival of action *put*(2). But we would not expect that *q*=[1,3] could evolve into *q*=[3,1], since the data type does not support deletion or reordering.

Burckhardt et al. (2012) provide a formal definition of validity using partial orders over events: linearizations respect the partial order on events; monotonicity

is ensured by requiring that evolution extends the partial order. Similar definitions can be found in Jagadeesan and Riely (2015) and Perrin et al. (2015). Replicated data structures that are sound with respect to this definition enjoy many good properties, which we discuss throughout this paper. However, this notion of correctness is not general enough to capture common CRDTs, such as the add-wins set.

This lack of expressivity lead Burckhardt et al. (2014) to abandon notions of validity that appeal directly to a sequential specification. Instead they work directly with *concurrent* specifications, formalizing the style of specification found informally in Shapiro et al. (2011b). This has been a fruitful line of work, leading to proof rules (Gotsman et al. 2016) and extensions (Bouajjani et al. 2014). See (Burckhardt 2014; Viotti and Vukolic 2016) for a detailed treatment.

Positively, concurrent specifications can be used to validate any replicated structure, including CRDTs as well as anomalous structures such as Dynamo. Negatively, concurrent specifications have no the clear connection to their sequential counterparts. In this paper, we restore this connection. We arrive at a definition of SEC that admits CRDTs, but rejects Dynamo.

The following “corner cases” are a useful sanity-check for any proposed notion of validity.

- The principle of *single threaded semantics* (PSTS) (Haas et al. 2015) states that if an execution uses only a single replica, it should behave according to the sequential semantics.
- The principle of *single master* (PSM) (Budhiraja et al. 1993) states that if all mutators in an execution are initiated at a single replica, then the execution should be linearizable (Herlihy and Wing 1990).
- The principle of *permutation equivalence* (PPE) (Bieniusa et al. 2012) states that “if all sequential permutations of updates lead to equivalent states, then it should also hold that concurrent executions of the updates lead to equivalent states.”

PSTS and PSM say that a replicated structure should behave sequentially when replication is not used. PPE says that the order of independent operations should not matter. Our definition implies all three conditions. Dynamo fails PPE (Bieniusa et al. 2012), and thus fails to pass our definition of SEC.

In the next section, we describe the validity problem and our solution in detail, using the example of a binary set. The formal definitions follow in Sect. 3. We state some consequences of the definition and prove that the add-wins set satisfies our definition. In Sect. 4, we describe a collaborative text editor and prove that it is SEC. In Sect. 5 we characterize the programmer’s view of a CRDT by defining the *most general* CRDT that satisfies a given sequential specification. We show that any program that is correct using the most general CRDT will be correct using a more restricted CRDT. We also show that our validity criterion for SEC is *local* in the sense of Herlihy and Wing (1990): independent structures can be verified independently. In Sect. 6, we apply these results to prove the correctness of a graph that is implemented using two SEC sets.

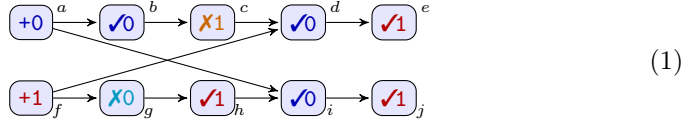
Our work is inspired by the study of relaxed memory, such as (Alglave 2012). In particular, we have drawn insight from the RMO model of Higham and Kawash (2000).

2 Understanding Replicated Sets

In this section, we motivate the definition of SEC using replicated sets as an example. The final definition is quite simple, but requires a fresh view of both executions and specifications. We develop the definition in stages, each of which requires a subtle shift in perspective. Each subsection begins with an example and ends with a summary.

2.1 Mutators and Non-mutators

An *implementation* is a set of *executions*. We model executions abstractly as labelled partial orders (LPOs). The ordering of the LPO captures the history that precedes an event, which we refer to as *visibility*.



Here the events are a through j , with labels $+0$, $+1$, etc., and order represented by arrows. The LPO describes an execution with two replicas, shown horizontally, with time passing from left to right. Initially, the top replica receives a request to add 0 to the set ($+0^a$). Concurrently, the bottom replica receives a request to add 1 ($+1^b$). Then each replica is twice asked to report on the items contained in the set. At first, the top replica replies that 0 is present and 1 is absent ($✓0^b ✗1^c$), whereas the bottom replica answers with the reverse ($✗0^g ✓1^h$). Once the add operations are visible at all replicas, however, the replicas give the same responses ($✓0^d ✓1^e$ and $✓0^i ✓1^j$).

LPOs with non-interacting replicas can be denoted compactly using sequential and parallel composition. For example, the prefix of (1) that only includes the first three events at each replica can be written $(+0^a; ✓0^b; ✗1^c) \parallel (+1^f; ✗0^g; ✓1^h)$.

A *specification* is a set of *strings*. Let SET be the specification of a sequential set with elements 0 and 1. Then we expect that SET includes the string “ $+0 ✓0 ✗1$ ”, but not “ $+0 ✗0 ✓1$ ”. Indeed, each specification string can uniquely be extended with either $✓0$ or $✗0$ and either $✓1$ or $✗1$.

There is an isomorphism between strings and labelled *total* orders. Thus, specification strings correspond to the restricted class of LPOs where the visibility relation provides a total order.

Linearizability (Herlihy and Wing 1990) is the gold standard for concurrent correctness in tightly coupled systems. Under linearizability, an execution is valid if there exists a linearization τ of the events in the execution such that for every event e , the prefix of e in τ is a valid specification string.

Execution (1) is not linearizable. The failure can already be seen in the sub-LPO $(+0^a; \mathbf{X1}^c) \parallel (+1^f; \mathbf{X0}^g)$. Any linearization must have either $+1^f$ before $\mathbf{X1}^c$ or $+0^a$ before $\mathbf{X0}^g$. In either case, the linearization is invalid for SET.

Although it is not linearizable, execution (1) is admitted by every CRDT SET in Shapiro et al. (2011a). To validate such examples, Burckhardt et al. (2012) develop a weaker notion of validity by dividing labels into *mutators* and *accessors* (also known as non-mutators). Similar definitions appear in Jagadeesan and Riely (2015) and Perrin et al. (2015). Mutators change the state of a replica, and accessors report on the state without changing it. For SET, the mutators \mathbf{M} and non-mutators $\overline{\mathbf{M}}$ are as follows.

$\mathbf{M} = \{+0, -0, +1, -1\}$, representing addition and removal of bits 0 and 1.

$\overline{\mathbf{M}} = \{\mathbf{X0}, \check{0}, \mathbf{X1}, \check{1}\}$, representing membership tests returning false or true.

Define the *mutator prefix* of an event e to include e and the *mutators* visible to e . An execution is valid if there exists a linearization of the execution, τ , such that for every event e , the *mutator prefix* of e in τ is a valid specification string.

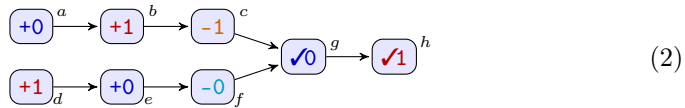
It is straightforward to see that execution (1) satisfies this weaker criterion. For both $\check{0}^b$ and $\mathbf{X1}^c$, the mutator prefix is $+0^a$. This includes $+0^a$ but not $+1^f$, and thus their answers are validated. Symmetrically, the mutator prefixes of $\mathbf{X0}^g$ and $\check{1}^h$ only include $+1^f$. The mutator prefixes for the final four events include both $+0^a$ and $+1^f$, but none of the prior accessors.

Summary: Convergent states must agree on the final order of mutators, but intermediate states may see incompatible subsequences of this order. By restricting attention to mutator prefixes, the later states need not linearize these incompatible views of the partial past.

This relaxation is analogous to the treatment of non-mutators in update serializability (Hansdah and Patnaik 1986; Garcia-Molina and Wiederhold 1982), which requires a global serialization order for mutators, ignoring non-mutators.

2.2 Dependency

The following LPO is admitted by the add-wins SET discussed in the introduction.



In any CRDT implementation, the effect of $+1^b$ is negated by the subsequent -1^c . The same reasoning holds for $+0^e$ and -0^f . In an add-wins set, however, the *concurrent* adds, $+0^a$ and $+1^d$, win over the deletions. Thus, in the final state both 0 and 1 are present.

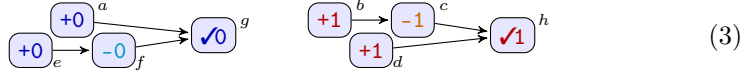
This LPO is not valid under the definition of the previous subsection: Since $\check{0}^g$ and $\check{1}^h$ see the same mutators, they must agree on a linearization of $(+0^a; +1^b; -1^c) \parallel (+1^d; +0^e; -0^f)$. Any linearization must end in either -1^c or -0^f ; thus it is not possible for both $\check{0}^g$ and $\check{1}^h$ to be valid.

Similar issues arise in relaxed memory models, where program order is often relaxed between uses of independent variables (Alglaive et al. 2014). Generalizing, we write $m \# n$ to indicate that labels m and n are dependent. Dependency is a property of a *specification*, not an implementation. Our results only apply to specifications that support a suitable notion of dependency, as detailed in Sect. 3. For SET, $\#$ is an equivalence relation with two equivalence classes, corresponding to actions on the independent values 0 and 1.

$$\# = \{+0, -0, \textcolor{teal}{X}0, \checkmark 0\}^2 \cup \{+1, -1, \textcolor{teal}{X}1, \checkmark 1\}^2, \text{ where } D^2 = D \times D.$$

While the dependency relation for SET is an equivalence, this is not required: In Sect. 4 we establish the correctness of collaborative text editing protocol with an intransitive dependency relation.

The *dependent restriction* of (2) is as follows.



In the previous subsection, we defined validity using the *mutator prefix* of an event. We arrive at a weaker definition by restricting attention to the *mutator prefix of the dependent restriction*.

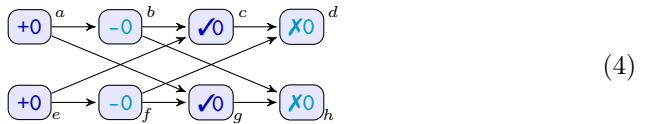
Under this definition, execution (2) is validated: Any interleaving of the strings $+0^e -0^f +0^a \checkmark 0^g$ and $+1^b -1^c +1^d \checkmark 1^h$ linearizes the dependent restriction of (2) given in (3).

Summary: CRDTs allow independent mutators to commute. We formalize this intuition by restricting attention to mutator prefixes of the dependent restriction. The CRDT must respect program order between dependent operations, but is free to reorder independent operations.

This relaxation is analogous to the distinction between *program order* and *preserved program order* (PPO) in relaxed memory models (Higham and Kawash 2000; Alglaive 2012). Informally, PPO is the suborder of program order that removes order between independent memory actions, such as successive reads on different locations without an intervening memory barrier.

2.3 Puns

The following LPO is admitted by the add-wins SET.



As in execution (2), the add $+0^a$ is undone by the following remove -0^b , but the concurrent add $+0^e$ wins over -0^b , allowing $\checkmark 0^c$. In effect, $\checkmark 0^c$ sees the order of the mutators as $+0^a -0^b +0^e$. Symmetrically, $\checkmark 0^g$ sees the order as $+0^e -0^f +0^a$.

While this is very natural from the viewpoint of a CRDT, there is no linearization of the events that includes both $+0^a -0^b +0^e$ and $+0^e -0^f +0^a$, since $+0^a$ and $+0^e$ must appear in different orders.

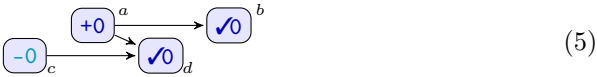
Indeed, this LPO is not valid under the definition of the previous subsection. First note that all events are mutually dependent. To prove validity we must find a linearization that satisfies the given requirements. Any linearization of the mutators must end in either -0^b or -0^f . Suppose we choose $+0^a -0^b +0^e -0^f$ and look for a mutator prefix to satisfy $\check{0}^g$. (All other choices lead to similar problems.) Since -0^f precedes $\check{0}^g$ and is the last mutator in our chosen linearization, every possible witness for $\check{0}^g$ must end with mutator -0^f . Indeed the only possible witness is $+0^a +0^e -0^f \check{0}^g$. However, this is not a valid specification string.

The problem is that we are linearizing *events*, rather than *labels*. If we shift to linearizing labels, then execution (4) is allowed. Fix the final order for the mutators to be $+0 -0 +0 -0$. The execution is allowed if we can find a subsequence that linearizes the labels visible at each event. It suffices to choose the witnesses as follows. In the table, we group events with a common linearization together.

$+0^a, +0^e: +0$	$\check{0}^c, \check{0}^g: +0-0+0\check{0}$
$-0^b, -0^f: +0-0$	$\cancel{0}^d, \cancel{0}^h: +0-0+0-0\cancel{0}$

Each of these is a valid specification string. In addition, looking only at mutators, each is a subsequence of $+0 -0 +0 -0$.

In execution (4), each of the witnesses is actually a *prefix* of the final mutator order, but, in general, it is necessary to allow *subsequences*.



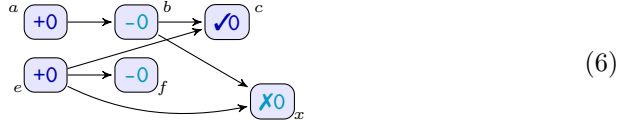
Execution (5) is admitted by the add-wins SET. It is validated by the final mutator sequence $-0 +0$. The mutator prefix $+0$ of b is a subsequence of $-0 +0$, but not a prefix.

Summary: While dependent events at a single replica must be linearized in order, concurrent events may slip anywhere into the linearization. A CRDT may *pun* on concurrent events with same label, using them in different positions at different replicas. Thus a CRDT may establish a final total over the labels of an execution even when there is no linearization of the events.

2.4 Frontiers

In the introduction, we mentioned that the validity problem can be decomposed into the separate concerns of *linearizability* and *monotonicity*. The discussion thus far has centered on the appropriate meaning of linearizability for CRDTs. In this subsection and the next, we look at the constraints imposed by monotonicity.

Consider the prefix $\{+0^a, -0^b, +0^e, \checkmark 0^c, -0^f\}$ of execution (4), extended with action $\times 0^x$, with visibility order as follows.



This execution is *not strong* EC, since $\checkmark 0^c$ and $\times 0^x$ see exactly the same mutators, yet provide incompatible answers.

Unfortunately, execution (6) is valid by the definition given in the previous section: The witnesses for a – f are as before. In particular, the witness for $\checkmark 0^c$ is “ $+0-0+0\checkmark 0$ ”. The witness for $\times 0^x$ is “ $+0+0-0\times 0$ ”. In each case, the mutator prefix is a subsequence of the global mutator order “ $+0-0+0-0$ ”.

It is well known that punning can lead to bad jokes. In this case, the problem is that $\times 0^x$ is punning on a concurrent -0 that cannot be matched by a visible -0 in its history: the execution -0 that is visible to $\times 0^x$ must appear *between* the two $+0$ operations; the specification -0 that is used by $\times 0^x$ must appear *after*. The final states of execution (4) have seen both remove operations, therefore the pun is harmless there. But $\checkmark 0^c$ and $\times 0^x$ have seen only one remove. They must agree on how it is used.

Up to now, we have discussed the linearization of each event in isolation. We must also consider the relationship between these linearizations. When working with linearizations of *events*, it is sufficient to require that the linearization chosen for each event be a subsequence for the linearization chosen for each visible predecessor; since events are unique, there can be no confusion in the linearization about which event is which. Execution (6) shows that when working with linearizations of *labels*, it is insufficient to consider the relationship between individual events. The linearization “ $+0+0-0\times 0$ ” chosen for $\times 0^x$ is a supersequence of those chosen for its predecessors: “ $+0$ ” for $+0^e$ and “ $+0-0$ ” for -0^b . The linearization “ $+0-0+0\checkmark 0$ ” chosen for $\checkmark 0^c$ is also a supersequence for the same predecessors. And yet, $\checkmark 0^c$ and $\times 0^x$ are incompatible states.

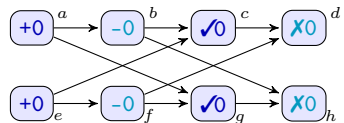
Sequential systems have a single state, which evolves over time. In distributed systems, each replica has its own state, and it is this *set* of states that evolves. Such a set of states is called a (*consistent*) *cut* (Chandy and Lamport 1985).

A *cut* of an LPO is a sub-LPO that is down-closed with respect to visibility. The *frontier* of cut is the set of maximal elements. For example, there are 14 frontiers of execution (6): the singletons $\{+0^a\}$, $\{-0^b\}$, $\{\checkmark 0^c\}$, $\{+0^e\}$, $\{-0^f\}$, $\{\times 0^x\}$, the pairs $\{+0^a, +0^e\}$, $\{+0^a, -0^f\}$, $\{-0^b, +0^e\}$, $\{-0^b, -0^f\}$, $\{\checkmark 0^c, -0^f\}$, $\{\checkmark 0^c, \times 0^x\}$, $\{\times 0^x, -0^f\}$, and the triple $\{\checkmark 0^c, \times 0^x, -0^f\}$. As we explain below, we consider non-mutators in isolation. Thus we do not consider the last four cuts, which include a non-mutator with other events. That leaves 10 frontiers. The definition of the previous section only considered the 6 singletons. Singleton frontiers are generated by *pointed cuts*, with a single maximal element.

When applied to frontiers, the monotonicity requirement invalidates execution (6). Monotonicity requires that the linearization chosen for a frontier be a

subsequence of the linearization chosen for any extension of that frontier. If we are to satisfy state $\check{0}^c$ in execution (6), the frontier $\{-0^b, +0^e\}$ must linearize to “ $+0-0+0$ ”. If we are to satisfy state $\check{X}0^x$, the frontier $\{-0^b, +0^e\}$ must linearize to “ $+0+0-0$ ”. Since we require a unique linearization for each frontier, the execution is disallowed.

Since CRDTs execute non-mutators locally, it is important that we ignore frontiers with multiple non-mutators. Recall execution (4):



There is no specification string that linearizes the cut with frontier $\{\check{0}^c, \check{0}^g\}$, since we cannot have $\check{0}$ immediately after -0 . If we consider only pointed cuts for non-mutators, then the execution is SEC, with witnesses as follows.

$\{+0^a\}, \{+0^e\}$: $+0$	$\{\check{0}^c\}, \{\check{0}^g\}$:	$+0-0+0\check{0}$
$\{+0^a, +0^e\}$: $+0+0$	$\{-0^b, -0^f\}$: $+0-0+0-0$
$\{-0^b\}, \{-0^f\}$: $+0-0$	$\{\check{X}0^d\}, \{\check{X}0^h\}$:	$+0-0+0-0\check{X}0$
$\{-0^b, +0^e\}, \{+0^a, -0^f\}$:	$+0-0+0$		

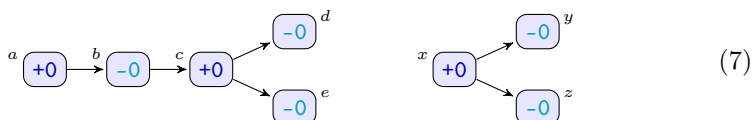
In order to validate non-mutators, we *must* consider singleton non-mutator frontiers. The example shows that we *must not* consider frontiers with multiple non-mutators. There is some freedom in the choices otherwise. For SET, we can “saturate” an execution with accessors by augmenting the execution with accessors that witness each cut of the mutators. In a saturated execution, it is sufficient to consider only the *pointed accessor* cuts, which end in a maximal accessor. For non-saturated executions, we are forced to examine each mutator cut: it is possible that a future accessor extension may witness that cut. The status of “mixed” frontiers, which include mutators with a single maximal non-mutator, is open for debate. We choose to ignore them, but the definition does not change if they are included.

Summary: A CRDT must have a strategy for linearizing all mutator labels, even in the face of partitions. In order to ensure *strong* EC, the definition must consider sets of events across multiple replicas. Because non-mutators are resolved locally, SEC must ignore frontiers with multiple non-mutators.

Cuts and frontiers are well-known concepts in the literature of distributed systems (Chandy and Lamport 1985). It is natural to consider frontiers when discussing the evolving correctness of a CRDT.

2.5 Stuttering

Consider the following execution.



This LPO represents a partitioned system with events a – e in one partition and x – z in the other. As the partition heals, we must be able to account for the intermediate states. Because of the large number of events in this example, we have elided all accessors. We will present the example using the semantics of the add-wins set. Recall that the add-wins set validates $\checkmark 0$ if and only if there is a maximal $+0$ beforehand. Thus, a replica that has seen the cut with frontier $\{+0^a, -0^y, -0^z\}$ must answer $\checkmark 0$, whereas a replica that has seen $\{-0^b, -0^y, -0^z\}$ must answer $\times 0$.

Any linearization of $\{+0^a, -0^y, -0^z\}$ must end in $+0$, since the add-win set must reply $\checkmark 0$: the only possibility is “ $+0-0-0+0$ ”. The linearization of $\{-0^b, -0^y, -0^z\}$ must end in -0 . If it must be a supersequence, the only possibility is “ $+0-0-0+0-0$ ”. Taking one more step on the left, $\{+0^c, -0^y, -0^z\}$ must linearize to “ $+0-0-0+0-0+0$ ”. Thus the final state $\{-0^d, -0^e, -0^y, -0^z\}$ must linearize to “ $+0-0-0+0-0+0-0$ ”. Reasoning symmetrically, the linearization of $\{-0^d, -0^e, +0^x\}$ must be “ $+0-0+0-0-0+0$ ”, and thus the final $\{-0^d, -0^e, -0^y, -0^z\}$ must linearize to “ $+0-0+0-0-0+0-0-0$ ”. The constraints on the final state are incompatible. Each of these states can be verified in isolation; it is the relation between them that is not satisfiable.

Recall that monotonicity requires that the linearization chosen for a frontier be a *subsequence* of the linearization chosen for any extension of that frontier. The difficulty here is that subsequence relation ignores the similarity between “ $+0-0-0+0-0+0-0-0$ ” and “ $+0-0+0-0-0+0-0-0$ ”. Neither of these is a subsequence of the other, yet they capture exactly the same sequence of *states*, each with six alternations between $\times 0$ and $\checkmark 0$. The canonical state-based representative for these sequences is “ $+0-0+0-0+0-0$ ”.

CRDTs are defined in terms of states. In order to relate CRDTs to sequential specifications, it is necessary to extract information about states from the specification itself. Adapting Brookes (1996), we define strings as *stuttering equivalent* (notation $\sigma \sim \tau$) if they pass through the same states. So $+0+1+0 \sim +0+1$ but $+0-0+0 \not\sim +0$. If we consider subsequences up to stuttering, then execution (7) is SEC, with witnesses as follow:

$\{a\}, \{x\}, \{a, x\}$: $+0$
$\{b\}, \{y\}, \{y, z\}, \{z\}$: $+0-0$
$\{a, y\}, \{a, y, z\}, \{a, z\}, \{b, x\}$: $+0-0+0$
$\{b, y\}, \{b, y, z\}, \{b, z\}, \{d\}, \{d, e\}, \{e\}$: $+0-0+0-0$
$\{c, y\}, \{c, y, z\}, \{c, z\}, \{d, x\}, \{d, e, x\}, \{e, x\}$: $+0-0+0-0+0$
$\{d, y\}, \{d, y, z\}, \{d, z\},$	
$\{e, y\}, \{e, y, z\}, \{e, z\}, \{d, e, y\}, \{d, e, y, z\}, \{d, e, z\}$: $+0-0+0-0+0-0$

Recall that without stuttering, we deduced that $\{+0^c, -0^y, -0^z\}$ must linearize to “ $+0-0-0+0-0+0$ ” and $\{-0^d, -0^e, +0^x\}$ must linearize to “ $+0-0+0-0-0+0$ ”. Under stuttering equivalence, these are the same, with canonical representative “ $+0-0+0-0+0$ ”. Thus, monotonicity under stuttering allows both linearizations to be extended to satisfy the final state $\{-0^d, -0^e, -0^y, -0^z\}$, which has canonical representative “ $+0-0+0-0+0-0$ ”.

Summary: CRDTs are described in terms of convergent states, whereas specifications are described as strings of actions. Actions correspond to labels in the LPO of an execution. Many strings of actions may lead to equivalent states. For example, idempotent actions can be applied repeatedly without modifying the state.

The stuttering equivalence of Brookes (1996) addresses this mismatch. In order to capture the validity of CRDTs, the definition of subsequence must change from a definition over individual specification strings to a definition over *equivalence classes* of strings *up to stuttering*.

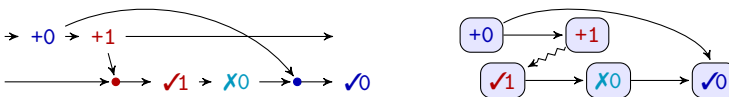
3 Eventual Consistency for CRDTs

This section formalizes the intuitions developed in Sect. 2. We define executions, specifications and strong eventual consistency (SEC). We discuss properties of eventual consistency and prove that the add-wins set is SEC.

3.1 Executions

An execution realizes *causal delivery* if, whenever an event is received at a replica, all predecessors of the event are also received. Most of the CRDTs in Shapiro et al. (2011a) assume causal delivery, and we assumed it throughout the introductory section. There are costs to maintaining causality, however, and not all CRDTs assume that executions incur these costs. In the formal development, we allow non-causal executions.

Shapiro et al. (2011a) draw executions as timelines, explicitly showing the delivery of remote mutators. Below left, we give an example of such a timeline.



This is a non-causal execution: at the bottom replica, +1 is received before +0, even though +0 precedes +1 at the top replica.

Causal executions are naturally described as Labelled Partial Orders (LPOs), which are transitive and antisymmetric. Section 2 presented several examples of LPOs. To capture non-causal systems, we move to *Labelled Visibility Orders* (LVOs), which are merely acyclic. Acyclicity ensures that the transitive closure of an LVO is an LPO. The right picture above shows the LVO corresponding to the timeline on the left. The zigzag arrow represents an intransitive communication. When drawing executions, we use straight lines for “transitive” edges, with the intuitive reading that “this and all preceding actions are delivered”.

LVOs arise directly due to non-causal implementations. As we will see in Sect. 4, they also arise via projection from an LPO.

LVOs are unusual in the literature. To make this paper self-contained, we define the obvious generalizations of concepts familiar from LPOs, including isomorphism, suborder, restriction, maximality, downclosure and cut.

Fix a set \mathbf{L} of labels. A *Labelled Visibility Order* (LVO, also known as an *execution*) is a triple $u = \langle \mathbf{E}_u, \lambda_u, \rightsquigarrow_u \rangle$ where \mathbf{E}_u is a finite set of events, $\lambda_u \in (\mathbf{E}_u \mapsto \mathbf{L})$ and $\rightsquigarrow_u \subseteq (\mathbf{E}_u \times \mathbf{E}_u)$ is reflexive and acyclic.

Let u, v range over LVOs. Many concepts extend smoothly from LPOs to LVOs.

- *Isomorphism*: Write $u =_{\text{iso}} v$ when u and v differ only in the carrier set. We are often interested in the isomorphism class of an LVO.
- *Pomset*: We refer to the isomorphism class of an LVO as a *pomset*. Pomset abbreviates *Partially Ordered Multiset* (Plotkin and Pratt 1997). We stick with the name “pomset” here, since “vomset” is not particularly catchy.
- *Suborder*: Write $u \subseteq v$ when $\mathbf{E}_u \subseteq \mathbf{E}_v$, $\lambda_u \subseteq \lambda_v$, $\rho_u \subseteq \rho_v$, and $(\rightsquigarrow_v) \subseteq (\rightsquigarrow_u)$.
- *Restriction*.¹ When $D \subseteq \mathbf{E}_v$, define $v \upharpoonright D = \langle D, \lambda_v \upharpoonright D, \rightsquigarrow_v \upharpoonright D \rangle$. Restriction lifts subsets to suborders: $v \upharpoonright D$ denotes the sub-LVO derived from a subset D of events. See Sect. 2.2 for an example of restriction.
- *Maximal elements*: $\max(v) = \{d \in \mathbf{E}_v \mid \nexists e \in (\mathbf{E}_v \setminus \{d\}). d \rightsquigarrow_v e\}$. We say that d is maximal for v when if $d \in \max(v)$.
- *Non-maximal suborder*: $\overline{\max}(v) = v \upharpoonright (\mathbf{E}_v \setminus \max(v))$. $\overline{\max}(v)$ is the suborder with the maximal elements removed.
- *Downclosure*: D is *downclosed* for v if $D \subseteq \{e \in \mathbf{E}_v \mid \exists d \in D. d \rightsquigarrow_v e\}$.
- *Cut*: u is a *cut* of v if $u \subseteq v$ and \mathbf{E}_u is downclosed for v . Let $\text{cuts}(v)$ be the set of all cuts of v . A cut is the sub-LVO corresponding to a downclosed set. Cuts are also known as prefixes. See Sect. 2.4 for an example. A cut is determined by its maximal elements: if $u \in \text{cuts}(v)$ then $u = v \upharpoonright \{d \in \mathbf{E}_v \mid \exists e \in \max(v). d \rightsquigarrow_v e\}$.
- *Linearization*: For $a_i \in \mathbf{L}$, we say that $a_1 \dots a_n$ is a *linearization* of $E \subseteq \mathbf{E}_v$ if there exists a bijection $\alpha : E \rightarrow [1, n]$ such that $\forall e \in E. \lambda_v(e) = a_{\alpha(e)}$ and $\forall d, e \in E. d \rightsquigarrow_v e$ implies $\alpha(d) \leq \alpha(e)$.

Replica-Specific Properties. In the literature on replicated data types, some properties of interest (such as “read your writes” (Tanenbaum and Steen 2007)) require the concept of “session” or a distinction between local and remote events. These can be accommodated by augmenting LVOs with a replica labelling $\rho_u \in (\mathbf{E}_u \mapsto \mathbf{R})$, which maps events to a set \mathbf{R} of *replica identifiers*.

Executions can be generated operationally as follows: Replicas receive mutator and accessor events from the local client; they also receive mutator events that are forwarded from other replicas. Each replica maintains a set of *seen* events: an event that is received is added to this set. When an event is received from the local client, the event is additionally added to the execution, with the predecessors in the visibility relation corresponding to the current *seen* set. If we wish to restrict attention to causal executions, then we require that replicas forward all the mutators in their *seen* sets, rather than individual events, and, thus, the visibility relation is transitive over mutators.

All executions that are operationally generated satisfy the additional property that \rightsquigarrow_u is per-replica total: if $\rho(d) = \rho(e)$ then either $d \rightsquigarrow_u e$ or $e \rightsquigarrow_u d$.

¹ We use the standard definitions for restriction on functions and relations. Given a function $f : E \rightarrow X$, $\mathcal{R} : E \times E$ and $D \subseteq E$, define $f \upharpoonright D = \{\langle d, f(d) \rangle \mid d \in D\}$ and $\mathcal{R} \upharpoonright D = \{\langle d_1, d_2 \rangle \mid d_1, d_2 \in D \text{ and } d_1 \mathcal{R} d_2\}$.

We do not demand per-replica totality because our results do not rely on replica-specific information.

3.2 Specifications and Stuttering Equivalence

Specifications are sets of strings, equipped with a distinguished set of mutators and a dependency relation between labels. Specifications are subject to some constraints to ensure that the mutator set and dependency relations are sensible; these are inspired by the conditions on Mazurkiewicz executions (Diekert and Rozenberg 1995). Every specification set yields a derived notion of stuttering equivalence. This leads to the definition of *observational subsequence* (\leq_{obs}).

We use standard notation for strings: Let σ and τ range over strings. Then $\sigma\tau$ denotes concatenation, σ^* denotes Kleene star, $\sigma \parallel \tau$ denotes the set of interleavings, ε denotes the empty string and σ^i denotes the i^{th} element of σ . These notations lift to sets of strings via set union.

A *specification* is a quadruple $\langle \mathbf{L}, \mathbf{M}, \#, \Sigma \rangle$ where

- \mathbf{L} is a set of *actions* (also known as *labels*),
- $\mathbf{M} \subseteq \mathbf{L}$ is a distinguished set of *mutator* actions,
- $\# \subseteq (\mathbf{L} \times \mathbf{L})$ is a symmetric and reflexive *dependency* relation, and
- $\Sigma \subseteq \mathbf{L}^*$ is a set of *valid strings*.

Let $\overline{\mathbf{M}} = \mathbf{L} \setminus \mathbf{M}$ be the sets of *non-mutators*.

A specification must satisfy the following properties:

- (a) prefix closed: $\sigma\tau \in \Sigma$ implies $\sigma \in \Sigma$
- (b) non-mutators are closed under stuttering, and commutation:
 - $\forall a \in \overline{\mathbf{M}}. \sigma a \tau \in \Sigma$ implies $\sigma a^* \tau \subseteq \Sigma$
 - $\forall a, b \in \overline{\mathbf{M}}. \{\sigma a, \sigma b\} \subseteq \Sigma$ implies $\{\sigma ab, \sigma ba\} \subseteq \Sigma$
- (c) independent actions commute:
 - $\forall a, b \in \mathbf{L}. \neg(a \# b)$ implies $(\sigma ab \tau \in \Sigma \text{ iff } \sigma ba \tau \in \Sigma)$

Property (b) ensures that non-mutators do not affect the state of the data structure. Property (c) ensures that commuting of independent actions does not affect the state of the data structure.

Recall that the SET specification takes $\mathbf{M} = \{+0, -0, +1, -1\}$, representing addition and removal of bits 0 and 1, and $\overline{\mathbf{M}} = \{\textcolor{blue}{x}0, \textcolor{blue}{\checkmark}0, \textcolor{orange}{x}1, \textcolor{red}{\checkmark}1\}$, representing membership tests returning false or true. The dependency relation is $\# = \{+0, -0, \textcolor{blue}{x}0, \textcolor{blue}{\checkmark}0\}^2 \cup \{+1, -1, \textcolor{orange}{x}1, \textcolor{red}{\checkmark}1\}^2$, where $D^2 = D \times D$.

The dependency relation for SET is an equivalence, but this need not hold generally. We will see an example in Sect. 4.

The definitions in the rest of the paper assume that we have fixed a specification $\langle \mathbf{L}, \mathbf{M}, \#, \Sigma \rangle$. In the examples of this section, we use SET.

State and Stuttering Equivalence. Specification strings σ and τ are *state equivalence* (notation $\sigma \approx \tau$) if every valid extension of σ is also a valid extension of τ , and vice versa. For example, $+0+1+0 \approx +0+1$ and $+0-0+0 \approx +0$, but $+0-0 \not\approx +0$.

In particular, state equivalent strings agree on the valid accessors that can immediately follow them: either $\checkmark 0$ or $\times 0$ and either $\checkmark 1$ or $\times 1$. Formally, we define state equivalence, $\approx \subseteq \mathbf{L}^* \times \mathbf{L}^*$, as follows².

$$(\sigma \approx \sigma') \triangleq (\sigma = \sigma') \text{ or } (\{\sigma, \sigma'\} \subseteq \Sigma \text{ and } \forall \tau \in \mathbf{L}^*. \sigma\tau \in \Sigma \text{ iff } \sigma'\tau \in \Sigma).$$

From specification property (b), we know that non-mutators do not affect the state. Thus we have that $ua \approx u$ whenever $a \in \bar{\mathbf{M}}$ and $ua \in \Sigma$. From specification property (c), we know that independent actions commute. Thus we have that $\sigma ab \approx \sigma ba$ whenever $\neg(a \# b)$ and $\{\sigma ab, \sigma ba\} \subseteq \Sigma$.

Two strings are *stuttering equivalent*³ if they only differ in operations that have no effect on the state of the data structure, as given by Σ . Adapting Brookes (1996) to our notion of state equivalence, we define stuttering equivalence, $\sim \subseteq \mathbf{L}^* \times \mathbf{L}^*$, to be the least equivalence relation generated by the following rules, where a ranges over \mathbf{L} .

$$\frac{}{\varepsilon \sim \varepsilon} \quad \frac{\sigma \sim \sigma'}{\sigma a \sim \sigma' a} \quad \frac{\sigma \approx \sigma a}{\sigma \sim \sigma a} \quad \frac{\sigma b \sim \sigma \quad \neg(a \# b)}{\sigma ab \sim \sigma a}$$

The first rule above handles the empty string. The second rule allows stuttering in any context. The third rule motivates the name stuttering equivalence, for example, allowing $+0+0 \sim +0$. The last case captures the equivalence generated by independent labels, for example, allowing $+0+1+0 \sim +0+1$ but not $+0-0+0 \sim +0-0$. Using the properties of \approx discussed above, we can conclude, for example, that $+0\checkmark 0\checkmark 0+0-0\times 0 \sim +0-0$.

Consider specification strings for a unary SET over value 0. Since stuttering equivalence allows us to remove both accessors and adjacent mutators with the same label we deduce that the *canonical representatives* of the equivalence classes induced by \sim are generated by the regular expression $(+0)^?(-0+0)^*(-0)^?$.

Observational Subsequence. Recall that ac is a *subsequence* of abc , although it is not a *prefix*. We write \leq_{seq} for subsequence and \leq_{obs} for *observational subsequence*, defined as follows.

$$\sigma_1 \cdots \sigma_n \leq_{\text{seq}} \tau_0 \sigma_1 \tau_1 \cdots \sigma_n \tau_n \quad \sigma \leq_{\text{obs}} \tau \text{ if } \exists \sigma' \sim \sigma. \exists \tau' \sim \tau. \sigma' \leq_{\text{seq}} \tau'$$

Note that observational subsequence includes both subsequence and stuttering equivalence ($\leq_{\text{obs}} \subseteq (\leq_{\text{seq}}) \cup (\sim)$).

\leq_{seq} can be understood in isolation, whereas \leq_{obs} can only be understood with respect to a given specification. In the remainder of the paper, the implied specification will be clear from context. \leq_{seq} is a partial order, whereas \leq_{obs} is only a preorder, since it is not antisymmetric.

Let σ and τ be strings over the unary SET with canonical representatives $a\sigma'$ and $b\tau'$. Then we have that $\sigma \leq_{\text{obs}} \tau$ exactly when either $a = b$ and $|\sigma'| \leq |\tau'|$

² To extend the definition to non-specification strings, we allow $\sigma \approx \sigma'$ when $\sigma = \sigma'$.

³ Readers of Brookes (1996) should note that mumbling is not relevant here, since all mutators are visible.

or $a \neq b$ and $|\sigma'| < |\tau'|$. Thus, observational subsequence order is determined by the number of alternations between the mutators.

Specification strings for the binary SET, then, are stuttering equivalent exactly when they yield the same canonical representatives when restricted to 0 and to 1. Thus, observational subsequence order is determined by the number of alternations between the mutators, when restricted to each dependent subsequence. (The final rule in the definition of stuttering, which allows stuttering across independent labels, is crucial to establishing this canonical form.)

3.3 Eventual Consistency

Eventual consistency is defined using the *cuts* of an execution and the *observational subsequence order* of the specification. As noted in Sects. 2.2 and 2.4, it is important that we not consider all cuts. Thus, before we define SEC, we must define *dependent cuts*.

The *dependent restriction* of an execution is defined: $v \downarrow \# = \langle E_v, \lambda_v, \overset{\#}{\rightsquigarrow}_v \rangle$, where $d \overset{\#}{\rightsquigarrow}_v e$ when $\lambda_v(d) \# \lambda_v(e)$ and $d \rightsquigarrow_v e$. See Sect. 2.2 for an example of dependent restriction.

The *dependent cuts* of v are cuts of the dependent restriction. As discussed in Sect. 2.4, we only consider pointed cuts (with a single maximal element) for non-mutators. See Sect. 2.4 for an example.

$$\text{cuts}_{\#}(v) = \{u \in \text{cuts}(v \downarrow \#) \mid \forall e \in E_u. \text{ if } \lambda_u(e) \in \overline{\mathbf{M}} \text{ then } \max(u) = \{e\}\}$$

An execution v is *Eventually Consistent* (SEC) for specification $\langle \mathbf{L}, \mathbf{M}, \#, \Sigma \rangle$ iff there exists a function $\tau : \text{cuts}_{\#}(v) \rightarrow \Sigma$ that satisfies the following.

Linearization: $\forall p \in \text{cuts}_{\#}(v). p$ linearizes to $\tau(p)$, and

Monotonicity: $\forall p, q \in \text{cuts}_{\#}(v). p \subseteq q$ implies $\tau(p) \leq_{\text{obs}} \tau(q)$.

A data structure implementation is SEC if all of its executions are SEC.

In Sect. 2, we gave several examples that are SEC. See Sects. 2.4 and 2.5 for examples where τ is given explicitly. Section 2.4 also includes an example that is not SEC.

The concerns raised in Sect. 2 are reflected in the definition.

- Non-mutators are ignored by the dependent restriction of other non-mutators. As discussed in Sect. 2.1, this relaxation is similar that of update-serializability (Hansdah and Patnaik 1986; Garcia-Molina and Wiederhold 1982).
- Independent events are ignored by the dependent restriction of an event. As discussed in Sect. 2.2, this relaxation is similar to preserved program order in relaxed memory models (Higham and Kawash 2000; Alglave 2012).
- As discussed in Sect. 2.3, punning is allowed: each cut p is linearized separately to a specification string $\tau(p)$.
- As discussed in Sect. 2.4, we constrain the power puns by considering cuts of the distributed system (Chandy and Lamport 1985).

- Monotonicity ensures that the system evolves in a sensible way: new order may be introduced, but old order cannot be forgotten. As discussed in Sect. 2.5, the preserved order is captured in the observational subsequence relation, which allows stuttering (Brookes 1996).

3.4 Properties of Eventual Consistency

We discuss some basic properties of SEC. For further analysis, see Sect. 5.

An important property of CRDTs is *prefix closure*: If an execution is valid, then every prefix of the execution should also be valid. Prefix closure follows immediately from the definition, since whenever u is a prefix of v we have that $\text{cuts}_\#(u) \subseteq \text{cuts}_\#(v)$.

Prefix closure looks back in time. It is also possible to look forward: A system satisfies *eventual delivery* if every valid execution can be extended to a valid execution with a maximal element that sees every mutator. If one assumes that every specification string can be extended to a longer specification string by adding non-mutators, then eventual delivery is immediate.

The properties PSTS, PSM and PPE are discussed in the introduction. An SEC implementation must satisfy PPE since every dependent set of mutators is linearized: SEC enforces the stronger property that there are no new intermediate states, even when executing all mutators in parallel. For causal systems, where \rightsquigarrow_u is transitive, PSTS and PSM follow by observing that if there is a total order on the mutators of u then any linearization of u is a specification string.

Burckhardt (2014, Sect. 5) provides a taxonomy of correctness criteria for replicated data types. Our definition implies NOCIRCULARCAUSALITY and CAUSALARBITRATION, but does not imply either CONSISTENTPREFIX or CAUSALVISIBILITY. For LPOS, which model causal systems, our definition implies CAUSALVISIBILITY. READMYWRITES and MONOTONICREADS require a distinction between local and remote events. If one assumes the replica-specific constraints given in Sect. 3.1, then our definition satisfies these properties; without them, our definition is too abstract.

3.5 Correctness of the Add-Wins Set

The add-wins set is defined to answer $\checkmark \mathbf{k}$ for a cut u exactly when

$$\exists d \in u. \lambda_u(d) = +\mathbf{k} \ \wedge \ (\nexists e \in u. \lambda_u(e) = -\mathbf{k} \ \wedge \ d \rightsquigarrow_u e).$$

It answers \mathbf{xk} otherwise. The add-wins set is called the “observed-remove” set.

We show that any LPO that meets this specification is SEC with respect to SET. We restrict attention to LPOS since causal delivery is assumed for the add-wins set in (Shapiro et al. 2011a).

For SET, the dependency relation is an equivalence. For an equivalence relation R , let $\mathbf{L}/R \subseteq 2^{\mathbf{L}}$ denote the set of (disjoint) equivalence classes for R . For SET, $\mathbf{L}/\# = \{\{+0, -0, \mathbf{x}0, \checkmark0\}, \{+1, -1, \mathbf{x}1, \checkmark1\}\}$. When dependency is an

equivalence, then *every* interleaving of independent actions is valid if *any* interleaving is valid. Formally, we have the following, where \parallel denotes interleaving.

$$\forall D \in (\mathbf{L}/\#). \forall \sigma \in D^*. \forall \tau \in (\mathbf{L} \setminus D)^*. (\sigma \parallel \tau) \cap \Sigma \neq \emptyset \text{ implies } (\sigma \parallel \tau) \subseteq \Sigma$$

Using the forthcoming composition result (Theorem 2), it suffices for us to address the case when u only involves operations on a single element, say 0. For any such LVO u , we choose a linearization $\tau(u) \in (-0|+0)^*$ that has a maximum number of alternations between -0 and $+0$. If there is a linearization that begins with -0 , then we choose one of these. Below, we summarize some of the key properties of such a linearization.

- $\tau(u)$ ends with $+0$ iff there is an $+0$ that is not followed by any -0 in u .
- For any LPO $v \subseteq u$, $\tau(v)$ has at most as many alternations as $\tau(u)$.

The first property above ensures that the accessors are validated correctly, *i.e.*, 0 is deemed to be present iff there is an $+0$ that is not followed by any -0 .

We are left with proving monotonicity, *i.e.*, if $u \subseteq v$, then $\tau(u) \leq_{\text{obs}} \tau(v)$. Consider $\tau(u) = a\sigma$ and $\tau(v) = b\rho$.

- If $b = a$, the second property above ensures that $\tau(u) \leq_{\text{obs}} \tau(v)$.
- In the case that $b \neq a$, we deduce by construction that $b = -0$ and $a = +0$. In this case, ρ starts with $+0$ and has at least as many alternations as $\tau(u)$. So, we deduce that $\tau(u) \leq_{\text{obs}} \rho$. The required result follows since $\rho \leq_{\text{obs}} \tau(v)$.

4 A Collaborative Text Editing Protocol

In this section we consider a variant of the collaborative text editing protocol defined by Attiya et al. (2016). After stating the sequential specification, TEXT, we sketch a correctness proof with respect to our definition of eventual consistency. This example is interesting formally: the dependency relation is not an equivalence, and therefore the dependent projection does not preserve transitivity. The generality of intransitive LVOs is necessary to understand TEXT, even assuming a causal implementation.

Specification. Let a, b range over *nodes*, which contain some text, a unique identifier, and perhaps other information. Labels have the following forms:

- Mutator $!a$ initializes the text to node a .
- Mutator $+a<b$ adds node a immediately before node b .
- Mutator $+a>b$ adds node a immediately after node b .
- Mutator $-b$ removes node b .
- Non-mutator query $?b_1 \cdots b_n$ returns the current state of the document.

We demonstrate the correct answers to queries by example. Initially, the document is empty, whereas after initialization, the document contains a single node; thus the specification contains strings such as “ $? \varepsilon !c ?c$ ”, where ε represents the empty document. Nodes can be added either before or after other nodes; thus

“!c +b<c +d>c” results in the document ?bcd. Nodes are always added adjacent to the target; thus, order matters in “!c +e>c +d>c” which results in ?cde rather than ?ced. Removal does what one expects; thus “!c +e>c +d>c -c” results in ?de.

Attiya et al. (2016) define the interface for TEXT using integer indices as targets, rather than nodes. Using the unique correspondence between the nodes and it indices (since node are unique), one can easily adapt an implementation that satisfies our specification to their interface.

We say that node a is a *added* in the actions !a, +a<b and +a>b. Node b is a *target* in +a<b and +a>b. In addition to correctly answering queries, specifications must satisfy the following constraints:

- Initialization may occur at most once,
- each node may be added at most once,
- a node may be removed only after it is added, and
- a node may be used as a target only if it has been added and not removed.

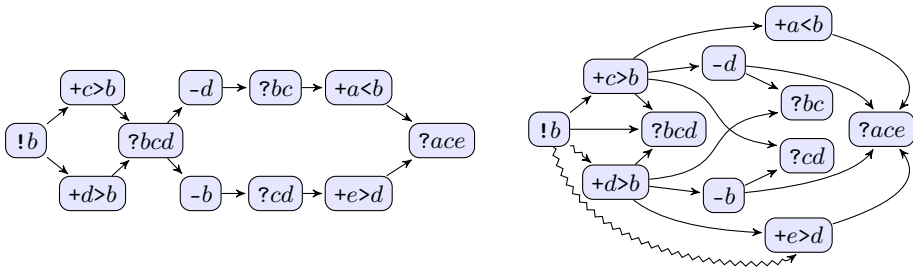
These constraints forbid adding to a target that has been removed; thus “!c +d>c -c” is a valid string, but “!c -c +d>c” is not. It also follows that initialization must precede any other mutators.

Because add operations use unique identifiers, punning and stuttering play little role in this example. In order to show the implementation correct, we need only choose an appropriate notion of dependency. As we will see, it is necessary that removes be independent of adds with disjoint label sets, but otherwise all actions may be dependent. Let $\mathbf{L}_{!+?}$ be the set of add and query labels, and let nodes return the set of nodes that appear in a label. Then we define dependency as follows.

$$\ell \# k \text{ iff } \{\ell, k\} \subseteq \mathbf{L}_{!+?} \text{ or } \text{nodes}(\ell) \cap \text{nodes}(k) \neq \emptyset$$

Implementation. We consider executions that satisfy the same four conditions above imposed on specifications. We refer the reader to the algorithm of Attiya et al. (2016) that provides timestamps for insertions that are monotone with respect to causality.

As an example, Attiya et al. (2016) allow the execution given on the left below. In this case, the dependent restriction is an intransitive LVO, even though the underlying execution is an LPO: in particular, !b does not precede -d in the dependent restriction. We give the order considered by dependent cuts on the right—this is a restriction of the dependent restriction: since we only consider pointed accessor cuts, we can safely ignore order out of non-mutators.



This execution is not linearizable, but it is SEC, choosing witnesses to be subsequences of the mutator string “ $!b +d>b +c>b +a<b +e>d -b -d$ ”. Here, the document is initialized to b , then c and d are added after b , resulting in $?bcd$. The order of c and d is determined by their timestamps. Afterwards, the top replica removes d and adds a ; the bottom replica removes b and adds e , resulting in the final state $?ace$. In the right execution, the removal of order out of the non-mutators shows the “update serializability” effect; the removal of order between $-b$ and $+e>d$ (and between $-d$ and $+a<b$) shows the “preserved program order” effect.

Correctness. Given an execution, we can find a specification string s_1s_2 that linearizes the mutators in the dependent restriction of the execution such that s_1 contains only adds and s_2 contains only removes. Such a specification string exists because by the conditions on executions, deletes do not have any outgoing edges to other mutators in the dependent restriction; so, they can be moved to the end in the matching specification string. In order to find s_1 that linearizes the add events, any linearization that respects causality and timestamps (yielded by the algorithm of Attiya et al. (2016)) suffices for our purposes. The conditions required by SEC follow immediately.

5 Compositional Reasoning

The aim of this section is to establish compositional methods to reason about replicated data structures. We do so using *Labelled Transition Systems* (LTSs), where the transitions are labelled by dependent cuts. We show how to derive an LTS from an execution, $\text{lts}(u)$. We also define an LTS for the *most general* CRDT that validates a specification, $\text{lts}(\Sigma)$. We show that u is SEC for Σ exactly when $\text{lts}(u)$ is a refinement of $\text{lts}(\Sigma)$. We use this alternative characterization to establish composition and abstraction results.

LTSs. An LTS is a triple consisting of a set of states, an initial state and a labelled transition function between states. We first define the LTSs for executions and specifications, then provide examples and discussion.

For both executions and specifications, the labels of the LTS are dependent cuts: for executions, these are dependent cuts of the execution itself; for specifications, they are drawn from the set $\mathcal{L}_\# = \bigcup_{v \in \mathcal{L}} \text{cuts}_\#(v)$ of all possible dependent cuts. We compare LTS labels up to isomorphism, rather than identity. Thus

it is safe to think of LTS labels as (potentially intransitive) pomsets (Plotkin and Pratt 1997).

The states of the LTS are different for the execution and specification. For executions, the states are cuts of the execution u itself, $\text{cuts}(u)$; these are general cuts, not just dependent cuts. For specifications, the states are the stuttering equivalence classes of strings allowed by the specification, Σ/\sim .

There is an isomorphism between strings and total orders. We make use of this in the definition, treating strings as totally-ordered LVOS.

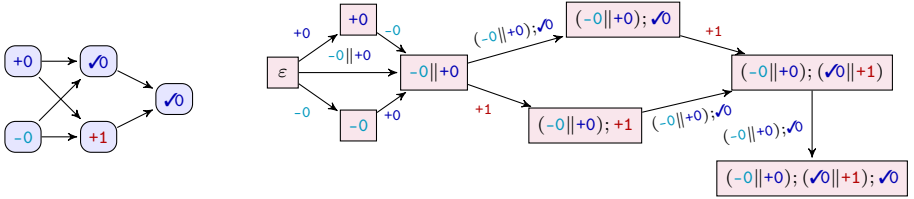
Define $\text{lts}(u) = \langle \text{cuts}(u), \emptyset, \mapsto_i \rangle$, where $p \mapsto_i^v q$ if $v \in \text{cuts}_\#(q)$ and

$$\begin{array}{lll} p \subseteq q & E_{\max(v)} \cup E_p = E_q & \overline{\max}(v) \subseteq p \\ v \subseteq q & E_{\max(v)} \cap E_p = \emptyset & E_{\max(v)} \subseteq E_{\max(q)} \end{array}$$

Define $\text{lts}(\Sigma) = \langle \Sigma/\sim, \varepsilon, \mapsto_s \rangle$, where $[\sigma] \mapsto_s^v [\rho]$ if $v \in \mathcal{L}_\#$ and

$$\begin{array}{lll} \sigma \subseteq \rho & E_{\max(v)} \cup E_\sigma = E_\rho & \overline{\max}(v) \subseteq \sigma \\ v \subseteq \rho & E_{\max(v)} \cap E_\sigma = \emptyset & \end{array}$$

We explain the definitions using examples from SET, first for executions, then for specifications. Consider the execution on the left below. The derived LTS is given on the right.



The states of the LTS are cuts of the execution. The labels on transitions are *dependent* cuts. The requirements for execution transitions relate the source p , target q and label v . The leftmost requirements state that the target state must extend both the source and the label; thus the target state must be a combination of events and order from source and label. The middle requirements state that the maximal elements of the label must be new in the target; only the maximal elements of the label are added when moving from source to target. The upper right requirement states that the non-maximal order of the label must be respected by the source; thus the causal history reported by the label cannot contradict the causal history of the source. The lower right requirement ensures that maximal elements of the label are also maximal in the target. The restriction to dependent cuts explains the labels on transitions $(-0||+0) \xrightarrow{+1} (-0||+0); +1$ and $(-0||+0); (\check{0}||+1); \check{0} \xrightarrow{(-0||+0); \check{0}} (-0||+0); (\check{0}||+1)$. By definition, there is a self-transition labelled with the empty LVO at every state; we elide these transitions in drawings.

The specification LTS for SET is infinite, of course. To illustrate, below we give two sub-LTSS with limitations on mutators. On the left, we only allow $+0$ and $+1$. On the right, we only allow $+0$ and -0 and only consider the case in which

restriction attention to mutators, $p \downarrow \# \downarrow \mathbf{M}$. The required refinement maps p to the equivalence class of the linearization of $p \downarrow \# \downarrow \mathbf{M}$ chosen by τ : $f(p) \triangleq [\tau(p \downarrow \# \downarrow \mathbf{M})]$. We abuse notation below by identifying each equivalence class with a canonical element of the class.

We show that $p \xrightarrow{v}_i q$ implies $f(p) \leq_{\text{obs}} f(q)$. Since $p \subseteq q$, we deduce that $p \downarrow \# \downarrow \mathbf{M} \subseteq q \downarrow \# \downarrow \mathbf{M}$ and by monotonicity, $f(p) = \tau(p \downarrow \# \downarrow \mathbf{M}) \leq_{\text{obs}} \tau(q \downarrow \# \downarrow \mathbf{M}) = f(q)$.

We show that $p \xrightarrow{v}_i q$ implies $\tau(v) \leq_{\text{obs}} f(q)$. Suppose v only contains mutators. Since $v \subseteq q$, we deduce that $v \subseteq q \downarrow \# \downarrow \mathbf{M}$ and by monotonicity, $\tau(v) \leq_{\text{obs}} \tau(q \downarrow \# \downarrow \mathbf{M}) = f(q)$. On the other hand, suppose v contains the non-mutator a . Let $A = \mathbf{M} \cup \{a\}$. Since $v \subseteq q$, we deduce that $v \downarrow \mathbf{M} \subseteq q \downarrow \# \downarrow \mathbf{M}$. By monotonicity, $\tau(v \downarrow \mathbf{M}) \leq_{\text{obs}} \tau(q \downarrow \mathbf{M})$. Since $\tau(q \downarrow \mathbf{M}) = \tau(q \downarrow \# \downarrow \mathbf{M})$, we have $\tau(v \downarrow \mathbf{M}) \leq_{\text{obs}} \tau(q \downarrow \# \downarrow \mathbf{M}) = f(q)$, as required.

Thus $f(p) \xrightarrow{v}_s f(q)$, completing this direction of the proof.

For the reverse direction, we are given a refinement $f : \text{cuts}(u) \rightarrow \Sigma/\sim$. For any $p \in \text{cuts}_{\#}(u)$, define $\tau(p)$ to be a string in the equivalence class $f(p)$ that includes any non-mutator found in p .

We first prove that $\tau(p)$ is a linearization of p . A simple inductive proof demonstrates that for any $p \in \text{cuts}_{\#}(u)$, there is a transition sequence of the form $\emptyset \xrightarrow{*}_i \xrightarrow{p}_i p$. Thus, we deduce from the label on the final transition into p that the $\tau(p)$ related to p is a linearization of p .

We now establish monotonicity. A simple inductive proof shows that for any $p, q \in \text{cuts}(u)$, $p \subseteq q$ implies $p \xrightarrow{*}_i q$. Thus $\tau(p) \leq_{\text{obs}} \tau(q)$, by the properties of f and the definition of τ .

Composition. Given two *non-interacting* data structures whose replicated implementations satisfy their sequential specifications, the implementation that combines them satisfies the interleaving of their specifications. We formalize this as a composition theorem in the style of Herlihy and Wing (1990).

Given an execution u and $L \subseteq \mathbf{L}$, write $u \downarrow L$ for the execution that results by restricting u to events with labels in L : $u \downarrow L = u \downarrow \{e \in E_u \mid \lambda_u(e) \in L\}$. This notation lifts to sets in the standard way: $U \downarrow L = \bigcup_{u \in U} \{u \downarrow L\}$. Write $u \models_{\text{sec}} \Sigma$ to indicate that u is SEC for Σ .

Theorem 2 (Composition). *Let L_1 and L_2 be mutually independent subsets of \mathbf{L} . For $i \in \{1, 2\}$, let Σ_i be a specification with labels chosen from L_i , such that $\Sigma_1 \parallel \Sigma_2$ is also a specification. If $(U \downarrow L_1) \models_{\text{sec}} \Sigma_1$ and $(U \downarrow L_2) \models_{\text{sec}} \Sigma_2$ then $U \models_{\text{sec}} (\Sigma_1 \parallel \Sigma_2)$ (equivalently $\text{Its}(\Sigma_1 \parallel \Sigma_2) \approx \text{Its}(\Sigma_1) \parallel \text{Its}(\Sigma_2)$).*

The proof is immediate. Since L_1 and L_2 are mutually independent, any interleaving of the labels will satisfy the definition.

Abstraction. We describe a process algebra with parallel composition and restriction and establish congruence results. We ignore syntactic details and work directly with LTSS. Replica identities do not play a role in the definition; thus, we permit implicit mobility of the client amongst replicas with the only constraint being that the replica has at least as much history on the current item

of interaction as the client. This constraint is enforced by the synchronization of the labels, defined below. While the definition includes the case where the client itself is replicated, it does not provide for out-of-band interaction between the clients at different replicas: All interaction is assumed to happen through the data structure.

The relation \parallel is defined between LTSS so that $P \parallel Q$ describes the system that results when client P interacts with data structure Q . For LTSS P and Q , define \mapsto_{\times} inductively, as follows, where \emptyset represents the empty LVO.

$$\frac{q \xrightarrow{v}_Q q'}{\langle p, q \rangle \xrightarrow{v}_{\times} \langle p, q' \rangle} \quad \frac{p \xrightarrow{v}_P p' \quad q \xrightarrow{w}_Q q'}{\langle p, q \rangle \xrightarrow{\emptyset}_{\times} \langle p', q' \rangle} \quad \exists v' =_{\text{iso}} v. v' \subseteq w \text{ and } \max(v') = \max(w)$$

Let $S_{\times} = \{\langle p, q \rangle \mid \exists \langle p', q' \rangle. \langle p, q \rangle \mapsto^*_{\times} \langle p', q' \rangle \text{ and } \nexists v, p''. p' \xrightarrow{v}_P p''\}$

$$P \parallel Q = \begin{cases} \{\langle S_{\times}, \langle p_0, q_0 \rangle, \mapsto_{\times} \rangle\} & \text{if } S_{\times} \text{ is non-empty} \\ \emptyset & \text{otherwise} \end{cases}$$

The \parallel operator is asymmetric between the client and data structure in two ways. First, note that every action of the client must be matched by the data structure. The condition of client quiescence in the definition of S_{\times} , that all of the actions of the client P must be matched by Q ; otherwise $P \parallel Q = \emptyset$. However, the first rule for \mapsto_{\times} explicitly permits actions of the data structure that may not be matched by the client. This asymmetry permits the composition of the data structure with multiple clients to be described incrementally, one client at a time. Thus, we expect that $(P_1 \mid P_2) \parallel Q \approx P_1 \parallel (P_2 \parallel Q)$.

Second, note that right rule for \mapsto_{\times} interaction permits the data structure Q to introduce order not found in the clients. This is clearly necessary to ensure that that the composition of client $\textcolor{blue}{\checkmark}0 \mid +0$ with the SET data structure is nonempty. In this case, the client has no order between $+0$ and $\textcolor{blue}{\checkmark}0$ whereas the data structure orders $\textcolor{blue}{\checkmark}0$ after $+0$. In this paper, we do not permit the client to introduce order that is not seen in the data structure. For a discussion of this issue, see (Jagadeesan and Riely 2015).

We can also define restriction for some set $A \subseteq \mathbf{L}$ of labels, a lá CCS. $P \setminus A = \langle S_P, p_0, \{\langle p, v, q \rangle \mid \langle p, v, q \rangle \in (\mapsto_P) \text{ and } \text{labels}(v) \cap A = \emptyset\} \rangle$. The definitions lift to sets: $\mathcal{P} \parallel \mathcal{Q} = \bigcup_{P \in \mathcal{P}, Q \in \mathcal{Q}} P \parallel Q$ and $\mathcal{P} \setminus A = \{(P \setminus A) \mid P \in \mathcal{P}\}$.

Lemma 3. *If $\mathcal{P} \sqsubseteq \mathcal{P}'$ and $\mathcal{Q} \sqsubseteq \mathcal{Q}'$ then $\mathcal{P} \parallel \mathcal{Q} \sqsubseteq \mathcal{P}' \parallel \mathcal{Q}'$ and $\mathcal{P} \setminus A \sqsubseteq \mathcal{P}' \setminus A$. \square*

It suffices to show that: $P \sqsubseteq \text{Its}(u)$ implies $\mathcal{P} \parallel \text{Its}(u) \sqsubseteq \mathcal{P} \parallel \text{Its}(\Sigma)$. The proof proceeds in the traditional style of such proofs in process algebra. We illustrate by sketching the case for client parallel composition. Let f be the witness for $P \sqsubseteq \text{Its}(u)$. The proof proceeds by constructing a “product” refinement \mathcal{S} relation of the identity on the states of P with f , i.e.: $f(q) = q'$ implies $\langle p, q \rangle \mathcal{S} \langle p, q' \rangle$.

Thus, an SEC implementation can be replaced by the specification.

Theorem 4 (Abstraction). *If u is SEC for Σ , then $\mathcal{P} \parallel \text{Its}(u) \sqsubseteq \mathcal{P} \parallel \text{Its}(\Sigma)$.*

6 A Replicated Graph Algorithm

We describe a graph implemented with sets for vertices and edges, as specified by Shapiro et al. (2011a). The graph maintains the invariant that the vertices of an edge are also part of the graph. Thus, an edge may be added only if the corresponding vertices exist; conversely, a vertex may be removed only if it supports no edge. In the case of a concurrent addition of an edge with the deletion of either of its vertices, the deletion takes precedence.

The vertices v, w, \dots are drawn from some universe \mathcal{U} . An edge e, e', \dots is a pair of vertices. Let $\text{vert}(e) = \{v, w\}$ be the vertices of edge $e = (v, w)$. The vocabulary of the set specification includes mutators for the addition and removal of vertices and edges and non-mutators for membership tests.

$$\begin{aligned}\mathbf{M} &= \{+v, -v, +(v, w), -(v, w) \mid v, w \in \mathcal{U}\} \\ \overline{\mathbf{M}} &= \{\checkmark v, \times v, \checkmark(v, w), \times(v, w) \mid v, w \in \mathcal{U}\} \\ \# &= \{(e, v), (v, e) \mid v \in \text{vert}(e)\} \cup \{(e, e') \mid \text{vert}(e) \cap \text{vert}(e') \neq \emptyset\}\end{aligned}$$

Valid graph specification strings answer queries like sets. In addition, we require the following.

- Vertices and edges added at most once: Each add label is unique.
- Removal of a vertex or edge is preceded by a corresponding add.
- Vertices are added before they are mentioned in any edges: If $\sigma^j = +(v, w)$, or $\sigma^j = -(v, w)$ there exists $i, i' < j$ such that: $\sigma^i = +v$, $\sigma^{i'} = +w$.
- Vertices are removed only after they are mentioned in edges: If $\sigma^j = +(v, w)$, or $\sigma^j = -(v, w)$, then for all $i < j$: $\sigma^i \neq -v$ and $\sigma^i \neq -w$.

Graph Implementation. We rewrite the graph program of Shapiro et al. (2011a) in a more abstract form. Our distributed graph implementation is written as a client of two replicate set: for vertices (**V**) and for edges (**E**). The implementation uses USETs, which require that an element be added at most once and that each remove causally follow the corresponding add. Here we show the graph implementation for various methods as client code that runs at each replica. At each replica, the code accesses its local copy of the USETs. All the message passing needed to propagate the updates is handled by the USET implementations of the sets **V**, **E**. For several methods, we list preconditions, which prescribe the natural assumptions that need to be satisfied when these client methods are invoked. For example, an edge operation requires the presence of the vertices at the current replica.

addVertex(v)	removeVertex(v)	bool ?(v)
Pre: fresh(v)	Pre: V.?(v)	return V.?(v)
V.add(v)	V.remove(v)	
addEdge(v,w)	removeEdge(v,w)	bool ?(v,w)
Pre: V.?(v), V?(w)	Pre: V.?(v), V?(w)	if V.?(v)
Pre: fresh((v,w))	Pre: E.?(v,w)	then return E.?(v,w)
E.add((v,w))	E.remove((v,w))	else return false

We assume a causal transition system (as needed in Shapiro et al. (2011a)).

Correctness Using the Set Specification. We first show the correctness of the graph algorithm, using the SET specification for the vertex and edge sets. We then apply the abstraction and composition theorems to show the correctness of the algorithm using a set implementation.

Let u be a LVO generated in an execution of the graph implementation. The preconditions ensure that u has the following properties:

- (a) For any v , $+v$ is never ordered after $-v$, and likewise for e .
- (b) $-(v, w)$ or $+(v, w)$ is never ordered after $-v$ or $-w$.
- (c) $-(v, w)$ or $+(v, w)$ is always ordered after some $+v$ and $+w$.

Define σ_1 , σ_2 and σ_3 as follows.

- All elements of σ_1 are of the form $+v$. σ_1 exists by (c) above.
- All elements of σ_3 are of the form $-v$. σ_3 exists by (b) above.
- For each edge (v, w) that is accessed in u , let $\sigma_{(v, w)}$ be any interleaving of the events involving (v, w) in u such that no $+(v, w)$ occurs after any $-(v, w)$ in $\sigma_{(v, w)}$. $\sigma_{(v, w)}$ exists by (a) above. σ_2 is any interleaving of all the $\sigma_{(v, w)}$.

Then u is SEC with witness $\sigma_u = \sigma_1\sigma_2\sigma_3$.

Full Correctness of the Implementation. We now turn to proving the correctness of the algorithm when the two sets are replaced by their implementations.

Consider two (distributed implementations of) separate and independent sets for vertices and edges, *i.e.* $\mathbf{L}_{\Sigma_1} \cap \mathbf{L}_{\Sigma_2} = \emptyset$. Suppose we have two implementations, each of which is correct individually: $\text{Its}(U_i) \sqsubseteq \text{Its}(\Sigma_i)$. By composition, we have that they are correct when composed together: $U_1 \parallel U_2 \sqsubseteq \Sigma_1 \parallel \Sigma_2$. Let \mathcal{P} be the graph implementation, which is a client of the two sets. By abstraction, we know that $\mathcal{P} \parallel (\Sigma_1 \parallel \Sigma_2) \sqsubseteq T$ implies $\mathcal{P} \parallel (U_1 \parallel U_2) \sqsubseteq T$. By congruence, we deduce:

$$(\mathcal{P} \parallel (\Sigma_1 \parallel \Sigma_2)) \setminus (\mathbf{L}_{\Sigma_1} \cup \mathbf{L}_{\Sigma_2}) \sqsubseteq T \text{ implies } (\mathcal{P} \parallel (U_1 \parallel U_2)) \setminus (\mathbf{L}_{\Sigma_1} \cup \mathbf{L}_{\Sigma_2}) \sqsubseteq T.$$

Thus, in order to validate the full graph implementation, it is sufficient to establish the correctness of the graph client when interacting with the *specification* of the two independent SETs for edges and vertices, which we have already done in the previous treatment of abstract correctness.

7 Conclusions

We have provided a definition of *strong eventual consistency* that captures *validity* with respect to a *sequential specification*. Our definition reflects an attempt to resolve the tension between expressivity (cover the extant examples in the literature) and facilitating reasoning (by retaining a direct relationship with the sequential specification). The notion of *concurrent specification* developed by Burckhardt et al. (2014) has been used to prove the validity of several replicated

data structure implementations. In future work, we would like to discover sufficient conditions relating concurrent and sequential specifications such that any implementation that is correct under the concurrent specification (as defined by Burckhardt et al. (2014)) will also be correct under the sequential counterpart (as defined here).

Acknowledgements. This paper has been greatly improved by the comments of the anonymous reviewers.

This material is based upon work supported by the National Science Foundation under Grant No. 1617175. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

- Alglave, J.: A formal hierarchy of weak memory models. *Formal Methods Syst. Des.* **41**(2), 178–210 (2012)
- Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* **36**(2), 7:1–7:74 (2014)
- Attiya, H., Burckhardt, S., Gotsman, A., Morrison, A., Yang, H., Zawirski, M.: Specification and complexity of collaborative text editing. In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA*, pp. 259–268, 25–28 July 2016
- Bieniusa, A., Zawirski, M., Preguiça, N., Shapiro, M., Baquero, C., Balegas, V., Duarte, S.: Brief announcement: semantics of eventually consistent replicated sets. In: Aguilera, M.K. (ed.) *DISC 2012. LNCS*, vol. 7611, pp. 441–442. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33651-5_48
- Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: *POPL 2014*, pp. 285–296 (2014)
- Brookes, S.D.: Full abstraction for a shared-variable parallel language. *Inf. Comput.* **127**(2), 145–163 (1996)
- Budhiraja, N., Marzullo, K., Schneider, F.B., Toueg, S.: The primary-backup approach. In: Mullender, S. (ed.) *Distributed Systems*, 2nd edn., pp. 199–216 (1993)
- Burckhardt, S.: Principles of eventual consistency. *Found. Trends Program. Lang.* **1**(1–2), 1–150 (2014). ISSN 2325-1107
- Burckhardt, S., Leijen, D., Fähndrich, M., Sagiv, M.: Eventually consistent transactions. In: Seidl, H. (ed.) *ESOP 2012. LNCS*, vol. 7211, pp. 67–86. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_4
- Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: *POPL 2014*, pp. 271–284 (2014)
- Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.* **3**(1), 63–75 (1985)
- DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Voshall, P., Vogels, W.: Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.* **41**(6), 205–220 (2007)
- Diekert, V., Rozenberg, G. (eds.): *The Book of Traces*. World Scientific Publishing Co., Inc., River Edge (1995). ISBN 9810220588

- Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. *ACM SIGMOD Rec.* **18**(2), 399–407 (1989)
- Garcia-Molina, H., Wiederhold, G.: Read-only transactions in a distributed database. *ACM Trans. Database Syst.* **7**(2), 209–234 (1982)
- Gilbert, S., Lynch, N.: Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* **33**(2), 51–59 (2002)
- Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: ‘Cause i’m strong enough: reasoning about consistency choices in distributed systems. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT POPL*, pp. 371–384 (2016)
- Haas, A., Henzinger, T.A., Holzer, A., Kirsch, C.M., Lippautz, M., Payer, H., Sezgin, A., Sokolova, A., Veith, H.: Local linearizability. *CoRR*, abs/1502.07118 (2015)
- Hansdah, R.C., Patnaik, L.M.: Update serializability in locking. In: Ausiello, G., Atzeni, P. (eds.) *ICDT 1986. LNCS*, vol. 243, pp. 171–185. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-17187-8_36
- Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS* **12**(3), 463–492 (1990)
- Higham, L., Kawash, J.: Memory consistency and process coordination for SPARC multiprocessors. In: Valero, M., Prasanna, V.K., Vajapeyam, S. (eds.) *HiPC 2000. LNCS*, vol. 1970, pp. 355–366. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44467-X_32
- Hoare, C.A.R.: Proof of correctness of data representations. *Acta Informatica* **1**(4), 271–281 (1972)
- Jagadeesan, R., Riely, J.: From sequential specifications to eventual consistency. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) *ICALP 2015, Part II. LNCS*, vol. 9135, pp. 247–259. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_20
- Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978)
- Lamport, L.: Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.* **5**(2), 190–222 (1983)
- Lynch, N., Vaandrager, F.: Forward and backward simulations. *Inf. Comput.* **121**(2), 214–233 (1995)
- Perrin, M., Mostéfaoui, A., Jard, C.: Update consistency for wait-free concurrent objects. In: *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India*, pp. 219–228, 25–29 May 2015
- Plotkin, G., Pratt, V.: Teams can see pomsets. In: *Workshop on Partial Order Methods in Verification. DIMACS Series*, vol. 29, pp. 117–128. AMS (1997)
- Saito, Y., Shapiro, M.: Optimistic replication. *ACM Comput. Surv.* **37**(1), 42–81 (2005)
- Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. *TR 7506*, Inria (2011a)
- Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pp. 386–400 (2011b)
- Shavit, N.: Data structures in the multicore age. *Commun. ACM* **54**(3), 76–84 (2011)
- Tanenbaum, A., Steen, M.V.: *Distributed systems*. Pearson Prentice Hall, Upper Saddle River, NJ (2007)
- Viotti, P., Vukolic, M.: Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.* **49**(1), 19 (2016)
- Vogels, W.: Eventually consistent. *Commun. ACM* **52**(1), 40–44 (2009)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Compiler Verification



A Verified Compiler from Isabelle/HOL to CakeML

Lars Hupe^(✉)  and Tobias Nipkow 

Technische Universität München, Munich, Germany

lars.hupe@tum.de, nipkow@in.tum.de

Abstract. Many theorem provers can generate functional programs from definitions or proofs. However, this code generation needs to be trusted. Except for the HOL4 system, which has a proof producing code generator for a subset of ML. We go one step further and provide a verified compiler from Isabelle/HOL to CakeML. More precisely we combine a simple proof producing translation of recursion equations in Isabelle/HOL into a deeply embedded term language with a fully verified compilation chain to the target language CakeML.

Keywords: Isabelle · CakeML · Compiler
Higher-order term rewriting

1 Introduction

Many theorem provers have the ability to generate executable code in some (typically functional) programming language from definitions, lemmas and proofs (e.g. [6, 8, 9, 12, 16, 27, 37]). This makes code generation part of the trusted kernel of the system. Myreen and Owens [30] closed this gap for the HOL4 system: they have implemented a tool that translates from HOL4 into *CakeML*, a subset of SML, and proves a theorem stating that a result produced by the CakeML code is correct w.r.t. the HOL functions. They also have a verified implementation of CakeML [24, 40]. We go one step further and provide a once-and-for-all verified compiler from (deeply embedded) function definitions in Isabelle/HOL [32, 33] into CakeML proving partial correctness of the generated CakeML code w.r.t. the original functions. This is like the step from dynamic to static type checking. It also means that preconditions on the input to the compiler are explicitly given in the correctness theorem rather than implicitly by a failing translation. To the best of our knowledge this is the first verified (as opposed to certifying) compiler from function definitions in a logic into a programming language.

Our compiler is composed of multiple phases and in principle applicable to other languages than Isabelle/HOL or even HOL:

- We erase types right away. Hence the type system of the source language is irrelevant.
- We merely assume that the source language has a semantics based on equational logic.

The compiler operates in three stages:

1. The preprocessing phase eliminates features that are not supported by our compiler. Most importantly, *dictionary construction* eliminates occurrences of type classes in HOL terms. It introduces dictionary datatypes and new constants and proves the equivalence of old and new constants (Sect. 7).
2. The *deep embedding* lifts HOL terms into terms of type `term`, a HOL model of HOL terms. For each constant c (of arbitrary type) it defines a constant c' of type `term` and proves a theorem that expresses equivalence (Sect. 3).
3. There are multiple *compiler phases* that eliminate certain constructs from the `term` type, until we arrive at the CakeML expression type. Most phases target a different intermediate term type (Sect. 5).

The first two stages are preprocessing, are implemented in ML and produce certificate theorems. Only these stages are specific to Isabelle. The third (and main) stage is implemented completely in the logic HOL, without recourse to ML. Its correctness is verified once and for all.¹

2 Related Work

There is existing work in the Coq [2, 15] and HOL [30] communities for proof producing or verified extraction of functions defined in the logic. Anand *et al.* [2] present work in progress on a verified compiler from Gallina (Coq’s specification language) via untyped intermediate languages to CompCert C light. They plan to connect their extraction routine to the CompCert compiler [26].

Translation of type classes into dictionaries is an important feature of Haskell compilers. In the setting of Isabelle/HOL, this has been described by Wenzel [44] and Krauss *et al.* [23]. Haftmann and Nipkow [17] use this construction to compile HOL definitions into target languages that do not support type classes, e.g. Standard ML and OCaml. In this work, we provide a certifying translation that eliminates type classes inside the logic.

Compilation of pattern matching is well understood in literature [3, 36, 38]. In this work, we contribute a transformation of sets of equations with pattern matching on the left-hand side into a single equation with nested pattern matching on the right-hand side. This is implemented and verified inside Isabelle.

Besides CakeML, there are many projects for verified compilers for functional programming languages of various degrees of sophistication and realism (e.g.

¹ All Isabelle definitions and proofs can be found on the paper website: <https://lars.hupel.info/research/codegen/>, or archived as <https://doi.org/10.5281/zenodo.1167616>.

[4, 11, 14]). Particularly modular is the work by Neis *et al.* [31] on a verified compiler for an ML-like imperative source language. The main distinguishing feature of our work is that we start from a set of higher-order recursion equations with pattern matching on the left-hand side rather than a lambda calculus with pattern matching on the right-hand side. On the other hand we stand on the shoulders of CakeML which allows us to bypass all complications of machine code generation. Note that much of our compiler is not specific to CakeML and that it would be possible to retarget it to, for example, Pilsner abstract syntax with moderate effort.

Finally, Fallenstein and Kumar [13] have presented a model of HOL inside HOL using large cardinals, including a reflection proof principle.

3 Deep Embedding

Starting with a HOL definition, we derive a new, *reified* definition in a deeply embedded term language depicted in Fig. 1a. This term language corresponds closely to the `term` datatype of Isabelle’s implementation (using de Bruijn indices [10]), but without types and schematic variables.

To establish a formal connection between the original and the reified definitions, we use a *logical relation*, a concept that is well-understood in literature [20] and can be nicely implemented in Isabelle using type classes. Note that the use of type classes here is restricted to correctness proofs; it is not required for the execution of the compiler itself. That way, there is no contradiction to the elimination of type classes occurring in a previous stage.

Notation. We abbreviate `App t u` to $t \$ u$ and `Abs t` to Λt . Other term types introduced later in this paper use the same conventions. We reserve λ for abstractions in HOL itself. Typing judgments are written with a double colon: $t :: \tau$.

Embedding Operation. Embedding is implemented in ML. We denote this operation using angle brackets: $\langle t \rangle$, where t is an arbitrary HOL expression and the result $\langle t \rangle$ is a HOL value of type `term`. It is a purely syntactic transformation, without preliminary evaluation or reduction, and it discards type information. The following examples illustrate this operation and typographical conventions concerning variables and constants:

$$\langle x \rangle = \text{Free } "x" \quad \langle f \rangle = \text{Const } "f" \quad \langle \lambda x. f \ x \rangle = \Lambda (\langle f \rangle \$ \text{Bound } 0)$$

Small-Step Semantics. Figure 1b specifies the small-step semantics for `term`. It is reminiscent of *higher-order term rewriting*, and modelled closely after equality in HOL. The basic idea is that if the proposition $t = u$ can be proved equationally in HOL (without symmetry), then $R \vdash \langle t \rangle \longrightarrow^* \langle u \rangle$ holds (where $R :: (\text{term} \times \text{term}) \text{ set}$). We call R the *rule set*. It is the result of translating a set of defining equations $lhs = rhs$ into pairs $(\langle lhs \rangle, \langle rhs \rangle) \in R$.

datatype term =	
Const string	
Free string	
Abs term	
Bound nat	
App term term	
	$\text{STEP} \frac{(lhs, rhs) \in R \quad \text{match } lhs \ t = \text{Some } \sigma}{R \vdash t \longrightarrow \text{subst } \sigma \ rhs}$
	$\text{BETA} \frac{\text{closed } t'}{R \vdash (\lambda t) \$ t' \longrightarrow t[t']} \quad \text{FUN} \frac{R \vdash t \longrightarrow t'}{R \vdash t \$ u \longrightarrow t' \$ u}$
	$\text{ARG} \frac{R \vdash u \longrightarrow u'}{R \vdash t \$ u \longrightarrow t \$ u'}$
(a) Abstract syntax of de Bruijn terms	(b) Small-step semantics

Fig. 1. Basic syntax and semantics of the term type

Rule STEP performs a rewrite step by picking a rewrite rule from R and rewriting the term at the root. For that purpose, **match** and **subst** are (mostly) standard first-order matching and substitution (see Sect. 4 for details).

Rule BETA performs β -reduction. Type **term** represents bound variables by de Bruijn indices. The notation $t[t']$ represents the substitution of the outermost bound variable in t with t' .

Our semantics does not constitute a fully-general higher-order term rewriting system, because we do not allow substitution under binders. For de Bruijn terms, this would pose no problem, but as soon as we introduce named bound variables, substitution under binders requires dealing with capture. To avoid this altogether, all our semantics expect terms that are substituted into abstractions to be closed. However, this does not mean that we restrict ourselves to any particular evaluation order. Both call-by-value and call-by-name can be used in the small-step semantics. But later on, the target semantics will only use call-by-value.

Embedding Relation. We denote the concept that an embedded term t corresponds to a HOL term a of type τ w.r.t. rule set R with the syntax $R \vdash t \approx a$. If we want to be explicit about the type, we index the relation: \approx_τ .

For ground types, this can be defined easily. For example, the following two rules define \approx_{nat} :

$$\frac{}{R \vdash \langle 0 \rangle \approx_{\text{nat}} 0} \quad \frac{R \vdash \langle t \rangle \approx_{\text{nat}} n}{R \vdash \langle \text{Suc } t \rangle \approx_{\text{nat}} \text{Suc } n}$$

Definitions of \approx for arbitrary datatypes without nested recursion can be derived mechanically in the same fashion as for **nat**, where they constitute one-to-one relations. Note that for ground types, \approx ignores R . The reason why \approx is parametrized on R will become clear in a moment.

For function types, we follow Myreen and Owen’s approach [30]. The statement $R \vdash t \approx f$ can be interpreted as “ $t \$ \langle a \rangle$ can be rewritten to $\langle f a \rangle$ for all a ”. Because this might involve applying a function definition from R , the \approx relation must be indexed by the rule set. As a notational convenience, we define

another relation $R \vdash t \downarrow x$ to mean that there is a t' such that $R \vdash t \longrightarrow^* t'$ and $R \vdash t' \approx x$. Using this notation, we formally define \approx for functions as follows:

$$R \vdash t \approx f \leftrightarrow (\forall u x. R \vdash u \downarrow x \rightarrow R \vdash t \$ u \downarrow f x)$$

Example. As a running example, we will use the `map` function on lists:

$$\begin{aligned} \text{map } f \ [] &= [] \\ \text{map } f \ (x \# xs) &= f \ x \# \text{map } f \ xs \end{aligned}$$

The result of embedding this function is a set of rules `map'`:

`map' =`
`{(Const "List.list.map" $ Free "f" $ (Const "List.list.Cons" $ Free "x21" $ Free "x22"),`
`Const "List.list.Cons" $ (Free "f" $ Free "x21") $...),`
`(Const "List.list.map" $ Free "f" $ Const "List.list.Nil",`
`Const "List.list.Nil")}`

together with the theorem $\text{map}' \vdash \text{Const "List.list.map"} \downarrow \text{map}$, which is proven by simple induction over `map`. Constant names like `"List.list.map"` come from the fully-qualified internal names in HOL.

The induction principle for the proof arises from the use of the **fun** command that is used to define recursive functions in HOL [22]. But the user is also allowed to specify custom equations for functions, in which case we will use heuristics to generate and prove the appropriate induction theorem. For simplicity, we will use the term (*defining*) *equation* uniformly to refer to any set of equations, either default ones or ones specified by the user. Embedding partially-specified functions – in particular, proving the certificate theorem about them – is currently not supported. In the future, we plan to leverage the domain predicate as produced by **fun** to generate conditional theorems.

4 Terms, Matching and Substitution

The compiler transforms the initial term type (Fig. 1a) through various intermediate stages. This section gives an overview and introduces necessary terminology.

Preliminaries. The function arrow in HOL is \Rightarrow . The cons operator on lists is the infix `#`.

Throughout the paper, the concept of *mappings* is pervasive: We use the type notation $\alpha \rightarrow \beta$ to denote a function $\alpha \Rightarrow \beta$ **option**. In certain contexts, a mapping may also be called an *environment*. We write mapping literals using brackets: $[a \Rightarrow x, b \Rightarrow y, \dots]$. If it is clear from the context that σ is defined on a , we often treat the lookup σa as returning an $x :: \beta$.

The functions `dom :: ($\alpha \rightarrow \beta$) \Rightarrow α set` and `range :: ($\alpha \rightarrow \beta$) \Rightarrow β set` return the *domain* and *range* of a mapping, respectively.

Dropping entries from a mapping is denoted by $\sigma - k$, where σ is a mapping and k is either a single key or a set of keys. We use $\sigma' \subseteq \sigma$ to denote that σ' is a sub-mapping of σ , that is, $\text{dom } \sigma' \subseteq \text{dom } \sigma$ and $\forall a \in \text{dom } \sigma'. \sigma' a = \sigma a$.

Merging two mappings σ and ρ is denoted with $\sigma ++ \rho$. It constructs a new mapping with the union domain of σ and ρ . Entries from ρ override entries from σ . That is, $\rho \subseteq \sigma ++ \rho$ holds, but not necessarily $\sigma \subseteq \sigma ++ \rho$.

All mappings and sets are assumed to be finite. In the formalization, this is enforced by using subtypes of \rightarrow and **set**. Note that one cannot define datatypes by recursion through sets for cardinality reasons. However, for finite sets, it is possible. This is required to construct the various term types. We leverage facilities of Blanchette *et al.*'s **datatype** command to define these subtypes [7].

Standard Functions. All type constructors that we use (\rightarrow , **set**, **list**, **option**, ...) support the standard operations **map** and **rel**. For lists, **map** is the regular covariant map. For mappings, the function has the type $(\beta \Rightarrow \gamma) \Rightarrow (\alpha \rightarrow \beta) \Rightarrow (\alpha \rightarrow \gamma)$. It leaves the domain unchanged, but applies a function to the range of the mapping.

Function rel_τ lifts a binary predicate $P :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$ to the type constructor τ . We call this lifted relation the *relator* for a particular type.

For datatypes, its definition is structural, for example:

$$\frac{}{\text{rel}_{\text{list}} P [] []} \quad \frac{\text{rel}_{\text{list}} P \text{ xs ys} \quad P \text{ x y}}{\text{rel}_{\text{list}} P (x \# \text{xs}) (y \# \text{ys})}$$

For sets and mappings, the definition is a little bit more subtle.

Definition 1 (Set relator). For each element $a \in A$, there must be a corresponding element $b \in B$ such that $P a b$, and vice versa. Formally:

$$\text{rel}_{\text{set}} P A B \leftrightarrow (\forall x \in A. \exists y \in B. P x y) \wedge (\forall y \in B. \exists x \in A. P x y)$$

Definition 2 (Mapping relator). For each a , $m a$ and $n a$ must be related according to $\text{rel}_{\text{option}} P$. Formally:

$$\text{rel}_{\text{mapping}} P m n \leftrightarrow (\forall a. \text{rel}_{\text{option}} P (m a) (n a))$$

Term Types. There are four distinct term types: **term**, **nterm**, **pterm**, and **stern**. All of them support the notions of free variables, matching and substitution. Free variables are always a finite set of strings. Matching a term against a *pattern* yields an optional mapping of type **string** $\rightarrow \alpha$ from free variable names to terms.

Note that the type of patterns is itself **term** instead of a dedicated pattern type. The reason is that we have to subject patterns to a linearity constraint anyway and may use this constraint to carve out the relevant subset of terms:

Definition 3. A term is linear if there is at most one occurrence of any variable, it contains no abstractions, and in an application $f \$ x$, f must not be a free variable. The HOL predicate is called $\text{linear} :: \text{term} \Rightarrow \text{bool}$.

Because of the similarity of operations across the term types, they are all instances of the `term` type class. Note that in Isabelle, classes and types live in different namespaces. The `term` type and the `term` type class are separate entities.

Definition 4. A term type τ supports the operations $\text{match}::\text{term} \Rightarrow \tau \Rightarrow (\text{string} \rightarrow \tau)$, $\text{subst}::(\text{string} \rightarrow \tau) \Rightarrow \tau \Rightarrow \tau$ and $\text{frees}::\tau \Rightarrow \text{string set}$. We also define the following derived functions:

- `matches` matches a list of patterns and terms sequentially, producing a single mapping
- `closed` t is an abbreviation for $\text{frees } t = \emptyset$
- `closed` σ is an overloading of `closed`, denoting that all values in a mapping are closed

Additionally, some (obvious) axioms have to be satisfied. We do not strive to fully specify an abstract term algebra. Instead, the axioms are chosen according to the needs of this formalization.

A notable deviation from matching as discussed in term rewriting literature is that the result of matching is only well-defined if the pattern is linear.

Definition 5. An equation is a pair of a pattern (left-hand side) and a term (right-hand side). The pattern is of the form $f \$p_1 \$ \dots \p_n , where f is a constant (i.e. of the form `Const name`). We refer to both f or name interchangeably as the function symbol of the equation.

Following term rewriting terminology, we sometimes refer to an equation as *rule*.

4.1 De Bruijn terms (`term`)

The definition of `term` is almost an exact copy of Isabelle’s internal term type, with the notable omissions of type information and schematic variables (Fig. 1a). The implementation of β -reduction is straightforward via index shifting of bound variables.

4.2 Named Bound Variables (`nterm`)

datatype `nterm` = `Nconst string` | `Nvar string` | `Nabs string nterm` | `Napp nterm nterm`

The `nterm` type is similar to `term`, but removes the distinction between *bound* and *free* variables. Instead, there are only named variables. As mentioned in the previous section, we forbid substitution of terms that are not closed in order to avoid capture. This is also reflected in the syntactic side conditions of the correctness proofs (Sect. 5.1).

4.3 Explicit Pattern Matching (**pterm**)

datatype pterm =

Pconst string | Pvar string | Pabs ((term \times pterm) set) | Papp pterm pterm

Functions in HOL are usually defined using *implicit* pattern matching, that is, the terms p_i occurring on the left-hand side $\langle f\ p_1 \ \dots\ p_n \rangle$ of an equation must be constructor patterns. This is also common among functional programming languages like Haskell or OCaml. CakeML only supports *explicit* pattern matching using case expressions. A function definition consisting of multiple defining equations must hence be translated to the form $f = \lambda x. \textbf{case } x \textbf{ of } \dots$. The elimination proceeds by iteratively removing the last parameter in the block of equations until none are left.

In our formalization, we opted to combine the notion of abstraction and case expression, yielding *case abstractions*, represented as the **Pabs** constructor. This is similar to the **fn** construct in Standard ML, which denotes an anonymous function that immediately matches on its argument [28]. The same construct also exists in Haskell with the **LambdaCase** language extension. We chose this representation mainly for two reasons: First, it allows for a simpler language grammar because there is only one (shared) constructor for abstraction and case expression. Second, the elimination procedure outlined above does not have to introduce fresh names in the process. Later, when translating to CakeML syntax, fresh names are introduced and proved correct in a separate step.

The set of pairs of pattern and right-hand side inside a case abstraction is referred to as *clauses*. As a short-hand notation, we use $\Lambda\{p_1 \Rightarrow t_1, p_2 \Rightarrow t_2, \dots\}$.

4.4 Sequential Clauses (**stern**)

datatype stern =

Sconst string | Svar string | Sabs ((term \times stern) list) | Sapp stern stern

In the term rewriting fragment of HOL, the order of rules is not significant. If a rule matches, it can be applied, regardless when it was defined or proven. This is reflected by the use of sets in the rule and term types. For CakeML, the rules need to be applied in a deterministic order, i.e. sequentially. The **stern** type only differs from **pterm** by using list instead of set. Hence, case abstractions use list brackets: $\Lambda[p_1 \Rightarrow t_1, p_2 \Rightarrow t_2, \dots]$.

4.5 Irreducible Terms (**value**)

CakeML distinguishes between *expressions* and *values*. Whereas expressions may contain free variables or β -redexes, values are closed and fully evaluated. Both have a notion of abstraction, but values differ from expressions in that they contain an environment binding free variables.

Consider the expression $(\lambda x. \lambda y. x) (\lambda z. z)$, which is rewritten (by β -reduction) to $\lambda y. \lambda z. z$. Note how the bound variable x disappears, since it is replaced. This

is contrary to how programming languages are usually implemented: evaluation does not happen by substituting the argument term t for the bound variable x , but by recording the binding $x \mapsto t$ in an environment [24]. A pair of an abstraction and an environment is usually called a *closure* [25, 41].

In CakeML, this means that evaluation of the above expression results in the closure

$$(\lambda y.x, ["x" \mapsto (\lambda z.z, [])])$$

Note the nested structure of the closure, whose environment itself contains a closure.

To reflect this in our formalization, we introduce a type `value` of values (explanation inline):

```
datatype value =
  (* constructor value: a data constructor applied to multiple values *)
  Vconstr string (value list) |
  (* closure: clauses combined with an environment mapping variables to values *)
  Vabs ((term × stern) list) (string → value) |
  (* recursive closures: a group of mutually recursive function bodies with an environment *)
  Vrecabs (string → ((term × stern) list)) string (string → value)
```

The above example evaluates to the closure:

$$\mathbf{Vabs} \left[\langle y \rangle \Rightarrow \langle x \rangle \right] \left["x" \mapsto \mathbf{Vabs} [\langle z \rangle \Rightarrow \langle z \rangle] [] \right]$$

The third case for recursive closures only becomes relevant when we conflate variables and constants. As long as the rule set rs is kept separate, recursive calls are straightforward: the appropriate definition for the constant can be looked up there. CakeML knows no such distinction between constants and variables, hence everything has to reside in a single environment σ .

Consider this example of `odd` and `even`:

$$\begin{array}{ll} \text{odd } 0 = \text{False} & \text{even } 0 = \text{True} \\ \text{odd } (\text{Suc } n) = \text{even } n & \text{even } (\text{Suc } n) = \text{odd } n \end{array}$$

When evaluating the term `odd k` , the definitions of `even` and `odd` themselves must be available in the environment captured in the definition of `odd`. However, it would be cumbersome in HOL to construct such a `Vabs` that refers to itself. Instead, we capture the expressions used to define `odd` and `even` in a recursive closure. Other encodings might be possible, but since we are targeting CakeML, we are opting to model it in a similar way as its authors do.

For the above example, this would result in the following global environment:

$$\begin{aligned} &["\text{odd}" \mapsto \mathbf{Vrecabs} \text{ css } "\text{odd}" [], "\text{even}" \mapsto \mathbf{Vrecabs} \text{ css } "\text{even}" []] \\ &\text{where } \text{css} = ["\text{odd}" \mapsto [\langle 0 \rangle \Rightarrow \langle \text{False} \rangle, \langle \text{Suc } n \rangle \Rightarrow \langle \text{even } n \rangle], \\ &\quad "\text{even}" \mapsto [\langle 0 \rangle \Rightarrow \langle \text{True} \rangle, \langle \text{Suc } n \rangle \Rightarrow \langle \text{odd } n \rangle]] \end{aligned}$$

Note that in the first line, the right-hand sides are values, but in *css*, they are expressions. The additional **string** argument of **Vrecabs** denotes the selected function. When evaluating an application of a recursive closure to an argument (β -reduction), the semantics adds all constituent functions of the closure to the environment used for recursive evaluation.

5 Intermediate Semantics and Compiler Phases

In this section, we will discuss the progression from de Bruijn based term language with its small-step semantics given in Fig. 1a to the final CakeML semantics. The compiler starts out with terms of type **term** and applies multiple phases to eliminate features that are not present in the CakeML source language.

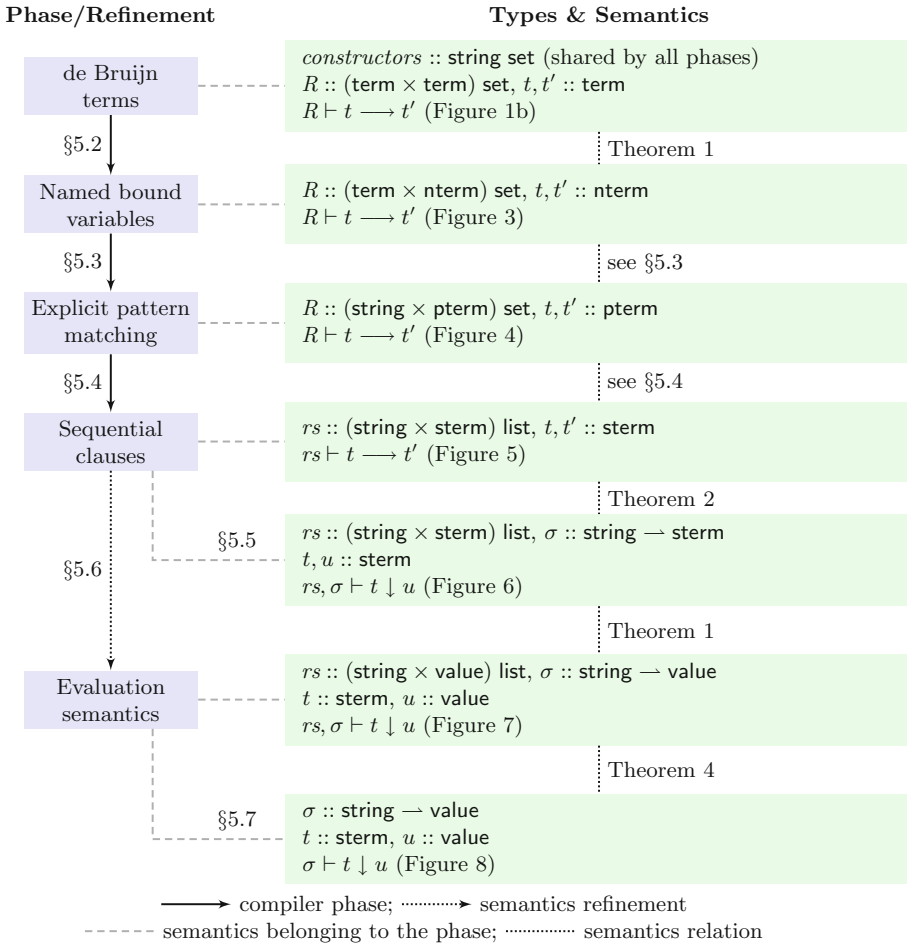


Fig. 2. Intermediate semantics and compiler phases

Types **term**, **nterm** and **pterm** each have a small-step semantics only. Type **stern** has a small-step and several intermediate big-step semantics that bridge the gap to CakeML. An overview of the intermediate semantics and compiler phases is depicted in Fig. 2. The left-hand column gives an overview of the different phases. The right-hand column gives the types of the rule set and the semantics for each phase; you may want to skip it upon first reading.

$$\text{STEP} \frac{(lhs, rhs) \in R \quad \text{match } lhs \ t = \text{Some } \sigma}{R \vdash t \longrightarrow \text{subst } \sigma \ rhs} \quad \text{BETA} \frac{\text{closed } t'}{R \vdash (\lambda x. t) \$ t' \longrightarrow \text{subst } [x \mapsto t'] \ t}$$

Fig. 3. Small-step semantics for **nterm** with named bound variables

5.1 Side Conditions

All of the following semantics require some side conditions on the rule set. These conditions are purely syntactic. As an example we list the conditions for the correctness of the first compiler phase:

- Patterns must be linear, and constructors in patterns must be fully applied.
- Definitions must have at least one parameter on the left-hand side (Sect. 5.6).
- The right-hand side of an equation refers only to free variables occurring in patterns on the left-hand side and contain no dangling de Bruijn indices.
- There are no two defining equations $lhs = rhs_1$ and $lhs = rhs_2$ such that $rhs_1 \neq rhs_2$.
- For each pair of equations that define the same constant, their arity must be equal and their patterns must be compatible (Sect. 5.3).
- There is at least one equation.
- Variable names occurring in patterns must not overlap with constant names (Sect. 5.7).
- Any occurring constants must either be defined by an equation or be a constructor.

The conditions for the subsequent phases are sufficiently similar that we do not list them again.

In the formalization, we use named contexts to fix the rules and assumptions on them (*locales* in Isabelle terminology). Each phase has its own locale, together with a proof that after compilation, the preconditions of the next phase are satisfied. Correctness proofs assume the above conditions on R and similar conditions on the term that is reduced. For brevity, this is usually omitted in our presentation.

5.2 Naming Bound Variables: From term to nterm

Isabelle uses de Bruijn indices in the term language for the following two reasons: For substitution, there is no need to rename bound variables. Additionally, α -equivalent terms are equal. In implementations of programming languages, these advantages are not required: Typically, substitutions do not happen inside abstractions, and there is no notion of equality of functions. Therefore CakeML uses named variables and in this compilation step, we get rid of de Bruijn indices.

The “named” semantics is based on the `nterm` type. The rules that are changed from the original semantics (Fig. 1b) are given in Fig. 3 (FUN and ARG remain unchanged). Notably, β -reduction reuses the substitution function.

For the correctness proof, we need to establish a correspondence between terms and nterms. Translation from `nterm` to `term` is trivial: Replace bound variables by the number of abstractions between occurrence and where they were bound in, and keep free variables as they are. This function is called `nterm_to_term`.

The other direction is not unique and requires introduction of *fresh* names for bound variables. In our formalization, we have chosen to use a *monad* to produce these names. This function is called `term_to_nterm`. We can also prove the obvious property `nterm_to_term (term_to_nterm t) = t`, where t is a `term` without dangling de Bruijn indices.

Generation of fresh names in general can be thought of as picking a string that is not an element of a (finite) set of already existing names. For Isabelle, the *Nominal* framework [42, 43] provides support for reasoning over fresh names, but unfortunately, its definitions are not executable.

Instead, we chose to model generation of fresh names as a monad α `fresh` with the following primitive operations in addition to the monad operations:

$$\begin{aligned} \text{run} &:: \alpha \text{ fresh} \Rightarrow \text{string set} \Rightarrow \alpha \\ \text{fresh_name} &:: \text{string fresh} \end{aligned}$$

In our implementation, we have chosen to represent α `fresh` as roughly isomorphic to the state monad.

Compilation of a rule set proceeds by translation of the right-hand side of all rules:

$$\text{compile } R = \{(p, \text{term_to_nterm } t) \mid (p, t) \in R\}$$

The left-hand side is left unchanged for two reasons: function `match` expects an argument of type `term` (see Sect. 4), and patterns do not contain abstractions or bound variables.

Theorem 1 (Correctness of compilation). *Assuming a step can be taken with the compiled rule set, it can be reproduced with the original rule set.*

$$\frac{\text{compile } R \vdash t \longrightarrow u \quad \text{closed } t}{R \vdash \text{nterm_to_term } t \longrightarrow \text{nterm_to_term } u}$$

We prove this by induction over the semantics (Fig. 3).

$$\begin{array}{c}
\text{BETA} \frac{(pat, rhs) \in C \quad \text{match } pat \ t = \text{Some } \sigma \quad \text{closed } t}{R \vdash (\Lambda C) \$ t \longrightarrow \text{subst } \sigma \ rhs} \\
\\
\text{STEP}' \frac{(name, rhs) \in R}{R \vdash \text{Pconst } name \longrightarrow rhs}
\end{array}$$

Fig. 4. Small-step semantics for **pterm** with pattern matching

5.3 Explicit Pattern Matching: From **nterm** to **pterm**

Usually, functions in HOL are defined using *implicit* pattern matching, that is, the left-hand side of an equation is of the form $\langle f \ p_1 \ \dots \ p_n \rangle$, where the p_i are patterns over datatype constructors. For any given function f , there may be multiple such equations. In this compilation step, we transform sets of equations for f defined using implicit pattern matching into a single equation for f of the form $\langle f \rangle = \Lambda C$, where C is a set of clauses.

The strategy we employ currently requires successive elimination of a single parameter from right to left, in a similar fashion as Slind’s pattern matching compiler [38, Sect. 3.3.1]. Recall our running example (**map**). It has arity 2. We omit the brackets $\langle \rangle$ for brevity. First, the list parameter gets eliminated:

$$\begin{aligned}
\text{map } f &= \lambda [] \Rightarrow [] \\
&\quad | x \# xs \Rightarrow f \ x \# \text{map } f \ xs
\end{aligned}$$

Finally, the function parameter gets eliminated:

$$\begin{aligned}
\text{map} &= \lambda f \Rightarrow (\lambda [] \Rightarrow []) \\
&\quad | x \# xs \Rightarrow f \ x \# \text{map } f \ xs
\end{aligned}$$

This has now arity 0 and is defined by a twice-nested abstraction.

Semantics. The target semantics is given in Fig. 4 (the **FUN** and **ARG** rules from previous semantics remain unchanged). We start out with a rule set R that allows only implicit pattern matching. After elimination, only explicit pattern matching remains. The modified **STEP** rule merely replaces a constant by its definition, without taking arguments into account.

Restrictions. For the transformation to work, we need a strong assumption about the structure of the patterns p_i to avoid the following situation:

$$\begin{aligned}
\text{map } f \ [] &= [] \\
\text{map } g \ (x \# xs) &= g \ x \# \text{map } g \ xs
\end{aligned}$$

Through elimination, this would turn into:

$$\begin{aligned}
\text{map} &= \lambda f \Rightarrow (\lambda [] \Rightarrow []) \\
&\quad | g \Rightarrow (\lambda x \# xs \Rightarrow f \ x \# \text{map } f \ xs)
\end{aligned}$$

$$\text{STEP} \frac{(name, rhs) \in R}{R \vdash \text{Sconst } name \longrightarrow rhs} \quad \text{BETA} \frac{\text{first_match } cs \ t = \text{Some } (\sigma, rhs) \quad \text{closed } t}{R \vdash (\Lambda \ cs) \$ t \longrightarrow \text{subst } \sigma \ rhs}$$

Fig. 5. Small-step semantics for **stern**

Even though the original equations were non-overlapping, we suddenly obtained an abstraction with two overlapping patterns. Slind observed a similar problem [38, Sect. 3.3.2] in his algorithm. Therefore, he only permits *uniform* equations, as defined by Wadler [36, Sect. 5.5]. Here, we can give a formal characterization of our requirements as a computable function on pairs of patterns:

```
fun pat_compat :: term  $\Rightarrow$  term  $\Rightarrow$  bool where
pat_compat (t1 $ t2) (u1 $ u2)  $\leftrightarrow$  pat_compat t1 u1  $\wedge$  (t1 = u1  $\rightarrow$  pat_compat t2 u2)
pat_compat t u  $\leftrightarrow$  (overlapping t u  $\rightarrow$  t = u)
```

This compatibility constraint ensures that any two overlapping patterns (of the same column) $p_{i,k}$ and $p_{j,k}$ are equal and are thus appropriately grouped together in the elimination procedure. We require all defining equations of a constant to be mutually compatible. Equations violating this constraint will be flagged during embedding (Sect. 3), whereas the pattern elimination algorithm always succeeds.

While this rules out some theoretically possible pattern combinations (e.g. the *diagonal* function [36, Sect. 5.5]), in practice, we have not found this to be a problem: All of the function definitions we have tried (Sect. 8) satisfied pattern compatibility (after automatic renaming of pattern variables). As a last resort, the user can manually instantiate function equations. Although this will always lead to a pattern compatible definition, it is not done automatically, due to the potential blow-up.

Discussion. Because this compilation phase is both non-trivial and has some minor restrictions on the set of function definitions that can be processed, we may provide an alternative implementation in the future. Instead of eliminating patterns from right to left, patterns may be grouped in tuples. The above example would be translated into:

$$\begin{aligned} \text{map} &= \lambda (f, []) \Rightarrow [] \\ &\quad | (f, x \# xs) \Rightarrow f \ x \# \text{map } f \ xs \end{aligned}$$

We would then leave the compilation of patterns for the CakeML compiler, which has no pattern compatibility restriction.

The obvious disadvantage however is that this would require the knowledge of a tuple type in the term language which is otherwise unaware of concrete datatypes.

5.4 Sequentialization: From **pterm** to **stern**

The semantics of **pterm** and **stern** differ only in rule STEP and BETA. Figure 5 shows the modified rules. Instead of any matching clause, the first matching clause in a case abstraction is picked.

For the correctness proof, the order of clauses does not matter: we only need to prove that a step taken in the sequential semantics can be reproduced in the unordered semantics. As long as no rules are dropped, this is trivially true. For that reason, the compiler orders the clauses lexicographically. At the same time the rules are also converted from type $(\text{string} \times \text{pterm}) \text{ set}$ to $(\text{string} \times \text{sterm}) \text{ list}$. Below, rs will always denote a list of the latter type.

$$\begin{array}{c}
\text{CONST} \frac{(name, rhs) \in rs}{rs, \sigma \vdash \text{Sconst } name \downarrow rhs} \quad \text{VAR} \frac{\sigma \text{ name} = \text{Some } v}{rs, \sigma \vdash \text{Svar } name \downarrow v} \\
\text{ABS} \frac{}{rs, \sigma \vdash \Lambda cs \downarrow \Lambda [(pat, \text{subst } (\sigma - \text{frees } pat) t \mid (pat, t) \leftarrow cs)]} \\
\text{COMB} \frac{rs, \sigma \vdash u \downarrow u' \quad \text{first_match } cs \ u' = \text{Some } (\sigma', rhs) \quad rs, \sigma \vdash \sigma' \vdash rhs \downarrow v}{rs, \sigma \vdash t \$ u \downarrow v} \\
\text{CONSTR} \frac{name \in \text{constructors} \quad rs, \sigma \vdash t_1 \downarrow u_1 \quad \dots \quad rs, \sigma \vdash t_n \downarrow u_n}{rs, \sigma \vdash \text{Sconst } name \$ t_1 \$ \dots \$ t_n \downarrow \text{Sconst } name \$ u_1 \$ \dots \$ u_n}
\end{array}$$

Fig. 6. Big-step semantics for **sterm**

5.5 Big-Step Semantics for **sterm**

This big-step semantics for **sterm** is not a compiler phase but moves towards the desired evaluation semantics. In this first step, we reuse the **sterm** type for evaluation results, instead of evaluating to the separate type **value**. This allows us to ignore environment capture in closures for now.

All previous \longrightarrow relations were parametrized by a rule set. Now the big-step predicate is of the form $rs, \sigma \vdash t \downarrow t'$ where $\sigma :: \text{string} \rightarrow \text{sterm}$ is a variable environment.

This semantics also introduces the distinction between *constructors* and *defined constants*. If C is a constructor, the term $\langle C t_1 \dots t_n \rangle$ is evaluated to $\langle C t'_1 \dots t'_n \rangle$ where the t'_i are the results of evaluating the t_i .

The full set of rules is shown in Fig. 6. They deserve a short explanation:

CONST. Constants are retrieved from the rule set rs .

VAR. Variables are retrieved from the environment σ .

ABS. In order to achieve the intended invariant, abstractions are evaluated to their fully substituted form.

COMB. Function application $t \$ u$ first requires evaluation of t into an abstraction Λcs and evaluation of u into an arbitrary term u' . Afterwards, we look for a clause matching u' in cs , which produces a local variable environment σ' , possibly overwriting existing variables in σ . Finally, we evaluate the right-hand side of the clause with the combined global and local variable environment.

CONSTR. For a constructor application $\langle C t_1 \dots \rangle$, evaluate all t_i . The set *constructors* is an implicit parameter of the semantics.

$$\begin{array}{c}
\text{CONST} \frac{(name, rhs) \in rs}{rs, \sigma \vdash \text{Sconst } name \downarrow rhs} \quad \text{VAR} \frac{\sigma \text{ name} = \text{Some } v}{rs, \sigma \vdash \text{Svar } name \downarrow v} \\
\text{ABS} \frac{}{rs, \sigma \vdash \Lambda cs \downarrow \text{Vabs } cs \sigma} \\
\text{COMB} \frac{rs, \sigma \vdash u \downarrow v \quad \text{first_match } cs \ v = \text{Some } (\sigma'', rhs) \quad rs, \sigma' \uparrow\uparrow \sigma'' \vdash rhs \downarrow v'}{rs, \sigma \vdash t \$ u \downarrow v'} \\
\text{RECCOMB} \frac{rs, \sigma \vdash t \downarrow \text{Vrecabs } css \ name \ \sigma' \quad \text{first_match } cs \ v = \text{Some } (\sigma'', rhs) \quad rs, \sigma' \uparrow\uparrow \sigma'' \vdash rhs \downarrow v'}{rs, \sigma \vdash t \$ u \downarrow v'} \\
\text{CONSTR} \frac{name \in constructors \quad rs, \sigma \vdash t_1 \downarrow v_1 \quad \dots \quad rs, \sigma \vdash t_n \downarrow v_n}{rs, \sigma \vdash \text{Sconst } name \$ t_1 \$ \dots \$ t_n \downarrow \text{Vconstr } name [v_1, \dots, v_n]}
\end{array}$$

Fig. 7. Evaluation semantics from **term** to **value**

Lemma 1 (Closedness invariant). *If σ contains only closed terms, frees $t \subseteq \text{dom } \sigma$ and $rs, \sigma \vdash t \downarrow t'$, then t' is closed.*

Correctness of the big-step w.r.t. the small-step semantics is proved easily by induction on the former:

Lemma 2. *For any closed environment σ satisfying $\text{frees } t \subseteq \text{dom } \sigma$,*

$$rs, \sigma \vdash t \downarrow u \rightarrow rs \vdash \text{subst } \sigma \ t \longrightarrow^* u$$

By setting $\sigma = []$, we obtain:

Theorem 2 (Correctness). $rs, [] \vdash t \downarrow u \wedge \text{closed } t \rightarrow rs \vdash t \longrightarrow^* u$

5.6 Evaluation Semantics: Refining **term** to **value**

At this point, we introduce the concept of values into the semantics, while still keeping the rule set (for constants) and the environment (for variables) separate. The evaluation rules are specified in Fig. 7 and represent a departure from the original rewriting semantics: a term does not evaluate to another term but to an object of a different type, a **value**. We still use \downarrow as notation, because big-step and evaluation semantics can be disambiguated by their types.

The evaluation model itself is fairly straightforward. As explained in Sect. 4.5, abstraction terms are evaluated to closures capturing the current variable environment. Note that at this point, recursive closures are not treated differently from non-recursive closures. In a later stage, when rs and σ are merged, this distinction becomes relevant.

We will now explain each rule that has changed from the previous semantics:

ABS. Abstraction terms are evaluated to a closure capturing the current environment.

COMB. As before, in an application $t\$u$, t must evaluate to a closure $\mathbf{Vabs}\ cs\ \sigma'$. The evaluation result of u is then matched against the clauses cs , producing an environment σ'' . The right-hand side of the clause is then evaluated using $\sigma' \uparrow\uparrow \sigma''$; the original environment σ is effectively discarded.

RECCOMB. Similar as above. Finding the matching clause is a two-step process: First, the appropriate clause list is selected by name of the currently active function. Then, matching is performed.

CONSTR. As before, for an n -ary application $\langle C\ t_1\ \dots \rangle$, where C is a data constructor, we evaluate all t_i . The result is a $\mathbf{Vconstr}$ value.

Conversion Between *term* and *value*. To establish a correspondence between evaluating a term to an *term* and to a *value*, we apply the same trick as in Sect. 5.2. Instead of specifying a complicated relation, we translate *value* back to *term*: simply apply the substitutions in the captured environments to the clauses.

The translation rules for \mathbf{Vabs} and $\mathbf{Vrecabs}$ are kept similar to the **ABS** rule from the big-step semantics (Fig. 6). Roughly speaking, the big-step semantics always keeps terms fully substituted, whereas the evaluation semantics defers substitution.

Similarly to Sect. 5.2, we can also define a function $\mathbf{term_to_value} :: \mathbf{term} \Rightarrow \mathbf{value}$ and prove that one function is the inverse of the other.

Matching. The *value* type, instead of using binary function application as all other term types, uses n -ary constructor application. This introduces a conceptual mismatch between (binary) patterns and values. To make the proofs easier, we introduce an intermediate type of n -ary patterns. This intermediate type can be optimized away by fusion.

Correctness. The correctness proof requires a number of interesting lemmas.

Lemma 3 (Substitution before evaluation). *Assuming that a term t can be evaluated to a value u given a closed environment σ , it can be evaluated to the same value after substitution with a sub-environment σ' . Formally: $rs, \sigma \vdash t \downarrow u \wedge \sigma' \subseteq \sigma \rightarrow rs, \sigma \vdash \mathbf{subst}\ \sigma'\ t \downarrow u$*

This justifies the “pre-substitution” exhibited by the **ABS** rule in the big-step semantics in contrast to the environment-capturing **ABS** rule in the evaluation semantics.

Theorem 3 (Correctness). *Let σ be a closed environment and t a term which only contains free variables in $\mathbf{dom}\ \sigma$. Then, an evaluation to a value $rs, \sigma \vdash t \downarrow v$ can be reproduced in the big-step semantics as $rs', \mathbf{map}\ \mathbf{value_to_term}\ \sigma \vdash t \downarrow \mathbf{value_to_term}\ v$, where $rs' = [(name, \mathbf{value_to_term}\ rhs) \mid (name, rhs) \leftarrow rs]$.*

Instantiating the Correctness Theorem. The correctness theorem states that, for any given evaluation of a term t with a given environment rs, σ containing values, we can reproduce that evaluation in the big-step semantics using a derived list of rules rs' and an environment σ' containing terms that are generated by the `value_to_term` function. But recall the diagram in Fig. 2. In our scenario, we start with a given rule set of terms (that has been compiled from a rule set of terms). Hence, the correctness theorem only deals with the opposite direction.

It remains to construct a suitable rs such that applying `value_to_term` to it yields the given `term` rule set. We can exploit the side condition (Sect. 5.1) that all bindings define functions, not constants:

Definition 6 (Global clause set). *The mapping $\text{global_css} :: \text{string} \rightarrow ((\text{term} \times \text{term}) \text{ list})$ is obtained by stripping the `Sabs` constructors from all definitions and converting the resulting list to a mapping.*

For each definition with name f we define a corresponding term $v_f = \text{Vrecabs } \text{global_css } f []$. In other words, each function is now represented by a recursive closure bundling all functions. Applying `value_to_term` to v_f returns the original definition of f . Let rs denote the original `term` rule set and rs_v the environment mapping all f 's to the v_f 's.

The variable environments σ and σ' can safely be set to the empty mapping, because top-level terms are evaluated without any free variable bindings.

Corollary 1 (Correctness). $rs_v, [] \vdash t \downarrow v \rightarrow rs, [] \vdash t \downarrow \text{value_to_term } v$

Note that this step was not part of the compiler (although rs_v is computable) but it is a refinement of the semantics to support a more modular correctness proof.

Example. Recall the `odd` and `even` example from Sect. 4.5. After compilation to `term`, the rule set looks like this:

$$rs = \{("odd", \text{Sabs } [\langle 0 \rangle \Rightarrow \langle \text{False} \rangle, \langle \text{Suc } n \rangle \Rightarrow \langle \text{even } n \rangle]), \\ ("even", \text{Sabs } [\langle 0 \rangle \Rightarrow \langle \text{True} \rangle, \langle \text{Suc } n \rangle \Rightarrow \langle \text{odd } n \rangle])\}$$

This can be easily transformed into the following global clause set:

$$\text{global_css} = ["odd" \mapsto [\langle 0 \rangle \Rightarrow \langle \text{False} \rangle, \langle \text{Suc } n \rangle \Rightarrow \langle \text{even } n \rangle], \\ "even" \mapsto [\langle 0 \rangle \Rightarrow \langle \text{True} \rangle, \langle \text{Suc } n \rangle \Rightarrow \langle \text{odd } n \rangle]]$$

Finally, rs_v is computed by creating a recursive closure for each function:

$$rs_v = ["odd" \mapsto \text{Vrecabs } \text{global_css } "odd" [], \\ "even" \mapsto \text{Vrecabs } \text{global_css } "even" []]$$

$$\begin{array}{c}
\text{CONST} \frac{\text{name} \notin \text{constructors} \quad \sigma \text{ name} = \text{Some } v}{\sigma \vdash \text{Sconst name} \downarrow v} \\
\\
\text{VAR} \frac{\sigma \text{ name} = \text{Some } v}{\sigma \vdash \text{Svar name} \downarrow v} \quad \text{ABS} \frac{}{\sigma \vdash \Lambda \text{ cs} \downarrow \text{Vabs cs } \sigma} \\
\\
\text{COMB} \frac{\sigma \vdash u \downarrow v \quad \text{first_match } \text{cs } v = \text{Some } (\sigma'', \text{rhs}) \quad \sigma' \dashv\vdash \sigma'' \vdash \text{rhs} \downarrow v' \quad \sigma \vdash t \downarrow \text{Vabs cs } \sigma'}{\sigma \vdash t \$ u \downarrow v'} \\
\\
\text{RECCOMB} \frac{\text{css name} = \text{Some } \text{cs} \quad \sigma \vdash u \downarrow v \quad \text{first_match } \text{cs } v = \text{Some } (\sigma'', \text{rhs}) \quad \sigma' \dashv\vdash \text{mk_rec_env } \text{css } \sigma' \dashv\vdash \sigma'' \vdash \text{rhs} \downarrow v'}{\sigma \vdash t \$ u \downarrow v'} \\
\\
\text{CONSTR} \frac{\text{name} \in \text{constructors} \quad \sigma \vdash t_1 \downarrow v_1 \quad \dots \quad \sigma \vdash t_n \downarrow v_n}{\sigma \vdash \text{Sconst name } \$ t_1 \$ \dots \$ t_n \downarrow \text{Vconstr name } [v_1, \dots, v_n]}
\end{array}$$

Fig. 8. ML-style evaluation semantics

5.7 Evaluation with Recursive Closures

CakeML distinguishes between non-recursive and recursive closures [30]. This distinction is also present in the *value* type. In this step, we will conflate variables with constants which necessitates a special treatment of recursive closures. Therefore we introduce a new predicate $\sigma \vdash t \downarrow v$ in Fig. 8 (in contrast to the previous $rs, \sigma \vdash t \downarrow v$). We examine the rules one by one:

CONST/VAR. Constant definition and variable values are both retrieved from the same environment σ . We have opted to keep the distinction between constants and variables in the *stern* type to avoid the introduction of another term type.

ABS. Identical to the previous evaluation semantics. Note that evaluation never creates recursive closures at run-time (only at compile-time, see Sect. 5.6). Anonymous functions, e.g. in the term $\langle \text{map } (\lambda x. x) \rangle$, are evaluated to non-recursive closures.

COMB. Identical to the previous evaluation semantics.

RECCOMB. Almost identical to the evaluation semantics. Additionally, for each function $(\text{name}, \text{cs}) \in \text{css}$, a new recursive closure $\text{Vrecabs css name } \sigma'$ is created and inserted into the environment. This ensures that after the first call to a recursive function, the function itself is present in the environment to be called recursively, without having to introduce coinductive environments.

CONSTR. Identical to the evaluation semantics.

Conflating Constants and Variables. By merging the rule set *rs* with the variable environment σ , it becomes necessary to discuss possible clashes. Previously, the syntactic distinction between *Svar* and *Sconst* meant that $\langle x \rangle$ and $\langle x \rangle$ are not ambiguous: all semantics up to the evaluation semantics clearly specify

where to look for the substitute. This is not the case in functional languages where functions and variables are not distinguished syntactically.

Instead, we rely on the fact that the initial rule set only defines constants. All variables are introduced by matching before β -reduction (that is, in the COMB and RECCOMB rules). The ABS rule does not change the environment. Hence it suffices to assume that variables in patterns must not overlap with constant names (see Sect. 5.1).

Correspondence Relation. Both constant definitions and values of variables are recorded in a single environment σ . This also applies to the environment contained in a closure. The correspondence relation thus needs to take a different sets of bindings in closures into account.

Hence, we define a relation \approx_v that is implicitly parametrized on the rule set rs and compares environments. We call it *right-conflating*, because in a correspondence $v \approx_v u$, any bound environment in u is thought to contain both variables and constants, whereas in v , any bound environment contains only variables.

Definition 7 (Right-conflating correspondence). We define \approx_v coinductively as follows:

$$\begin{array}{c}
 \frac{v_1 \approx_v u_1 \quad \cdots \quad v_n \approx_v u_n}{\text{Vconstr name } [v_1, \dots, v_n] \approx_v \text{Vconstr name } [u_1, \dots, u_n]} \\
 \frac{\forall x \in \text{frees cs. } \sigma_1 x \approx_v \sigma_2 x \quad \forall x \in \text{consts cs. } rs x \approx_v \sigma_2 x}{\text{Vabs cs } \sigma_1 \approx_v \text{Vabs cs } \sigma_2} \\
 \frac{\forall cs \in \text{range css. } \forall x \in \text{frees cs. } \sigma_1 x \approx_v \sigma_2 x \quad \forall cs \in \text{range css. } \forall x \in \text{consts cs. } \sigma_1 x \approx_v (\sigma_2 ++ \text{mk_rec_env css } \sigma_2) x}{\text{Vrecabs css name } \sigma_1 \approx_v \text{Vrecabs css name } \sigma_2}
 \end{array}$$

Consequently, \approx_v is not reflexive.

Correctness. The correctness lemma is straightforward to state:

Theorem 4 (Correctness). Let σ be an environment, t be a closed term and v a value such that $\sigma \vdash t \downarrow v$. If for all constants x occurring in t , $rs x \approx_v \sigma x$ holds, then there is an u such that $rs, [] \vdash t \downarrow u$ and $u \approx_v v$.

As usual, the rather technical proof proceeds via induction over the semantics (Fig. 8). It is important to note that the global clause set construction (Sect. 5.6) satisfies the preconditions of this theorem:

Lemma 4. If $name$ is the name of a constant in rs , then

$$\text{Vrecabs global_css name } [] \approx_v \text{Vrecabs global_css name } []$$

Because \approx_v is defined coinductively, the proof of this precondition proceeds by coinduction.

5.8 CakeML

CakeML is a verified implementation of a subset of Standard ML [24,40]. It comprises a parser, type checker, formal semantics and backend for machine code. The semantics has been formalized in Lem [29], which allows export to Isabelle theories.

Our compiler targets CakeML’s abstract syntax tree. However, we do not make use of certain CakeML features; notably mutable cells, modules, and literals. We have derived a smaller, executable version of the original CakeML semantics, called *CupCakeML*, together with an equivalence proof. The correctness proof of the last compiler phase establishes a correspondence between Cup-CakeML and the final semantics of our compiler pipeline.

For the correctness proof of the CakeML compiler, its authors have extracted the Lem specification into HOL4 theories [1]. In our work, we directly target CakeML abstract syntax trees (thereby bypassing the parser) and use its big-step semantics, which we have extracted into Isabelle.²

Conversion from *stern* to *exp*. After the series of translations described in the earlier sections, our terms are syntactically close to CakeML’s terms (*Cake.exp*). The only remaining differences are outlined below:

- CakeML does not combine abstraction and pattern matching. For that reason, we have to translate $\Lambda [p_1 \Rightarrow t_1, \dots]$ into $\Lambda x. \text{case } x \text{ of } p_1 \Rightarrow t_1 \mid \dots$, where x is a fresh variable name. We reuse the **fresh** monad to obtain a bound variable name. Note that it is not necessary to thread through already created variable names, only existing names. The reason is simple: a generated variable is bound and then immediately used in the body. Shadowing it somewhere in the body is not problematic.
- CakeML has two distinct syntactic categories for identifiers (that can represent variables or functions) and data constructors. Our term types however have two distinct syntactic categories for constants (that can represent functions or data constructors) and variables. The necessary prerequisites to deal with this are already present in the ML-style evaluation semantics (Sect. 5.7) which conflates constants and variables, but has a dedicated **CONSTR** rule for data constructors.

Types. During embedding (Sect. 3), all type information is erased. Yet, CakeML performs some limited form of type checking at run-time: constructing and matching data must always be fully applied. That is, data constructors must always occur with all arguments supplied on right-hand and left-hand sides.

Fully applied constructors in terms can be easily guaranteed by simple pre-processing. For patterns however, this must be ensured throughout the compilation pipeline; it is (like other syntactic constraints) another side condition imposed on the rule set (Sect. 5.1).

² Based on a repository snapshot from March 27, 2017 (0c48672).

The shape of datatypes and constructors is managed in CakeML’s environment. This particular piece of information is allowed to vary in closures, since ML supports local type definitions. Tracking this would greatly complicate our proofs. Hence, we fix a global set of constructors and enforce that all values use exactly that one.

Correspondence Relation. We define two different correspondence relations: One for values and one for expressions.

Definition 8 (Expression correspondence)

$$\begin{array}{c}
 \text{VAR} \frac{}{\text{rel_e} (\text{Svar } n) (\text{Cake.Var } n)} \quad \text{CONST} \frac{n \notin \text{constructors}}{\text{rel_e} (\text{Sconst } n) (\text{Cake.Var } n)} \\
 \\
 \text{CONSTR} \frac{n \in \text{constructors} \quad \text{rel_e } t_1 \ u_1 \quad \dots}{\text{rel_e} (\text{Sconst name } \$ t_1 \$ \dots \$ t_n) (\text{Cake.Con} (\text{Some} (\text{Cake.Short name}) [u_1, \dots, u_n]))} \\
 \\
 \text{APP} \frac{\text{rel_e } t_1 \ u_1 \quad \text{rel_e } t_2 \ u_2}{\text{rel_e } t_1 \$ t_2 \ \text{Cake.App } \text{Cake.Opapp } [u_1, u_2]} \\
 \quad n \notin \text{ids } (\Lambda [p_1 \Rightarrow t_1, \dots]) \cup \text{constructors} \\
 \quad q_1 = \text{mk_ml_pat } p_1 \quad \text{rel_e } t_1 \ u_1 \quad \dots \\
 \text{FUN} \frac{}{\text{rel_e} (\Lambda [p_1 \Rightarrow t_1, \dots]) (\text{Cake.Fun } n (\text{Cake.Mat} (\text{Cake.Var } n)) [q_1 \Rightarrow u_1, \dots])} \\
 \\
 \text{MAT} \frac{\text{rel_e } t \ u \quad q_1 = \text{mk_ml_pat } p_1 \quad \text{rel_e } t_1 \ u_1 \quad \dots}{\text{rel_e} (\Lambda [p_1 \Rightarrow t_1, \dots] \$ t) (\text{Cake.Mat } u [q_1 \Rightarrow u_1, \dots])}
 \end{array}$$

We will explain each of the rules briefly here.

VAR. Variables are directly related by identical name.

CONST. As described earlier, constructors are treated specially in CakeML. In order to not confuse functions or variables with data constructors themselves, we require that the constant name is not a constructor.

CONSTR. Constructors are directly related by identical name, and recursively related arguments.

APP. CakeML does not just support general function application but also unary and binary operators. In fact, function application is the binary operator **Opapp**. We never generate other operators. Hence the correspondence is restricted to **Opapp**.

FUN/MAT. Observe the symmetry between these two cases: In our term language, matching and abstraction are combined, which is not the case in CakeML. This means we relate a case abstraction to a CakeML function containing a match, and a case abstraction applied to a value to just a CakeML match.

There is no separate relation for patterns, because their translation is simple.

The value correspondence (**rel_v**) is structurally simpler. In the case of constructor values (**Vconstr** and **Cake.Conv**), arguments are compared recursively. Closures and recursive closures are compared extensionally, i.e. only bindings that occur in the body are checked recursively for correspondence.

Correctness. We use the same trick as in Sect. 5.6 to obtain a suitable environment for CakeML evaluation based on the rule set rs .

Theorem 5 (Correctness). *If the compiled expression `term_to_cake t` terminates with a value u in the CakeML semantics, there is a value v such that $\text{rel_v } v \ u$ and $rs \vdash t \downarrow v$.*

6 Composition

The complete compiler pipeline consists of multiple phases. Correctness is justified for each phase between intermediate semantics and correspondence relations, most of which are rather technical. Whereas the compiler may be complex and impenetrable, the trustworthiness of the constructions hinges on the obviousness of those correspondence relations.

Fortunately, under the assumption that terms to be evaluated and the resulting values do not contain abstractions – or closures, respectively – all of the correspondence relations collapse to simple structural equality: two terms are related if and only if one can be converted to the other by consistent renaming of term constructors.

The actual compiler can be characterized with two functions. Firstly, the translation of term to `Cake.exp` is a simple composition of each term translation function:

definition `term_to_cake` :: `term` \Rightarrow `Cake.exp` **where**
`term_to_cake` = `term_to_cake` \circ `pterm_to_sterm` \circ `nterm_to_pterm` \circ `term_to_nterm`

Secondly, the function that translates function definitions by composing the phases as outlined in Fig. 2, including iterated application of pattern elimination:

definition `compile` :: `(term \times term)` `fset` \Rightarrow `Cake.dec` **where**
`compile` = `Cake.Dletrec` \circ `compile_srules_to_cake` \circ `compile_prules_to_srules` \circ
`compile_irules_to_srules` \circ `compile_irules_iter` \circ `compile_crules_to_irules` \circ
`consts_of` \circ `compile_rules_to_nrules`

Each function `compile_*` corresponds to one compiler phase; the remaining functions are trivial. This produces a CakeML top-level declaration. We prove that evaluating this declaration in the top-level semantics (`evaluate_prog`) results in an environment `cake_sem_env`. But `cake_sem_env` can also be computed via another instance of the global clause set trick (Sect. 5.6).

Equipped with these functions, we can state the final correctness theorem:

theorem `compiled_correct`:

(If CakeML evaluation of a term succeeds ... *)*
assumes `evaluate False cake_sem_env s (term_to_cake t) (s', Rval ml_v)`
(... producing a constructor term without closures ... *)*
assumes `cake_abstraction.free ml_v`
(... and some syntactic properties of the involved terms hold ... *)*
assumes `closed t` **and** \neg `shadows_consts (heads rs \cup constructors) t` **and**
`welldefined (heads rs \cup constructors) t` **and** `wellformed t`
(... then this evaluation can be reproduced in the term-rewriting semantics *)*
shows $rs \vdash t \rightarrow^* \text{cake_to_term } ml_v$

	datatype 'a dict_add = Dict_add ('a \Rightarrow 'a \Rightarrow 'a)
class add = fixes plus :: 'a \Rightarrow 'a \Rightarrow 'a	fun cert_add :: ('a::add) dict_add \Rightarrow bool where cert_add (Dict_add pls) = (pls = plus)
definition f :: ('a::add) \Rightarrow 'a where f x = plus x x	fun f' :: 'a dict_add \Rightarrow 'a \Rightarrow 'a where f' (Dict_add pls) x = pls x x
(a) Source program	lemma f'_eq: cert_add dict \rightarrow f' dict = f <proof>
	(b) Result of translation

Fig. 9. Dictionary construction in Isabelle

This theorem directly relates the evaluation of a term t in the full CakeML (including mutability and exceptions) to the evaluation in the initial higher-order term rewriting semantics. The evaluation of t happens using the environment produced from the initial rule set. Hence, the theorem can be interpreted as the correctness of the pseudo-ML expression **let rec** rs **in** t .

Observe that in the assumption, the conversion goes from our terms to CakeML expressions, whereas in the conclusion, the conversion goes the opposite direction.

7 Dictionary Construction

Isabelle's type system supports *type classes* (or simply *classes*) [18, 44] whereas CakeML does not. In order to not complicate the correctness proofs, type classes are not supported by our embedded term language either. Instead, we eliminate classes and instances by a dictionary construction [19] before embedding into the term language. Haftmann and Nipkow give a pen-and-paper correctness proof of this construction [17, Sect. 4.1]. We augmented the dictionary construction with the generation of a certificate theorem that shows the equivalence of the two versions of a function, with type classes and with dictionaries. This section briefly explains our dictionary construction.

Figure 9 shows a simple example of a dictionary construction. Type variables may carry *class constraints* (e.g. $\alpha :: \text{add}$). The basic idea is that classes become *dictionary*s containing the functions of that class; class instances become dictionary definitions. Dictionaries are realized as datatypes. Class constraints become additional dictionary parameters for that class. In the example, class **add** becomes **dict_add**; function f is translated into f' which takes an additional parameter of type **dict_add**. In reality our tool does not produce the Isabelle source code shown in Fig. 9b but performs the constructions internally. The correctness lemma f'_{eq} is proved automatically. Its precondition expresses that the dictionary must contain exactly the function(s) of class **add**. For any monomorphic instance, the precondition can be proved outright based on the certificate theorems proved for each class instance as explained next.

Not shown in the example is the translation of class instances. The basic form of a class instance in Isabelle is $\tau::(c_1, \dots, c_n) \ c$ where τ is an n -ary type constructor. It corresponds to Haskell's $(c_1 \ \alpha_1, \dots, c_n \ \alpha_n) \Rightarrow c \ (\tau \ \alpha_1 \dots \alpha_n)$ and is translated into a function $\text{inst_c_}\tau::\alpha_1 \ \text{dict_}c_1 \Rightarrow \dots \Rightarrow \alpha_n \ \text{dict_}c_n \Rightarrow (\alpha_1, \dots, \alpha_n) \ \tau \ \text{dict_}c$ and the following certificate theorem is proved:

$$\text{cert_}c_1 \ \text{dict}_1 \rightarrow \dots \rightarrow \text{cert_}c_n \ \text{dict}_n \rightarrow \text{cert_}c \ (\text{inst_c_}\tau \ \text{dict}_1 \ \dots \ \text{dict}_n)$$

For a more detailed explanation of how the dictionary construction works, we refer to the corresponding entry in the Archive of Formal Proofs [21].

8 Evaluation

We have tried out our compiler on examples from existing Isabelle formalizations. This includes an implementation of Huffman encoding, lists and sorting, string functions [39], and various data structures from Okasaki's book [34], including binary search trees, pairing heaps, and leftist heaps. These definitions can be processed with slight modifications: functions need to be totalized (see the end of Sect. 3). However, parts of the tactics required for deep embedding proofs (Sect. 3) are too slow on some functions and hence still need to be optimized.

9 Conclusion

For this paper we have concentrated on the compiler from Isabelle/HOL to CakeML abstract syntax trees. Partial correctness is proved w.r.t. the big-step semantics of CakeML. In the next step we will link our work with the compiler from CakeML to machine code. Tan *et al.* [40, Sect. 10] prove a correctness theorem that relates their semantics with the execution of the compiled machine code. In that paper, they use a newer iteration of the CakeML semantics (functional big-step [35]) than we do here. Both semantics are still present in the CakeML source repository, together with an equivalence proof. Another important step consists of targeting CakeML's native types, e.g. integer numbers and characters.

Evaluation of our compiled programs is already possible via Isabelle's predicate compiler [5], which allows us to turn CakeML's big-step semantics into an executable function. We have used this execution mechanism to establish for sample programs that they terminate successfully. We also plan to prove that our compiled programs terminate, i.e. total correctness.

The total size of this formalization, excluding theories extracted from Lem, is currently approximately 20000 lines of proof text (90 %) and ML code (10 %). The ML code itself produces relatively simple theorems, which means that there are less opportunities for it to go wrong. This constitutes an improvement over certifying approaches that prove complicated properties in ML.

References

1. The HOL System Description (2014). <https://hol-theorem-prover.org/>
2. Anand, A., Appel, A.W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Bélanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: a verified compiler for Coq. In: CoqPL 2017: Third International Workshop on Coq for Programming Languages (2017)
3. Augustsson, L.: Compiling pattern matching. In: Jouannaud, J.P. (ed.) *Functional Programming Languages and Computer Architecture*, pp. 368–381. Springer, Heidelberg (1985)
4. Benton, N., Hur, C.: Biorthogonality, step-indexing and compiler correctness. In: Hutton, G., Tolmach, A.P. (eds.) *ICFP 2009*, pp. 97–108. ACM (2009)
5. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning inductive into equational specifications. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs 2009*. LNCS, vol. 5674, pp. 131–146. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_11
6. Berghofer, S., Nipkow, T.: Executing higher order logic. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R., Pollack, R. (eds.) *TYPES 2000*. LNCS, vol. 2277, pp. 24–40. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45842-5_2
7. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co)datatypes for Isabelle/HOL. In: Klein, G., Gamboa, R. (eds.) *ITP 2014*. LNCS, vol. 8558, pp. 93–110. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_7
8. Boespflug, M., Dénès, M., Grégoire, B.: Full reduction at full throttle. In: Jouannaud, J.-P., Shao, Z. (eds.) *CPP 2011*. LNCS, vol. 7086, pp. 362–377. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_26
9. Boyer, R.S., Strother Moore, J.: Single-threaded objects in ACL2. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) *PADL 2002*. LNCS, vol. 2257, pp. 9–27. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45587-6_3
10. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indag. Math. (Proceedings)* **75**(5), 381–392 (1972)
11. Chlipala, A.: A verified compiler for an impure functional language. In: Hermenegildo, M.V., Palsberg, J. (eds.) *POPL 2010*, pp. 93–106. ACM (2010)
12. Crow, J., Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001
13. Fallenstein, B., Kumar, R.: Proof-producing reflection for HOL. In: Urban, C., Zhang, X. (eds.) *ITP 2015*. LNCS, vol. 9236, pp. 170–186. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_11
14. Flatau, A.D.: A verified implementation of an applicative language with dynamic storage allocation. Ph.D. thesis, University of Texas at Austin (1992)
15. Forster, Y., Kunze, F.: Verified extraction from coq to a lambda-calculus. In: *The 8th Coq Workshop* (2016)
16. Greve, D.A., Kaufmann, M., Manolios, P., Moore, J.S., Ray, S., Ruiz-Reina, J., Summers, R., Vroon, D., Wilding, M.: Efficient execution in an automated reasoning environment. *J. Funct. Program.* **18**(1), 15–46 (2008)
17. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) *FLOPS 2010*. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12251-4_9

18. Haftmann, F., Wenzel, M.: Constructive type classes in Isabelle. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 160–174. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74464-1_11
19. Hall, C.V., Hammond, K., Jones, S.L.P., Wadler, P.L.: Type classes in Haskell. *ACM Trans. Program. Lang. Syst.* **18**(2), 109–138 (1996)
20. Hermida, C., Reddy, U.S., Robinson, E.P.: Logical relations and parametricity - a Reynolds programme for category theory and programming languages. *Electron. Notes Theoret. Comput. Sci.* **303**, 149–180 (2014)
21. Hupel, L.: Dictionary construction. Archive of Formal Proofs, May 2017. http://isa-afp.org/entries/Dict_Construction.html, Formal proof development
22. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. *J. Autom. Reason.* **44**(4), 303–336 (2010)
23. Krauss, A., Schropp, A.: A mechanized translation from higher-order logic to set theory. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 323–338. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_23
24. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: POPL 2014, pp. 179–191. ACM (2014)
25. Landin, P.J.: The mechanical evaluation of expressions. *Comput. J.* **6**(4), 308–320 (1964)
26. Leroy, X.: Formal verification of a realistic compiler. *Commun. ACM* **52**(7), 107–115 (2009). <http://doi.acm.org/10.1145/1538788.1538814>
27. Letouzey, P.: A new extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39185-1_12
28. Milner, R., Tofte, M., Harper, R., MacQueen, D.: *The Definition of Standard ML (Revised)*. MIT Press, Cambridge (1997)
29. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: reusable engineering of real-world semantics. In: ICFP 2014, pp. 175–188. ACM (2014)
30. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. *JFP* **24**(2–3), 284–315 (2014)
31. Neis, G., Hur, C.K., Kaiser, J.O., McLaughlin, C., Dreyer, D., Vafeiadis, V.: Pil-sner: a compositionally verified compiler for a higher-order imperative language. In: ICFP 2015, pp. 166–178. ACM, New York (2015)
32. Nipkow, T., Klein, G.: *Concrete Semantics*. Springer, Cham (2014). <https://doi.org/10.1007/978-3-319-10542-0>
33. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>. 218p.
34. Okasaki, C.: *Purely Functional Data Structures*. Cambridge University Press, Cambridge (1999)
35. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 589–615. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_23
36. Peyton Jones, S.L.: *The Implementation of Functional Programming Languages*. Prentice-Hall Inc., Upper Saddle River (1987)
37. Shankar, N.: Static analysis for safe destructive updates in a functional language. In: Pettorossi, A. (ed.) LOPSTR 2001. LNCS, vol. 2372, pp. 1–24. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45607-4_1
38. Slind, K.: Reasoning about terminating functional programs. Ph.D. thesis, Technische Universität München (1999)

39. Sternagel, C., Thiemann, R.: Haskell’s show class in Isabelle/HOL. Archive of Formal Proofs, July 2014. <http://isa-afp.org/entries/Show.html>, Formal proof development
40. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: A new verified compiler backend for CakeML. In: Proceedings of 21st ACM SIGPLAN International Conference on Functional Programming - ICFP 2016. Association for Computing Machinery (ACM) (2016)
41. Turner, D.A.: Some history of functional programming languages. In: Loidl, H.-W., Peña, R. (eds.) TFP 2012. LNCS, vol. 7829, pp. 1–20. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40447-4_1
42. Urban, C.: Nominal techniques in Isabelle/HOL. J. Autom. Reason. **40**(4), 327–356 (2008). <https://doi.org/10.1007/s10817-008-9097-2>
43. Urban, C., Berghofer, S., Kaliszyk, C.: Nominal 2. Archive of Formal Proofs, February 2013. Formal proof development: <http://isa-afp.org/entries/Nominal2.shtml>
44. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E.L., Felty, A. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 307–322. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0028402>



Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Compositional Verification of Compiler Optimisations on Relaxed Memory

Mike Dodds¹ , Mark Batty², and Alexey Gotsman³ 

¹ Galois Inc., Portland, Oregon, USA
miked@galois.com

² University of Kent, Canterbury, UK
M.J.Batty@kent.ac.uk

³ IMDEA Software Institute, Madrid, Spain
alexey.gotsman@imdea.org

Abstract. A valid compiler optimisation transforms a block in a program without introducing new observable behaviours to the program as a whole. Deciding which optimisations are valid can be difficult, and depends closely on the semantic model of the programming language. Axiomatic relaxed models, such as C++11, present particular challenges for determining validity, because such models allow subtle effects of a block transformation to be observed by the rest of the program. In this paper we present a denotational theory that captures optimisation validity on an axiomatic model corresponding to a fragment of C++11. Our theory allows verifying an optimisation compositionally, by considering only the block it transforms instead of the whole program. Using this property, we realise the theory in the first push-button tool that can verify real-world optimisations under an axiomatic memory model.

1 Introduction

Context and Objectives. Any program defines a collection of observable behaviours: a sorting algorithm maps unsorted to sorted sequences, and a paint program responds to mouse clicks by updating a rendering. It is often desirable to transform a program without introducing new observable behaviours – for example, in a compiler optimisation or programmer refactoring. Such transformations are called *observational refinements*, and they ensure that properties of the original program will carry over to the transformed version. It is also desirable for transformations to be *compositional*, meaning that they can be applied to a block of code irrespective of the surrounding program context. Compositional transformations are particularly useful for automated systems such as compilers, where they are known as *peephole optimisations*.

The semantics of the language is highly significant in determining which transformations are valid, because it determines the ways that a block of code being transformed can interact with its context and thereby affect the observable behaviour of the whole program. Our work applies to a relaxed memory concurrent setting. Thus, the context of a code-block includes both code sequentially

before and after the block, and code that runs in parallel. Relaxed memory means that different threads can observe different, apparently contradictory orders of events – such behaviour is permitted by programming languages to reflect CPU-level relaxations and to allow compiler optimisations.

We focus on *axiomatic* memory models of the type used in C/C++ and Java. In axiomatic models, program executions are represented by structures of memory actions and relations on them, and program semantics is defined by a set of axioms constraining these structures. Reasoning about the correctness of program transformations on such memory models is very challenging, and indeed, compiler optimisations have been repeatedly shown unsound with respect to models they were intended to support [23,25]. The fundamental difficulty is that axiomatic models are defined in a global, non-compositional way, making it very challenging to reason compositionally about the single code-block being transformed.

Approach. Suppose we have a code-block B , embedded into an unknown program context. We define a *denotation* for the code-block which summarises its behaviour in a restricted representative context. The denotation consists of a set of *histories* which track interactions across the boundary between the code-block and its context, but abstract from internal structure of the code-block. We can then validate a transformation from code-block B to B' by comparing their denotations. This approach is compositional: it requires reasoning only about the code-blocks and representative contexts; the validity of the transformation in an arbitrary context will follow. It is also *fully abstract*, meaning that it can verify any valid transformation: considering only representative contexts and histories does not lose generality.

We also define a variant of our denotation that is *finite* at the cost of losing full abstraction. We achieve this by further restricting the form of contexts one needs to consider in exchange for tracking more information in histories. For example, it is unnecessary to consider executions where two context operations read from the same write.

Using this finite denotation, we implement a prototype verification tool, Stellite. Our tool converts an input transformation into a model in the Alloy language [12], and then checks that the transformation is valid using the Alloy* solver [18]. Our tool can prove or disprove a range of introduction, elimination, and exchange compiler optimisations. Many of these were verified by hand in previous work; our tool verifies them automatically.

Contributions. Our contribution is twofold. First, we define the first fully abstract denotational semantics for an axiomatic relaxed model. Previous proposals in this space targeted either non-relaxed sequential consistency [6] or much more restrictive operational relaxed models [7,13,21]. Second, we show it is feasible to automatically verify relaxed-memory program transformations. Previous techniques required laborious proofs by hand or in a proof assistant [23–27]. Our target model is derived from the C/C++ 2011 standard [22]. However, our aim is not to handle C/C++ per se (especially as the model is in flux in several respects; see Sect. 3.7). Rather we target the simplest axiomatic model rich enough to demonstrate our approach.

2 Observation and Transformation

Observational Refinement. The notion of *observation* is crucial when determining how different programs are related. For example, observations might be I/O behaviour or writes to special variables. Given program executions X_1 and X_2 , we write $X_1 \preceq_{\text{ex}} X_2$ if the observations in X_1 are replicated in X_2 (defined formally in the following). Lifting this notion, a program P_1 *observationally refines* another P_2 if every observable behaviour of one could also occur with the other – we write this $P_1 \preceq_{\text{pr}} P_2$. More formally, let $\llbracket - \rrbracket$ be the map from programs to sets of executions. Then we define \preceq_{pr} as:

$$P_1 \preceq_{\text{pr}} P_2 \iff \forall X_1 \in \llbracket P_1 \rrbracket. \exists X_2 \in \llbracket P_2 \rrbracket. X_1 \preceq_{\text{ex}} X_2 \quad (1)$$

Compositional Transformation. Many common program transformations are *compositional*: they modify a sequential fragment of the program without examining the rest of the program. We call the former the *code-block* and the latter its *context*. Contexts can include sequential code before and after the block, and concurrent code that runs in parallel with it. Code-blocks are sequential, i.e. they do not feature internal concurrency. A context C and code-block B can be composed to give a whole program $C(B)$.

A transformation $B_2 \rightsquigarrow B_1$ replaces some instance of the code-block B_2 with B_1 . To validate such a transformation, we must establish whether *every* whole program containing B_1 observationally refines the same program with B_2 substituted. If this holds, we say that B_1 observationally refines B_2 , written $B_1 \preceq_{\text{bl}} B_2$, defined by lifting \preceq_{pr} as follows:

$$B_1 \preceq_{\text{bl}} B_2 \iff \forall C. C(B_1) \preceq_{\text{pr}} C(B_2) \quad (2)$$

If $B_1 \preceq_{\text{bl}} B_2$ holds, then the compiler can replace block B_2 with block B_1 irrespective of the whole program, i.e. $B_2 \rightsquigarrow B_1$ is a valid transformation. Thus, deciding $B_1 \preceq_{\text{bl}} B_2$ is the core problem in validating compositional transformations.

The language semantics is highly significant in determining observational refinement. For example, the code blocks $B_1: \text{store}(\mathbf{x}, 5)$ and $B_2: \text{store}(\mathbf{x}, 2); \text{store}(\mathbf{x}, 5)$ are observationally equivalent in a sequential setting. However, in a concurrent setting the intermediate state, $\mathbf{x} = 2$, can be observed in B_2 but not B_1 , meaning the code-blocks are no longer observationally equivalent. In a relaxed-memory setting there is no global state seen by all threads, which further complicates the notion of observation.

Compositional Verification. To establish $B_1 \preceq_{\text{bl}} B_2$, it is difficult to examine all possible syntactic contexts. Our approach is to construct a *denotation* for each code-block – a simplified, ideally finite, summary of possible interactions between the block and its context. We then define a *refinement relation* on denotations and use it to establish observational refinement. We write $B_1 \sqsubseteq B_2$ when the denotation of B_1 refines B_2 .

Refinement on denotations should be *adequate*, i.e., it should validly approximate observational refinement: $B_1 \sqsubseteq B_2 \implies B_1 \preceq_{\text{bl}} B_2$. Hence, if $B_1 \sqsubseteq B_2$, then $B_2 \rightsquigarrow B_1$ is a valid transformation. It is also desirable for the denotation to be *fully abstract*: $B_1 \preceq_{\text{bl}} B_2 \implies B_1 \sqsubseteq B_2$. This means any valid transformation can be verified by comparing denotations. Below we define several versions of \sqsubseteq with different properties.

3 Target Language and Core Memory Model

Our language’s memory model is derived from the C/C++ 2011 standard (henceforth ‘C11’), as formalised by [5, 22]. However, we simplify our model in several ways; see the end of section for details. In C11 terms, our model covers release-acquire and non-atomic operations, and sequentially consistent fences. To simplify the presentation, at first we omit non-atomics, and extend our approach to cover them in Sect. 7. Thus, all operations in this section correspond to C11’s release-acquire.

3.1 Relaxed Memory Primer

In a sequentially consistent concurrent system, there is a total temporal order on loads and stores, and loads take the value of the most recent store; in particular, they cannot read overwritten values, or values written in the future. A *relaxed* (or *weak*) memory model weakens this total order, allowing behaviours forbidden under sequential consistency. Two standard examples of relaxed behaviour are *store buffering* (SB) and *message passing* (MP), shown in Fig. 1.

<pre> store(x,0); store(y,0); store(x,1); store(y,1); v1 := load(y); v2 := load(x); </pre>	<pre> store(f,0); store(x,0); store(x,1); b := load(f); store(f,1); if (b == 1) r := load(x); </pre>
--	---

Fig. 1. *Left:* store-buffering (SB) example. *Right:* message-passing (MP) example.

In most relaxed models $v1 = v2 = 0$ is a possible post-state for SB. This cannot occur on a sequentially consistent system: if $v1 = 0$, then `store(y,1)` must be ordered after the load of `y`, which would order `store(x,1)` before the load of `x`, forcing it to assign $v2 = 1$. In some relaxed models, $b = 1 \wedge r = 0$ is a possible post-state for MP. This is undesirable if, for example, `x` is a complex data-structure and `f` is a flag indicating it has been safely created.

3.2 Language Syntax

Programs in the language we consider manipulate *thread-local variables* $l, l_1, l_2 \dots \in \text{LVar}$ and *global variables* $x, y, \dots \in \text{GVar}$, coming from disjoint sets

LVar and **GVar**. Each variable stores a value from a finite set **Val** and is initialised to $0 \in \text{Val}$. Constants are encoded by special read-only thread-local variables. We assume that each thread uses the same set of thread-local variable names **LVar**. The syntax of the programming language is as follows:

$$\begin{aligned}
C &:: l := E \mid \text{store}(x, l) \mid l := \text{load}(x) \mid l := \text{LL}(x) \mid l' := \text{SC}(x, l) \mid \text{fence} \mid \\
&\quad C_1 \parallel C_2 \mid C_1; C_2 \mid \text{if } (l) \{C_1\} \text{ else } \{C_2\} \mid \{-\} \\
E &:: l \mid l_1 = l_2 \mid l_1 \neq l_2 \mid \dots
\end{aligned}$$

Many of the constructs are standard. $\text{LL}(x)$ and $\text{SC}(x, l)$ are *load-link* and *store-conditional*, which are basic concurrency operations available on many platforms (e.g., Power and ARM). A load-link $\text{LL}(x)$ behaves as a standard load of global variable x . However, if it is followed by a store-conditional $\text{SC}(x, l)$, the store fails and returns false if there are intervening writes to the same location. Otherwise the store-conditional writes l and returns true. The **fence** command is a *sequentially consistent fence*: interleaving such fences between all statements in a program guarantees sequentially consistent behaviour. We do not include *compare-and-swap* (CAS) command in our language because LL-SC is more general [2]. Hardware-level LL-SC is used to implement C11 CAS on Power and ARM. Our language does not include loops because our model in this paper does not include infinite computations (see Sect. 3.7 for discussion). As a result, loops can be represented by their finite unrollings. Our **load** commands write into a local variable. In examples, we sometimes use ‘bare’ loads without a variable write.

The construct $\{-\}$ represents a block-shaped hole in the program. To simplify our presentation, we assume that at most one hole appears in the program. Transformations that apply to multiple blocks at once can be simulated by using the fact our approach is compositional: transformations can be applied in sequence using different divisions of the program into code-block and context.

The set **Prog** of *whole programs* consists of programs without holes, while the set **Contx** of *contexts* consists of programs with a hole. The set **Block** of *code-blocks* are whole programs without parallel composition. We often write $P \in \text{Prog}$ for a whole program, $B \in \text{Block}$ for a code-block, and $C \in \text{Contx}$ for a context. Given a context C and a code-block B , the composition $C(B)$ is C with its hole syntactically replaced by B . For example:

$$\begin{aligned}
C: \text{load}(x); \{-\}; \text{store}(y, 11), \quad B: \text{store}(x, 2) \\
\longrightarrow C(B): \text{load}(x); \text{store}(x, 2); \text{store}(y, 11)
\end{aligned}$$

We restrict **Prog**, **Contx** and **Block** to ensure LL-SC pairs are matched correctly. Each SC must be preceded in program order by a LL to the same location. Other types of operations may occur between the LL and SC, but intervening SC operations are forbidden. For example, the program $\text{LL}(x); \text{SC}(x, v1); \text{SC}(x, v2);$ is forbidden. We also forbid LL-SC pairs from spanning parallel compositions, and from spanning the block/context boundary.

3.3 Memory Model Structure

The semantics of a whole program P is given by a set $\llbracket P \rrbracket$ of *executions*, which consist of *actions*, representing memory events on global variables, and several relations on these. Actions are tuples in the set $\text{Action} \triangleq \text{ActID} \times \text{Kind} \times \text{Option}(\text{GVar}) \times \text{Val}^*$. In an action $(a, k, z, b) \in \text{Action}$: $a \in \text{ActID}$ is the unique action identifier; $k \in \text{Kind}$ is the kind of action – we use *load*, *store*, *LL*, *SC*, and the failed variant SC_f in the semantics, and will introduce further kinds as needed; $z \in \text{Option}(\text{GVar})$ is an option type consisting of either a single global variable $\text{Just}(x)$ or *None*; and $b \in \text{Val}^*$ is the vector of values (actions with multiple values are used in Sect. 4).

Given an action v , we use $\mathbf{gvar}(v)$ and $\mathbf{val}(v)$ as selectors for the different fields. We often write actions so as to elide action identifiers and the option type. For example, $\text{load}(x, 3)$ stands for $\exists i. (i, \text{load}, \text{Just}(x), [3])$. We also sometimes elide values. We call *load* and *LL* actions *reads*, and *store* and successful *SC* actions *writes*. Given a set of actions \mathcal{A} , we write, e.g., $\text{reads}(\mathcal{A})$ to identify read actions in \mathcal{A} . Below, we range over all actions by u, v ; read actions by r ; write actions by w ; and *LL*, *SC* actions by ll and sc respectively.

$$\begin{aligned}
 \langle l := \text{load}(x), \sigma \rangle &\triangleq \{(\{\text{load}(x, a)\}, \emptyset, \sigma[l \mapsto a]) \mid a \in \text{Val}\} \\
 \langle \text{store}(x, l), \sigma \rangle &\triangleq \{(\{\text{store}(x, a)\}, \emptyset, \sigma) \mid \sigma(l) = a\} \\
 \langle C_1; C_2, \sigma \rangle &\triangleq \{(\mathcal{A}_1 \cup \mathcal{A}_2, \mathbf{sb}_1 \cup \mathbf{sb}_2 \cup (\mathcal{A}_1 \times \mathcal{A}_2), \sigma_2) \mid \\
 &\quad (\mathcal{A}_1, \mathbf{sb}_1, \sigma_1) \in \langle C_1, \sigma \rangle \wedge (\mathcal{A}_2, \mathbf{sb}_2, \sigma_2) \in \langle C_2, \sigma_1 \rangle\} \\
 \langle \text{fence}, \sigma \rangle &\triangleq \{(\{ll, sc\}, \{ll, sc\}, \sigma) \mid ll = \text{LL}(\text{fen}, 0) \wedge sc = \text{SC}(\text{fen}, 0)\}
 \end{aligned}$$

Fig. 2. Selected clauses of the thread-local semantics. The full semantics is given in [10, Sect. A]. We write $\mathcal{A}_1 \cup \mathcal{A}_2$ for a union that is defined only when actions in \mathcal{A}_1 and \mathcal{A}_2 use disjoint sets of identifiers. We omit identifiers from actions to avoid clutter.

The semantics of a program $P \in \text{Prog}$ is defined in two stages. First, a *thread-local semantics* of P produces a set $\langle P \rangle$ of *pre-executions* $(\mathcal{A}, \mathbf{sb}) \in \text{PreExec}$. A pre-execution contains a finite set of memory actions $\mathcal{A} \subseteq \text{Action}$ that could be produced by the program. It has a transitive and irreflexive *sequence-before* relation $\mathbf{sb} \subseteq \mathcal{A} \times \mathcal{A}$, which defines the sequential order imposed by the program syntax.

For example two sequential statements in the same thread produce actions ordered in \mathbf{sb} . The thread-local semantics takes into account control flow in P 's threads and operations on local variables. However, it does not constrain the behaviour of global variables: the values threads read from them are chosen arbitrarily. This is addressed by extending pre-executions with extra relations, and filtering the resulting *executions* using *validity axioms*.

3.4 Thread-Local Semantics

The thread-local semantics is defined formally in Fig. 2. The semantics of a program $P \in \text{Prog}$ is defined using function $\langle -, - \rangle : \text{Prog} \times \text{VMap} \rightarrow \mathcal{P}(\text{PreExec} \times \text{VMap})$. The values of local variables are tracked by a map $\sigma \in \text{VMap} \stackrel{\Delta}{=} \text{LVar} \rightarrow \text{Val}$. Given a program and an input local variable map, the function produces a set of pre-executions paired with an output variable map, representing the values of local variables at the end of the execution. Let σ_0 map every local variable to 0. Then $\langle P \rangle$, the thread-local semantics of a program P , is defined as

$$\langle P \rangle \stackrel{\Delta}{=} \{(\mathcal{A}, \text{sb}) \mid \exists \sigma'. (\mathcal{A}, \text{sb}, \sigma') \in \langle P, \sigma_0 \rangle\}$$

The significant property of the thread-local semantics is that it does not restrict the behaviour of global variables. For this reason, note that the clause for **load** in Fig. 2 leaves the value a unrestricted. We follow [16] in encoding the **fence** command by a successful LL-SC pair to a distinguished variable $\text{fen} \in \text{GVar}$ that is not otherwise read or written.

3.5 Execution Structure and Validity Axioms

The semantics of a program P is a set $\llbracket P \rrbracket$ of *executions* $X = (\mathcal{A}, \text{sb}, \text{at}, \text{rf}, \text{mo}, \text{hb}) \in \text{Exec}$, where (\mathcal{A}, sb) is a pre-execution and $\text{at}, \text{rf}, \text{mo}, \text{hb} \subseteq \mathcal{A} \times \mathcal{A}$. Given an execution X we sometimes write $\mathcal{A}(X), \text{sb}(X), \dots$ as selectors for the appropriate set or relation. The relations have the following purposes.

- *Reads-from* (**rf**) is an injective map from reads to writes at the same location of the same value. A read and a write actions are related $w \xrightarrow{\text{rf}} r$ if r takes its value from w .
- *Modification order* (**mo**) is an irreflexive, total order on write actions to each distinct variable. This is a per-variable order in which *all* threads observe writes to the variable; two threads cannot observe these writes in different orders.
- *Happens-before* (**hb**) is analogous to global temporal order – but unlike the sequentially consistent notion of time, it is partial. Happens-before is defined as $(\text{sb} \cup \text{rf})^+$: therefore statements ordered in the program syntax are ordered in time, as are reads with the writes they observe.
- *Atomicity* (**at** \subseteq **sb**) is an extension to standard C11 which we use to support LL-SC (see below). It is an injective function from a successful load-link action to a successful store-conditional, giving a LL-SC pair.

The semantics $\llbracket P \rrbracket$ of a program P is the set of executions $X \in \text{Exec}$ compatible with the thread-local semantics and the *validity axioms*, denoted $\text{valid}(X)$:

$$\llbracket P \rrbracket \stackrel{\Delta}{=} \{X \mid (\mathcal{A}(X), \text{sb}(X)) \in \langle P \rangle \wedge \text{valid}(X)\} \quad (3)$$

The validity axioms on an execution $(\mathcal{A}, \text{sb}, \text{at}, \text{rf}, \text{mo}, \text{hb})$ are:

- **HBDEF**: $\text{hb} = (\text{sb} \cup \text{rf})^+$ and hb is acyclic.

This axiom defines hb and enforces the intuitive property that there are no cycles in the temporal order. It also prevents an action reading from its hb -future: as rf is included in hb , this would result in a cycle.

- **HBvsMO**: $\neg \exists w_1, w_2. w_1 \xrightarrow{\text{hb}} w_2 \wedge w_1 \xrightarrow{\text{mo}} w_2$

This axiom requires that the order in which writes to a location become visible to threads cannot contradict the temporal order. But take note that writes may be ordered in mo but not hb .

- **COHERENCE**: $\neg \exists w_1, w_2, r. w_1 \xrightarrow{\text{mo}} w_2 \xrightarrow{\text{hb}} r \wedge w_1 \xrightarrow{\text{rf}} r$

This axiom generalises the sequentially consistent prohibition on reading overwritten values. If two writes are ordered in mo , then intuitively the second overwrites the first. A read that follows some write in hb or mo cannot read from writes earlier in mo – these earlier writes have been overwritten. However, unlike in sequential consistency, hb is partial, so there may be multiple writes that an action can legally read.

- **RFVAL**: $\forall r. (\neg \exists w'. w' \xrightarrow{\text{rf}} r) \implies (\text{val}(r) = 0 \wedge (\neg \exists w. w \xrightarrow{\text{hb}} r \wedge \text{gvar}(w) = \text{gvar}(r)))$

Most reads must take their value from a write, represented by an rf edge. However, the **RFVAL** axiom allows the rf edge to be omitted if the read takes the initial value 0 and there is no hb -earlier write to the same location. Intuitively, an hb -earlier write would supersede the initial value in a similar way to **COHERENCE**.

- **ATOM**: $\neg \exists w_1, w_2, ll, sc. w_1 \xrightarrow{\text{mo}} w_2 \wedge w_1 \xrightarrow{\text{rf}} ll \xrightarrow{\text{at}} sc \wedge w_2 \xrightarrow{\text{mo}} sc$

This axiom is adapted from [16]. For an LL-SC pair ll and sc , it ensures that there is no mo -intervening write w_2 that would invalidate the store.

Our model forbids the problematic relaxed behaviour of the message-passing (MP) program in Fig. 1 that yields $\mathbf{b} = 1 \wedge \mathbf{r} = 0$. Figure 3 shows an (invalid) execution that would exhibit this behaviour. To avoid clutter, here and in the following we omit hb edges obtained by transitivity and local variable values. This execution is allowed by the thread-local semantics of the MP program, but it is ruled out by the **COHERENCE** validity axiom. As hb is transitively closed, there is a derived hb edge $\text{store}(x, 1) \xrightarrow{\text{hb}} \text{load}(x, 0)$, which forms a **COHERENCE** violation. Thus, this is not an execution of the MP program. Indeed, any

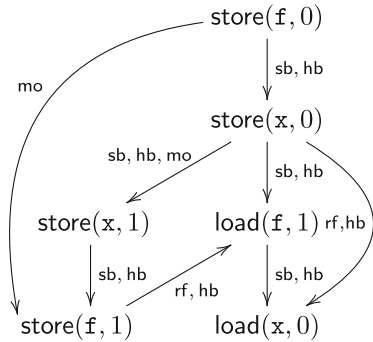


Fig. 3. An invalid execution of MP.

execution ending in $\text{load}(\mathbf{x}, 0)$ is forbidden for the same reason, meaning that the MP relaxed behaviour cannot occur.

3.6 Relaxed Observations

Finally, we define a notion of observational refinement suitable for our relaxed model. We assume a subset of *observable* global variables, $\text{OVar} \subseteq \text{GVar}$, which can only be accessed by the context and not by the code-block. We consider the actions and the hb relation on these variables to be the observations. We write $X|_{\text{OVar}}$ for the projection of X 's action set and relations to OVar , and use this to define \preceq_{ex} for our model:

$$X \preceq_{\text{ex}} Y \iff \mathcal{A}(X|_{\text{OVar}}) = \mathcal{A}(Y|_{\text{OVar}}) \wedge \text{hb}(Y|_{\text{OVar}}) \subseteq \text{hb}(X|_{\text{OVar}})$$

This is lifted to programs and blocks as in Sect. 2, def. (1) and (2). Note that in the more abstract execution, actions on observable variables must be the same, but hb can be weaker. This is because we interpret hb as a constraint on time order: two actions that are unordered in hb could have occurred in either order, or in parallel. Thus, weakening hb allows more observable behaviours (see Sect. 2).

3.7 Differences from C11

Our language's memory model is derived from the C11 formalisation in [5], with a number of simplifications. We chose C11 because it demonstrates most of the important features of axiomatic language models. However, we do not target the precise C11 model: rather we target an abstracted model that is rich enough to demonstrate our approach. Relaxed language semantics is still a very active topic of research, and several C11 features are known to be significantly flawed, with multiple competing fixes proposed. Some of our differences from [5] are intended to avoid such problematic features so that we can cleanly demonstrate our approach.

In C11 terms, our model covers release-acquire and non-atomic operations (the latter addressed in Sect. 7), and sequentially consistent fences. We deviate from C11 in the following ways:

- We omit *sequentially consistent* accesses because their semantics is known to be flawed in C11 [17]. We do handle sequentially consistent fences, but these are stronger than those of C11: we use the semantics proposed in [16]. It has been proved sound under existing compilation strategies to common multiprocessors.
- We omit *relaxed* (RLX) accesses to avoid well-known problems with thin-air values [4]. There are multiple recent competing proposals for fixing these problems, e.g. [14, 15, 20].
- Our model does not include infinite computations, because their semantics in C11-style axiomatic models remains undecided in the literature [4]. However, our proofs do not depend on the assumption that execution contexts are finite.

- Our language is based on shared variables, not dynamically allocated addressable memory, so for example we cannot write $y:=*x$; $z:=*y$. This simplifies our theory by allowing us to fix the variables accessed by a code-block upfront. We believe our results can be extended to support addressable memory, because C11-style models grant no special status to pointers; we elaborate on this in Sect. 4.
- We add LL-SC atomic instructions to our language in addition to C11’s standard CAS. To do this, we adapt the approach of [16]. This increases the observational power of a context and is necessary for full abstraction in the presence of non-atomics; see Sect. 8. LL-SC is available as a hardware instruction on many platforms supporting C11, such as Power and ARM. However, we do not propose adding LL-SC to C11: rather, it supports an interesting result in relaxed memory model theory. Our adequacy results do not depend on LL-SC.

4 Denotations of Code-Blocks

We construct the denotation for a code-block in two steps: (1) generate the *block-local* executions under a set of special cut-down contexts; (2) from each execution, extract a summary of interactions between the code-block and the context called a *history*.

4.1 Block-Local Executions

The block-local executions of a block $B \in \text{Block}$ omit context structure such as syntax and actions on variables not accessed in the block. Instead the context is represented by special actions **call** and **ret**, a set \mathcal{A}_B , and relations R_B and S_B , each covering an aspect of the interaction of the block and an arbitrary unrestricted context. Together, each choice of **call**, **ret**, \mathcal{A}_B , R_B , and S_B abstractly represents a set of possible syntactic contexts. By quantifying over the possible values of these parameters, we cover the behaviour of *all* syntactic contexts. The parameters are defined as follows:

- *Local variables.* A context can include code that precedes and follows the block on the same thread, with interaction through local variables, but – due to syntactic restriction – not through LL/SC atomic regions. We capture this with special action **call**(σ) at the start of the block, and **ret**(σ') at the end, where $\sigma, \sigma': \text{LVar} \rightarrow \text{Val}$ record the values of local variables at these points. Assume that variables in **LVar** are ordered: l_1, l_2, \dots, l_n . Then **call**(σ) is encoded by the action $(i, \text{call}, \text{None}, [\sigma(l_1), \dots, \sigma(l_n)])$, with fresh identifier i . We encode **ret** in the same way.
- *Global variable actions.* The context can also interact with the block through concurrent reads and writes to global variables. These interactions are represented by set \mathcal{A}_B of *context actions* added to the ones generated by the thread-local semantics of the block. This set only contains actions on the variables VS_B that B can access (VS_B can be constructed syntactically). Given an execution X constructed using \mathcal{A}_B (see below) we write $\text{ctx}(X)$ to recover the set \mathcal{A}_B .

- *Context happens-before*. The context can generate **hb** edges between its actions, which affect the behaviour of the block. We track these effects with a relation R_B over actions in \mathcal{A}_B , **call** and **ret**:

$$R_B \subseteq (\mathcal{A}_B \times \mathcal{A}_B) \cup (\mathcal{A}_B \times \{\text{call}\}) \cup (\{\text{ret}\} \times \mathcal{A}_B) \quad (4)$$

The context can generate **hb** edges between actions directly if they are on the same thread, or indirectly through inter-thread reads. Likewise **call/ret** may be related to context actions on the same or different threads.

- *Context atomicity*. The context can generate **at** edges between its actions that we capture in the relation $S_B \subseteq \mathcal{A}_B \times \mathcal{A}_B$. We require this relation to be an injective function from LL to SC actions. We consider only cases where LL/SC pairs do not cross block boundaries, so we need not consider boundary-crossing **at** edges.

Together, **call**, **ret**, \mathcal{A}_B , R_B , and S_B represent a limited context, stripped of syntax, relations **sb**, **mo**, and **rf**, and actions on global variables other than VS_B . When constructing block-local executions, we represent all possible interactions by quantifying over all possible choices of σ , σ' , \mathcal{A}_B , R_B and S_B . The set $\llbracket B, \mathcal{A}_B, R_B, S_B \rrbracket$ contains all executions of B under this special limited context. Formally, an execution $X = (\mathcal{A}, \text{sb}, \text{at}, \text{rf}, \text{mo}, \text{hb})$ is in this set if:

1. $\mathcal{A}_B \subseteq \mathcal{A}$ and there exist variable maps σ, σ' such that $\{\text{call}(\sigma), \text{ret}(\sigma')\} \subseteq \mathcal{A}$. That is, the call, return, and extra context actions are included in the execution.
2. There exists a set \mathcal{A}_l and relation sb_l such that (i) $(\mathcal{A}_l, \text{sb}_l, \sigma') \in \langle B, \sigma \rangle$; (ii) $\mathcal{A}_l = \mathcal{A} \setminus (\mathcal{A}_B \cup \{\text{call}, \text{ret}\})$; (iii) $\text{sb}_l = \text{sb} \setminus \{(\text{call}, u), (u, \text{ret}) \mid u \in \mathcal{A}_l\}$. That is, actions from the code-block satisfy the thread-local semantics, beginning with map σ and deriving map σ' . All actions arising from the block are between **call** and **ret** in **sb**.
3. X satisfies the validity axioms, but with modified axioms **HBDEF'** and **ATOM'**. We define **HBDEF'** as: $\text{hb} = (\text{sb} \cup \text{rf} \cup R_B)^+$ and **hb** is acyclic. That is, context relation R_B is added to **hb**. **ATOM'** is defined analogously with S_B added to **at**.

We say that \mathcal{A}_B , R_B and S_B are *consistent with B* if they act over variables in the set VS_B . In the rest of the paper we only consider consistent choices of \mathcal{A}_B , R_B , S_B . The *block-local executions* of B are then all executions $X \in \llbracket B, \mathcal{A}_B, R_B, S_B \rrbracket$.¹

¹ This definition relies on the fact that our language supports a fixed set of global variables, not dynamically allocated addressable memory (see Sect. 3.7). We believe that in the future our results can be extended to support dynamic memory. For this, the block-local construction would need to quantify over actions on all possible memory locations, not just the static variable set VS_B . The rest of our theory would remain the same, because C11-style models grant no special status to pointer values. Cutting down to a finite denotation, as in Sect. 5 below, would require some extra abstraction over memory – for example, a separation logic domain such as [9].

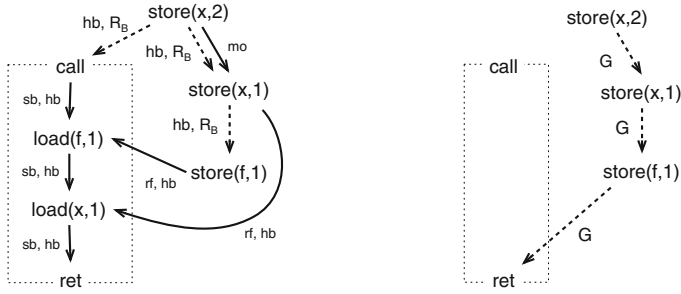


Fig. 4. *Left: block-local execution. Right: corresponding history.*

Example Block-Local Execution. The left of Fig. 4 shows a block-local execution for the code-block

$$11 := \text{load}(f); 12 := \text{load}(x) \quad (5)$$

Here the set VS_B of accessed global variables is $\{f, x\}$. As before, we omit local variables to avoid clutter. The context action set \mathcal{A}_B consists of the three stores, and R_B is denoted by dotted edges.

In this execution, both \mathcal{A}_B and R_B affect the behaviour of the code-block. The following path is generated by R_B and the load of $f = 1$:

$$\text{store}(x, 2) \xrightarrow{\text{mo}} \text{store}(x, 1) \xrightarrow{R_B} \text{store}(f, 1) \xrightarrow{\text{rf}} \text{load}(f, 1) \xrightarrow{\text{sb}} \text{load}(x, 1)$$

Because hb includes sb , rf , and R_B , there is a transitive edge $\text{store}(x, 1) \xrightarrow{\text{hb}} \text{load}(x, 1)$. The edge $\text{store}(x, 2) \xrightarrow{\text{mo}} \text{store}(x, 1)$ is forced because the HBVSMO axiom prohibits mo from contradicting hb . Consequently, the COHERENCE axiom forces the code-block to read $x = 1$.

4.2 Histories

From any block-local execution X , its *history* summarises the interactions between the code-block and the context. Informally, the history records hb over context actions, `call`, and `ret`. More formally the history, written $\text{hist}(X)$, is a pair (\mathcal{A}, G) consisting of an action set \mathcal{A} and *guarantee relation* $G \subseteq \mathcal{A} \times \mathcal{A}$. Recall that we use $\text{contx}(X)$ to denote the set of context actions in X . Using this, we define the history as follows:

- The action set \mathcal{A} is the projection of X 's action set to `call`, `ret`, and $\text{contx}(X)$.
- The guarantee relation G is the projection of $\text{hb}(X)$ to

$$(\text{contx}(X) \times \text{contx}(X)) \cup (\text{contx}(X) \times \{\text{ret}\}) \cup (\{\text{call}\} \times \text{contx}(X)) \quad (6)$$

The guarantee summarises the code-block's effect on its context: it suffices to only track hb and ignore other relations. Note the guarantee definition is similar to the context relation R_B , definition (4). The difference is that `call` and `ret` are

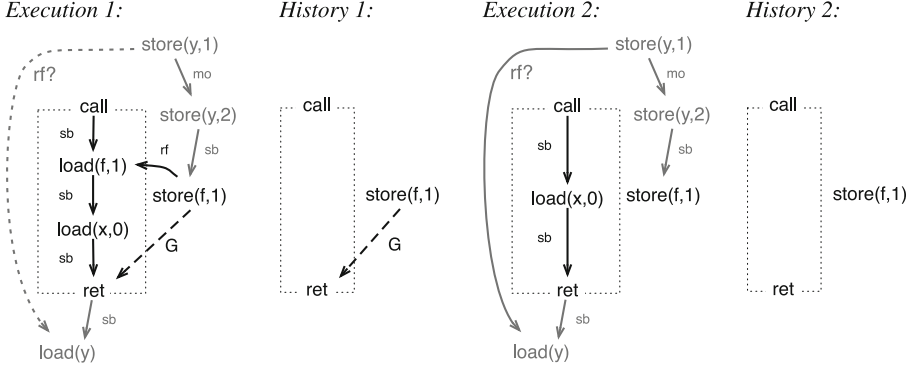


Fig. 5. Executions and histories illustrating the guarantee relation.

switched: this is because the guarantee represents **hb** edges generated by the code-block, while R_B represents the edges generated by the context. The right of Fig. 4 shows the history corresponding to the block-local execution on the left.

To see the interactions captured by the guarantee, compare the block given in def. (5) with the block `12:=load(x)`. These blocks have differing effects on the following syntactic context:

`store(y,1); store(y,2); store(f,1) || {-}; 13:=load(y)`

For the two-load block embedded into this context, $11 = 1 \wedge 13 = 1$ is not a possible post-state. For the single-load block, this post-state is permitted.²

In Fig. 5, we give executions for both blocks embedded into this context. We draw the context actions that are not included into the history in grey. In these executions, the code block determines whether the load of `y` can read value 1 (represented by the edge labelled ‘`rf?`’). In the first execution, the context load of `y` cannot read 1 because there is the path `store(y,1) \xrightarrow{mo} store(y,2) \xrightarrow{hb} load(y)` which would contradict the COHERENCE axiom. In the second execution there is no such path and the load may read 1.

It is desirable for our denotation to hide the precise operations inside the block – this lets it relate syntactically distinct blocks. Nonetheless, the history must record **hb** effects such as those above that are visible to the context. In Execution 1, the COHERENCE violation is still visible if we only consider context operations, `call`, `ret`, and the guarantee G – i.e. the history. In Execution 2, the fact that the read is permitted is likewise visible from examining the history. Thus the guarantee, combined with the local variable post-states, capture the effect of the block on the context without recording the actions inside the block.

² We choose these post-states for exposition purposes – in fact these blocks are also distinguishable through local variable `11` alone.

4.3 Comparing Denotations

The denotation of a code-block B is the set of histories of block-local executions of B under each possible context, i.e. the set

$$\{\text{hist}(X) \mid \exists \mathcal{A}_B, R_B, S_B. X \in \llbracket B, \mathcal{A}_B, R_B, S_B \rrbracket\}$$

To compare the denotations of two code-blocks, we first define a *refinement relation* on histories: $(\mathcal{A}_1, G_1) \sqsubseteq_h (\mathcal{A}_2, G_2)$ holds iff $\mathcal{A}_1 = \mathcal{A}_2 \wedge G_2 \subseteq G_1$. The history (\mathcal{A}_2, G_2) places fewer restrictions on the context than (\mathcal{A}_1, G_1) – a weaker guarantee corresponds to more observable behaviours. For example in Fig. 5, *History 1* \sqsubseteq_h *History 2* but not vice versa, which reflects the fact that History 1 rules out the read pattern discussed above.

We write $B_1 \sqsubseteq_q B_2$ to state that the denotation of B_1 *refines* that of B_2 . The subscript ‘q’ stands for the fact we *quantify* over both \mathcal{A} and R_B . We define \sqsubseteq_q by lifting \sqsubseteq_h :

$$B_1 \sqsubseteq_q B_2 \stackrel{\Delta}{\iff} \forall \mathcal{A}, R, S. \forall X_1 \in \llbracket B_1, \mathcal{A}, R, S \rrbracket. \exists X_2 \in \llbracket B_2, \mathcal{A}, R, S \rrbracket. \text{hist}(X_1) \sqsubseteq_h \text{hist}(X_2) \quad (7)$$

In other words, two code-blocks are related $B_1 \sqsubseteq_q B_2$ if for every block-local execution of B_1 , there is a corresponding execution of B_2 with a related history. Note that the corresponding history must be constructed under the same cut-down context \mathcal{A}, R, S .

Theorem 1 (ADEQUACY OF \sqsubseteq_q). $B_1 \sqsubseteq_q B_2 \implies B_1 \preceq_{bl} B_2$.

Theorem 2 (FULL ABSTRACTION OF \sqsubseteq_q). $B_1 \preceq_{bl} B_2 \implies B_1 \sqsubseteq_q B_2$.

As a corollary of the above theorems, a program transformation $B_2 \rightsquigarrow B_1$ is valid if and only if $B_1 \sqsubseteq_q B_2$ holds. We prove Theorem 1 in [10, Sect. B]. We give a proof sketch of Theorem 2 in Sect. 8 and a full proof in [10, Sect. F].

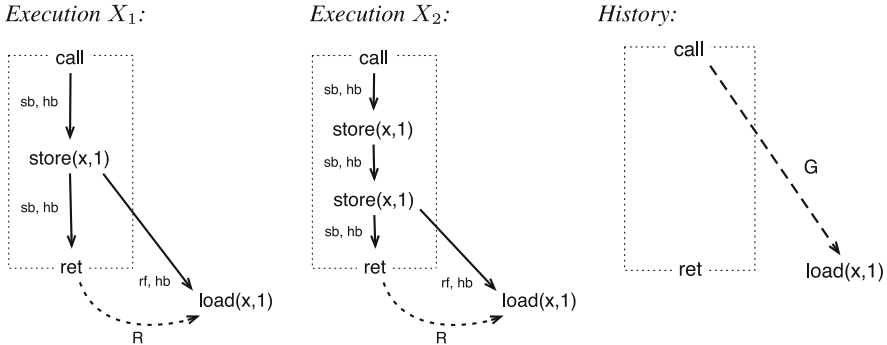


Fig. 6. History comparison for an example program transformation.

4.4 Example Transformation

We now consider how our approach applies to a simple program transformation:

$$B_2: \text{store}(x, l1); \text{store}(x, l1) \rightsquigarrow B_1: \text{store}(x, l1)$$

To verify this transformation, we must show that $B_1 \sqsubseteq_q B_2$. To do this, we must consider the unboundedly many block-local executions. Here we just illustrate the reasoning for a single block-local execution; in Sect. 5 below we define a context reduction which lets us consider a finite set of such executions.

In Fig. 6, we illustrate the necessary reasoning for an execution $X_1 \in \llbracket B_1, \mathcal{A}, R, S \rrbracket$, with a context action set \mathcal{A} consisting of a single load $x = 1$, a context relation R relating `ret` to the load, and an empty S relation. This choice of R forces the context load to read from the store in the block. We can exhibit an execution $X_2 \in \llbracket B_2, \mathcal{A}, R, S \rrbracket$ with a matching history by making the context load read from the final store in the block.

5 A Finite Denotation

The approach above simplifies contexts by removing syntax and non-hb structure, but there are still infinitely many $\mathcal{A}/R/S$ contexts for any code-block. To solve this, we introduce a type of context reduction which allows us to consider only finitely many block-local executions. This means that we can automatically check transformations by examining all such executions. However this ‘cut down’ approach is no longer fully abstract. We modify our denotation as follows:

- We remove the quantification over context relation R from definition (7) by fixing it as \emptyset . In exchange, we extend the history with an extra component called a *deny*.
- We eliminate redundant block-local executions from the denotation, and only consider a reduced set of executions X that satisfy a predicate $\text{cut}(X)$.

These two steps are both necessary to achieve finiteness. Removing the R relation reduces the amount of structure in the context. This makes it possible to then remove redundant patterns – for example, duplicate reads from the same write.

Before defining the two steps in detail, we give the structure of our modified refinement \sqsubseteq_c . In the definition, $\text{hist}_E(X)$ stands for the *extended history* of an execution X , and \sqsubseteq_E for refinement on extended histories.

$$B_1 \sqsubseteq_c B_2 \stackrel{\Delta}{\iff} \forall \mathcal{A}, S. \forall X_1 \in \llbracket B_1, \mathcal{A}, \emptyset, S \rrbracket. \\ \text{cut}(X_1) \implies \exists X_2 \in \llbracket B_2, \mathcal{A}, \emptyset, S \rrbracket. \text{hist}_E(X_1) \sqsubseteq_E \text{hist}_E(X_2) \quad (8)$$

As with \sqsubseteq_q above, the refinement \sqsubseteq_c is adequate. However, it is not fully abstract (we provide a counterexample in [10, Sect. D]). We prove the following theorem in [10, Sect. E].

Theorem 3 (ADEQUACY OF \sqsubseteq_c). $B_1 \sqsubseteq_c B_2 \implies B_1 \preceq_{bl} B_2$.

It should be intuitively clear why the first two of the above patterns are redundant. The main surprise is the third pattern, which preserves some non-visible writes. This is required by Theorem 3 for technical reasons connected to per-location coherence. We illustrate the application of $\text{cut}()$ to a block-local execution in Fig. 7.

5.2 Extended History (hist_E)

In our approach, each block-local execution represents a pattern of interaction between block and context. In our previous definition of \sqsubseteq_q , constraints imposed by the block are captured by the guarantee, while constraints imposed by the context are captured by the R relation. The definition (8) of \sqsubseteq_c removes the context relation R , but these constraints must still be represented. Instead, we replace R with a history component called a *deny*. This simplifies the block-local executions, but compensates by recording more in the denotation.

The deny records the hb edges that *cannot* be enforced due to the execution structure. For example, consider the block-local execution³ of Fig. 8.

This pattern could not occur in a context that generates the dashed edge D as a hb – to do so would violate the HBvsMO axiom. In our previous definition of \sqsubseteq_q , we explicitly represented the presence or absence of this edge through the R relation. In our new formulation, we represent such ‘forbidden’ edges in the history by a deny edge.

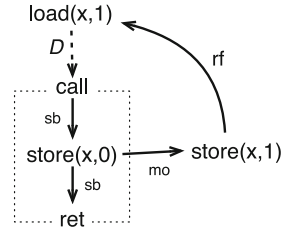


Fig. 8. A deny edge.

The *extended history* of an execution X , written $\text{hist}_E(X)$ is a triple (\mathcal{A}, G, D) , consisting of the familiar notions of action set \mathcal{A} and guarantee $G \subseteq \mathcal{A} \times \mathcal{A}$, together with deny $D \subseteq \mathcal{A} \times \mathcal{A}$ as defined below:

$$D \triangleq \{(u, v) \mid \text{HBvsMO-d}(u, v) \vee \text{Cohere-d}(u, v) \vee \text{RFval-d}(u, v)\} \cap ((\text{ctx}(X) \times \text{ctx}(X)) \cup (\text{ctx}(X) \times \{\text{call}\}) \cup (\{\text{ret}\} \times \text{ctx}(X)))$$

Each of the predicates HBvsMO-d, Cohere-d, and RFval-d generates the deny for one validity axiom. In the diagrammatic definitions below, dashed edges represent the deny edge, and hb^* is the reflexive-transitive closure of hb :

$$\text{HBvsMO-d}(u, v): \exists w_1, w_2. w_1 \xrightarrow{\text{hb}^*} u \xrightarrow{\text{D}} v \xrightarrow{\text{hb}^*} w_2$$

$\xleftarrow{\text{mo}}$

$$\text{Coherence-d}(u, v): w_1 \xrightarrow{\text{mo}} w_2 \xrightarrow{\text{hb}^*} u \xrightarrow{\text{D}} v \xrightarrow{\text{hb}^*} r$$

$\xleftarrow{\text{rf}}$

$$\text{RFval-d}(u, v): \exists w, r. \text{gvar}(w) = \text{gvar}(r) \wedge \neg \exists w'. w' \xrightarrow{\text{rf}} r \wedge w \xrightarrow{\text{hb}^*} u \xrightarrow{\text{D}} v \xrightarrow{\text{hb}^*} r$$

³ We use this execution for illustration, but in fact the $\text{cut}()$ predicate would forbid the load.

One can think of a deny edge as an ‘almost’ violation of an axiom. For example, if $\text{HBvsMO-d}(u, v)$ holds, then the context cannot generate an extra hb-edge $u \xrightarrow{\text{hb}} v$ – to do so would violate HBvsMO.

Because deny edges represent constraints on the context, weakening the deny places fewer constraints, allowing more behaviours, so we compare them with relational inclusion:

$$(\mathcal{A}_2, G_2, D_2) \sqsubseteq_E (\mathcal{A}_2, G_2, D_2) \xleftrightarrow{\Delta} \mathcal{A}_1 = \mathcal{A}_2 \wedge G_2 \subseteq G_1 \wedge D_2 \subseteq D_1$$

This refinement on extended histories is used to define our refinement relation on blocks, \sqsubseteq_c , def. (8).

5.3 Finiteness

Theorem 4 (FINITENESS). If for a block B and state σ the set of thread-local executions $\langle B, \sigma \rangle$ is finite, then so is the set of resulting block-local executions, $\{X \mid \exists \mathcal{A}, S. X \in \llbracket B, \mathcal{A}, \emptyset, S \rrbracket \wedge \text{cut}(X)\}$.

Proof (sketch). It is easy to see for a given thread-local execution there are finitely many possible visible reads and writes. Any two non-visible writes must be distinguished by at least one visible write, limiting their number. \square

Theorem 4 means that any transformation can be checked automatically if the two blocks have finite sets of thread-local executions. We assume a finite data domain, meaning action can only take finitely many distinct values in **Val**. Recall also that our language does not include loops. Given these facts, any transformations written in our language will satisfy finiteness, and can therefore be automatically checked.

6 Prototype Verification Tool

Stellite is our prototype tool that verifies transformations using the Alloy* model checker [12, 18]. Our tool takes an input transformation $B_2 \rightsquigarrow B_1$ written in a C-like syntax. It automatically converts the transformation into an Alloy* model encoding $B_1 \sqsubseteq_c B_2$. If the tool reports success, then the transformation is verified for unboundedly large syntactic contexts and executions.

An Alloy model consists of a collection of predicates on relations, and an instance of the model is a set of relations that satisfy the predicates. As previously noted in [28], there is therefore a natural fit between Alloy models and axiomatic memory models.

At a high level, our tool works as follows:

1. The two sides of an input transformation B_1 and B_2 are automatically converted into Alloy predicates expressing their syntactic structure. Intuitively, these block predicates are built by following the thread-local semantics from Sect. 3.

2. The block predicates are linked with a pre-defined Alloy model expressing the memory model and \sqsubseteq_c .
3. The Alloy* solver searches (using SAT) for a history of B_1 that has no matching history of B_2 . We use the higher-order Alloy* solver of [18] because the standard Alloy solver cannot support the existential quantification on histories in \sqsubseteq_c .

The Alloy* solver is parameterised by the maximum size of the model it will examine. However, our finiteness theorem for \sqsubseteq_c (Theorem 4) means there is a bound on the size of cut-down context that needs to be considered to verify any given transformation. If our tool reports that a transformation is correct, it is verified in all syntactic contexts of unbounded size.

Given a query $B_1 \sqsubseteq_c B_2$, the required context bound grows in proportion to the number of internal actions on distinct locations in B_1 . This is because our cutting predicate permits context actions if they interact with internal actions, either directly, or by interleaving between internal actions. In our experiments we run the tool with a model bound of 10, sufficient to give soundness for all the transformations we consider. Note that most of our example transformations do not require such a large bound, and execution times improve if it is reduced.

If a counter-example is discovered, the problematic execution and history can be viewed using the Alloy model visualiser, which has a similar appearance to the execution diagrams in this paper. The output model generated by our tool encodes the history of B_1 for which no history of B_2 could be found. As \sqsubseteq_c is not fully abstract, this counter-example could, of course, be spurious.

Stellite currently supports transformations on code-blocks with atomic reads, writes, and fences. It does not yet support code-blocks with non-atomic accesses (see Sect. 7), LL-SC, or branching control-flow. We believe supporting the above features would not present fundamental difficulties, since the structure of the Alloy encoding would be similar. Despite the above limitations, our prototype demonstrates that our cut-down denotation can be used for automatic verification of important program transformations.

Experimental Results. We have tested our tool on a range of different transformations. A table of experimental results is given in Fig. 9. Many of our examples are derived from [23] – we cover all their examples that fit into our tool’s input language. Transformations of the sort that we check have led to real-world bugs in GCC [19] and LLVM [8]. Note that some transformations are invalid because of their effect on local variables, e.g. `skip \rightsquigarrow l := load(x)`. The closely related transformation `skip \rightsquigarrow load(x)` throws away the result of the read, and is consequently valid.

Our tool takes significant time to verify some of the above examples, and two of the transformations cause the tool to time out. This is due to the complexity and non-determinism of the C11 model. In particular, our execution times are comparable to existing C++ model *simulators* such as Cppmem when they run on a few lines of code [3]. However, our tool is a sound transformation verifier, rather than a simulator, and thus solves a more difficult problem: transformations

Introduction, validity, time (s)			Elimination, validity, time (s)		
$\text{skip} \rightsquigarrow \text{fc}$	✓	76	$\text{fc} \rightsquigarrow \text{skip}$	×	15
$\text{skip} \rightsquigarrow \text{ld}(x)$	✓	429	$l := \text{ld}(x) \rightsquigarrow \text{skip}$	×	17
$\text{skip} \rightsquigarrow l := \text{ld}(x)$	×	18	$l := \text{ld}(x); \text{st}(x, l) \rightsquigarrow l := \text{ld}(x)$	×	64
$l := \text{ld}(x) \rightsquigarrow l := \text{ld}(x); \text{st}(x, l)$	×	72	$l := \text{ld}(x); l := \text{ld}(x) \rightsquigarrow l := \text{ld}(x)$	✓	2k
$l := \text{ld}(x) \rightsquigarrow l := \text{ld}(y); l := \text{ld}(x)$?	∞	$\text{st}(x, l); l := \text{ld}(x) \rightsquigarrow \text{st}(x, l)$	✓	9k
$l := \text{ld}(x) \rightsquigarrow l := \text{ld}(x); l := \text{ld}(x)$	✓	20k	$\text{st}(x, m); \text{st}(x, l) \rightsquigarrow \text{st}(x, l)$	✓	24k
$\text{st}(x, l) \rightsquigarrow \text{st}(x, l); \text{st}(x, l)$	×	136	$\text{fc}; \text{fc} \rightsquigarrow \text{fc}$	✓	382
$\text{fc} \rightsquigarrow \text{fc}; \text{fc}$	✓	248			
Exchange, validity, time (s)					
$\text{fc}; l := \text{ld}(x) \rightsquigarrow l := \text{ld}(x); \text{fc}$	×	26			
$\text{fc}; \text{st}(x, l) \rightsquigarrow \text{st}(x, l); \text{fc}$	×	50			
$l := \text{ld}(x); \text{fc} \rightsquigarrow \text{fc}; l := \text{ld}(x)$	×	79			
$\text{st}(x, l); \text{fc} \rightsquigarrow \text{fc}; \text{st}(x, l)$	×	145			
$l := \text{ld}(x); \text{st}(y, m) \rightsquigarrow \text{st}(y, m); l := \text{ld}(x)$	×	28			
$m := \text{ld}(y); l := \text{ld}(x) \rightsquigarrow l := \text{ld}(x); m := \text{ld}(y)$	×	118			
$\text{st}(y, m); l := \text{ld}(x) \rightsquigarrow l := \text{ld}(x); \text{st}(y, m)$?	∞			
$\text{st}(y, m); \text{st}(x, l) \rightsquigarrow \text{st}(x, l); \text{st}(y, m)$	×	641			

Fig. 9. Results from executing Stellite on a 32 core 2.3 GHz AMD Opteron, with 128 GB RAM, over Linux 3.13.0-88 and Java 1.8.0.91. **load/store/fence** are abbreviated to **ld/st/fc**. ✓ and × denote whether the transformation satisfies \sqsubseteq_c . ∞ denotes a timeout after 8 h.

are verified for unboundedly large syntactic contexts and executions, rather than for a single execution.

7 Transformations with Non-atomics

We now extend our approach to *non-atomic* (i.e. unsynchronised) accesses. C11 non-atomics are intended to enable sequential compiler optimisations that would otherwise be unsound in a concurrent context. To achieve this, any concurrent read-write or write-write pair of non-atomic actions on the same location is declared a *data race*, which causes the whole program to have undefined behaviour. Therefore, adding non-atomics impacts not just the model, but also our denotation.

7.1 Memory Model with Non-atomics

Non-atomic loads and stores are added to the model by introducing new commands $\text{store}_{\text{NA}}(x, l)$ and $l := \text{load}_{\text{NA}}(x)$ and the corresponding kinds of actions: $\text{store}_{\text{NA}}, \text{load}_{\text{NA}} \in \text{Kind}$. We let NA be the set of all actions of these kinds. We partition global variables so that they are either only accessed by non-atomics, or by atomics. We do not permit non-atomic LL-SC operations. Two new validity axioms ensure that non-atomics read from writes that happen before them, but not from stale writes:

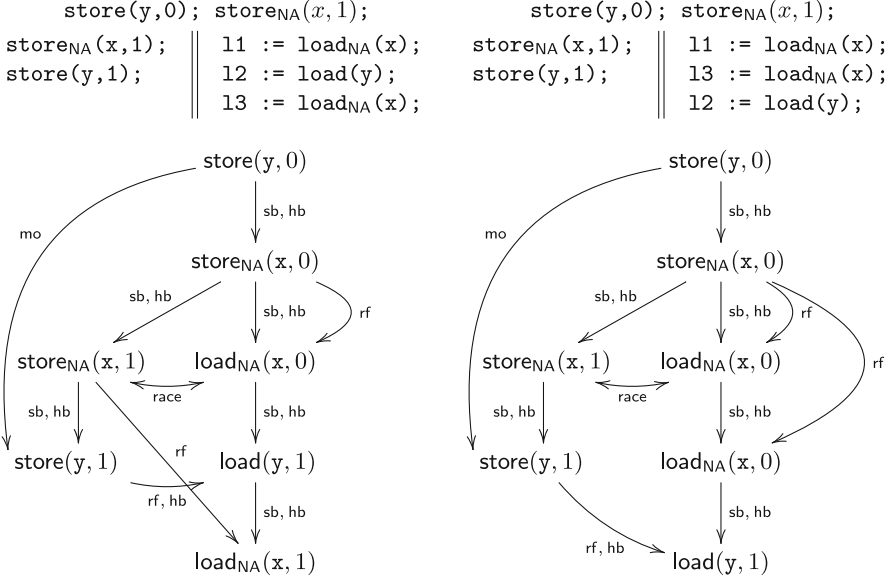


Fig. 10. *Top left:* augmented MP, with non-atomic accesses to x , and a new racy load. *Top right:* the same code optimised with $B_2 \rightsquigarrow B_1$. *Below each:* a valid execution.

- RFHBNA: $\forall w, r \in \mathbf{NA}. w \xrightarrow{rf} r \implies w \xrightarrow{hb} r$
- COHERNA: $\neg \exists w_1, w_2, r \in \mathbf{NA}. w_1 \xrightarrow{hb} w_2 \xrightarrow{hb} r$
 $\hspace{10em} \xrightarrow{rf}$

Modification order (**mo**) does not cover non-atomic accesses, and we change the definition of happens-before (**hb**), so that non-atomic loads do not add edges to it:

- HBDEF: $\mathbf{hb} = (\mathbf{sb} \cup (\mathbf{rf} \cap \{(w, r) \mid w, r \notin \mathbf{NA}\}))^+$

Consider the code on the left in Fig. 10: it is similar to MP from Fig. 1, but we have removed the if-statement, made all accesses to x non-atomic, and we have added an additional load of x at the start of the right-hand thread. The valid execution of this code on the left-hand side demonstrates the additions to the model for non-atomics:

- modification order (**mo**) relates writes to atomic y , but not non-atomic x ;
- the first load of x is forced to read from the initialisation by RFHBNA; and
- the second read of x is forced to read 1 because the **hb** created by the load of y obscures the now-stale initialisation write, in accordance with COHERNA.

The most significant change to the model is the introduction of a *safety axiom*, data-race freedom (DRF). This forbids non-atomic read-write and write-write pairs that are unordered in **hb**:

DRF:

$$\forall u, v \in \mathcal{A}. \left(\exists x. u \neq v \wedge u = (\text{store}(x, _)) \wedge \right. \\ \left. v \in \{(\text{load}(x, _)), (\text{store}(x, _))\} \right) \implies \left(\begin{array}{c} u \xrightarrow{\text{hb}} v \vee v \xrightarrow{\text{hb}} u \\ \vee u, v \notin \text{NA} \end{array} \right)$$

We write $\text{safe}(X)$ if an execution satisfies this axiom. Returning to the left of Fig. 10, we see that there is a violation of DRF – a race on non-atomics – between the first load of x and the store of x on the left-hand thread.

Let $\llbracket P \rrbracket_v^{\text{NA}}$ be defined same way as $\llbracket P \rrbracket$ is in Sect. 3, def. (3), but with adding the axioms RFHBNA and COHERNA and substituting the changed axiom HBDEF. Then the semantics $\llbracket P \rrbracket$ of a program with non-atomics is:

$$\llbracket P \rrbracket \triangleq \text{ if } \forall X \in \llbracket P \rrbracket_v^{\text{NA}}. \text{safe}(X) \text{ then } \llbracket P \rrbracket_v^{\text{NA}} \text{ else } \top$$

The undefined behaviour \top subsumes all others, so any program observationally refines a racy program. Hence we modify our notion of observational refinement on whole programs:

$$P_1 \preceq_{\text{pr}}^{\text{NA}} P_2 \iff (\text{safe}(P_2) \implies (\text{safe}(P_1) \wedge P_1 \preceq_{\text{pr}} P_2))$$

This always holds when P_2 is unsafe; otherwise, it requires P_1 to preserve safety and observations to match. We define observational refinement on blocks, $\preceq_{\text{bl}}^{\text{NA}}$, by lifting $\preceq_{\text{pr}}^{\text{NA}}$ as per Sect. 2, def. (2).

7.2 Denotation with Non-atomics

We now define our denotation for non-atomics, $\sqsubseteq_q^{\text{NA}}$, building on the ‘quantified’ denotation \sqsubseteq_q defined in Sect. 4. (We have also defined a finite variant of this denotation using the cutting strategy described in Sect. 5 – we leave this to [10, Sect. C].)

Non-atomic actions do not participate in happens-before (hb) or coherence order (mo). For this reason, we need not change the structure of the history. However, non-atomics introduce undefined behaviour \top , which is a special kind of observable behaviour. If a block races with its context in some execution, the whole program becomes unsafe, for all executions. Therefore, our denotation must identify how a block may race with its context. In particular, for the denotation to be adequate, for any context C and two blocks $B_1 \sqsubseteq_q^{\text{NA}} B_2$, we must have that if $C(B_1)$ is racy, then $C(B_2)$ is also racy.

To motivate the precise definition of $\sqsubseteq_q^{\text{NA}}$, we consider the following (sound) ‘anti-roach-motel’ transformation⁴, noting that it might be applied to the right-hand thread of the code in the left of Fig. 10:

$$\begin{array}{l} B_2: 11 := \text{load}_{\text{NA}}(x); 12 := \text{load}(y); 13 := \text{load}_{\text{NA}}(x) \\ \rightsquigarrow B_1: 11 := \text{load}_{\text{NA}}(x); 13 := \text{load}_{\text{NA}}(x); 12 := \text{load}(y) \end{array}$$

⁴ This example was provided to us by Lahav, Giannarakis and Vafeiadis in personal communication.

In a standard roach-motel transformation [25], operations are moved into a synchronised block. This is sound because it only introduces new happens-before ordering between events, thereby restricting the execution of the program and preserving data-race freedom. In the above transformation, the second NA load of x is moved past the atomic load of y , effectively *out* of the synchronised block, reducing happens-before ordering, and possibly introducing new races. However, this is sound, because any data-race generated by B_1 must have already occurred with the first NA load of x , matching a racy execution of B_2 . Verifying this transformation requires that we reason about races, so $\sqsubseteq_q^{\text{NA}}$ must account for both racy and non-racy behaviour.

The code on the left of Fig. 10 represents a context, composed with B_2 , and the execution of Fig. 10 demonstrates that together they are racy. If we were to apply our transformation to the fragment B_2 of the right-hand thread, then we would produce the code on the right in Fig. 10. On the right in Fig. 10, we present a similar execution to the one given on the left. The reordering on the right-hand thread has led to the second load of x taking the value 0 rather than 1, in accordance with RFHBNA. Note that the execution still has a race on the first load of x , albeit with different following events. As this example illustrates, when considering racy executions in the definition of $\sqsubseteq_q^{\text{NA}}$, we may need to match executions of the two code-blocks that behave differently after a race. This is the key subtlety in our definition of $\sqsubseteq_q^{\text{NA}}$.

In more detail, for two related blocks $B_1 \sqsubseteq_q^{\text{NA}} B_2$, if B_2 generates a race in a block-local execution under a given (reduced) context, then we require B_1 and B_2 to have corresponding histories *only up to the point the race occurs*. Once the race has occurred, the following behaviours of B_1 and B_2 may differ. This still ensures adequacy: when the blocks B_1 and B_2 are embedded into a syntactic context C , this ensures that a race can be reproduced in $C(B_2)$, and hence, $C(B_1) \preceq_{\text{pr}}^{\text{NA}} C(B_2)$.

By default, C11 executions represent a program's complete behaviour to termination. To allow us to compare executions up to the point a race occurs, we use *prefixes* of executions. We therefore introduce the *downclosure* X^\downarrow , the set of $(\text{hb} \cup \text{rf})^+$ -prefixes of an execution X :

$$X^\downarrow \triangleq \{X' \mid \exists \mathcal{A}. X' = X|_{\mathcal{A}} \wedge \forall (u, v) \in (\text{hb}(X) \cup \text{rf}(X))^+. (v \in \mathcal{A} \Rightarrow u \in \mathcal{A})\}$$

Here $X|_{\mathcal{A}}$ is the projection of the execution X to actions in \mathcal{A} . We lift the downclosure to sets of executions in the standard way.

Now we define our refinement relation $B_1 \sqsubseteq_q^{\text{NA}} B_2$ as follows:

$$\begin{aligned} B_1 \sqsubseteq_q^{\text{NA}} B_2 &\triangleleft\!\!\!\triangleleft \forall \mathcal{A}, R, S. \forall X_1 \in \llbracket B_1, \mathcal{A}, R, S \rrbracket_v^{\text{NA}}. \exists X_2 \in \llbracket B_2, \mathcal{A}, R, S \rrbracket_v^{\text{NA}}. \\ &(\text{safe}(X_2) \implies \text{safe}(X_1) \wedge \text{hist}(X_1) \sqsubseteq_{\text{h}} \text{hist}(X_2)) \wedge \\ &(\neg \text{safe}(X_2) \implies \exists X'_2 \in (X_2)^\downarrow. \exists X'_1 \in (X_1)^\downarrow. \\ &\quad \neg \text{safe}(X'_2) \wedge \text{hist}(X'_1) \sqsubseteq_{\text{h}} \text{hist}(X'_2)) \end{aligned}$$

In this definition, for each execution X_1 of block B_1 , we witness an execution X_2 of block B_2 that is related. The relationship depends on whether X_2 is safe or unsafe.

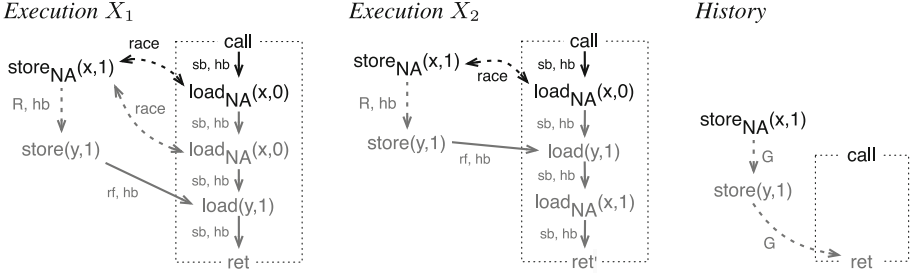


Fig. 11. History comparison for an NA-based program transformation

- If X_2 is safe, then the situation corresponds to \sqsubseteq_q – see Sect. 4, def. (7). In fact, if B_2 is *certain* to be safe, for example because it has no non-atomic accesses, then the above definition is equivalent to \sqsubseteq_q .
- If X_2 is unsafe then it has a race, and we do not have to relate the whole executions X_1 and X_2 . We need only show that the race in X_2 is feasible by finding a prefix in X_1 that refines the prefix leading to the race in X_2 . In other words, X_2 will behave consistently with X_1 *until it becomes unsafe*. This ensures that the race in X_2 will in fact occur, and its undefined behaviour will subsume the behaviour of B_1 . After X_2 becomes unsafe, the two blocks can behave entirely differently, so we need not show that the complete histories of X_1 and X_2 are related.

Recall the transformation $B_2 \rightsquigarrow B_1$ given above. To verify it, we must establish that $B_1 \sqsubseteq_q^{\text{NA}} B_2$. As before, we illustrate the reasoning for a single block-local execution – verifying the transformation would require a proof for all block-local executions.

In Fig. 11 we give an execution $X_1 \in \llbracket B_1, \mathcal{A}, R, S \rrbracket$, with a context action set \mathcal{A} consisting of a non-atomic store of $x = 1$ and an atomic store of $y = 1$, and a context relation R relating the store of x to the store of y . Note that this choice of context actions matches the left-hand thread in the code listings of Fig. 10, and there are data races between the loads and the store on x .

To prove the refinement for this execution, we exhibit a corresponding unsafe execution $X_2 \in \llbracket B_2, \mathcal{A}, R, S \rrbracket_v$. The histories of the *complete* executions X_1 and X_2 differ in their return action. In X_2 the load of y takes the value of the context store, so COHERNA forces the second load of x to read from the context store of x . This changes the values of local variables recorded in ret' . However, because X_2 is unsafe, we can select a prefix X'_2 which includes the race (we denote in grey the parts that we do not include). Similarly, we can select a prefix X'_1 of X_1 . We have that $\text{hist}(X'_1) = \text{hist}(X'_2)$ (shown in the figure), even though the histories $\text{hist}(X_1)$ and $\text{hist}(X_2)$ do not correspond.

Theorem 5 (ADEQUACY OF $\sqsubseteq_q^{\text{NA}}$). $B_1 \sqsubseteq_q^{\text{NA}} B_2 \implies B_1 \preceq_{\text{bl}}^{\text{NA}} B_2$.

Theorem 6 (FULL ABSTRACTION OF $\sqsubseteq_q^{\text{NA}}$). $B_1 \preceq_{\text{bl}}^{\text{NA}} B_2 \Rightarrow B_1 \sqsubseteq_q^{\text{NA}} B_2$.

We prove Theorem 5 in [10, Sect. B] and Theorem 6 in [10, Sect. F]. Note that the prefixing in our definition of $\sqsubseteq_q^{\text{NA}}$ is required for full abstraction—but it would be adequate to always require *complete* executions with related histories.

8 Full Abstraction

The key idea of our proofs of full abstraction (Theorems 2 and 6, given in full in [10, Sect. F]) is to construct a special syntactic context that is sensitive to one particular history. Namely, given an execution X produced from a block B with context happens-before R , this context C_X guarantees: (1) that X is the block portion of an execution of $C_X(B)$; and (2) for any block B' , if $C_X(B')$ has a different block history from X , then this is visible in different observable behaviour. Therefore for any blocks that are distinguished by different histories, C_X can produce a program with different observable behaviour, establishing full abstraction.

Special Context Construction. The precise definition of the special context construction C_X is given in [10, Sect. F] – here we sketch its behaviour. C_X executes the context operations from X in parallel with the block. It wraps these operations in auxiliary wrapper code to enforce context happens-before, R , and to check the history. If wrapper code fails, it writes to an error variable, which thereby alters the observable behaviour.

The context must generate edges in R . This is enforced by wrappers that use watchdog variables to create hb-edges: each edge $(u, v) \in R$ is replicated by a write and read on variable $h_{(u,v)}$. If the read on $h_{(u,v)}$ does not read the write, then the error variable is written. The shape of a successful read is given on the left in Fig. 12.

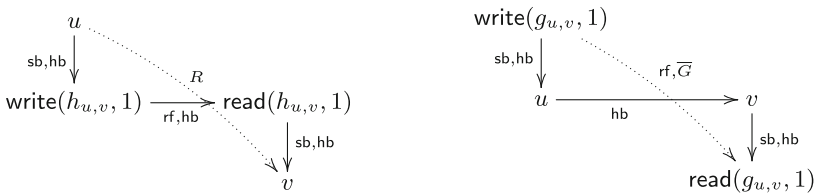


Fig. 12. The execution shapes generated by the special context for, on the *left*, generation of R , and on the *right*, errant history edges.

The context must also prohibit history edges beyond those in the original guarantee G , and again it uses watchdog variables. For each (u, v) *not* in G , the special context writes to watchdog variable $g_{(u,v)}$ before u and a reads $g_{(u,v)}$ after v . If the read of $g_{(u,v)}$ *does* read the value written before u , then there is an errant history edge, and the error location is written. An erroneous execution has the shape given on the right in Fig. 12 (omitting the write to the error location).

Full Abstraction and LL-SC. Our proof of full abstraction for the language with C11 non-atomics requires the language to also include LL-SC, not just C11's standard CAS: the former operation increases the observational power of the context. However, *without* non-atomics (Sect. 4) CAS would be sufficient to prove full abstraction.

9 Related Work

Our approach builds on our prior work [3], which generalises linearizability [11] to the C11 memory model. This work represented interactions between a library and its clients by sets of histories consisting of a guarantee and a deny; we do the same for code-block and context. However, our previous work assumed *information hiding*, i.e., that the variables used by the library cannot be directly accessed by clients; we lift this assumption here. We also establish both adequacy and full abstraction, propose a finite denotation, and build an automated verification tool.

Our approach is similar in structure to the seminal concurrency semantics of Brookes [6]: i.e. a code block is represented by a denotation capturing possible interactions with an abstracted context. In [6], denotations are sets of traces, consisting of sequences of global program states; context actions are represented by changes in these states. To handle the more complex axiomatic memory model, our denotation consists of sets of context actions and relations on them, with context actions explicitly represented as such. Also, in order to achieve full abstraction, Brookes assumes a powerful atomic `await()` instruction which blocks until the global state satisfies a predicate. Our result does not require this: all our instructions operate on single locations, and our strongest instruction is LL-SC, which is commonly available on hardware.

Brookes-like approaches have been applied to several relaxed models: operational hardware models [7], TSO [13], and SC-DRF [21]. Also, [7, 21] define tools for verifying program transformations. All three approaches are based on traces rather than partial orders, and are therefore not directly portable to C11-style axiomatic memory models. All three also target substantially stronger (i.e. more restrictive) models.

Methods for verifying code transformations, either manually or using proof assistants, have been proposed for several relaxed models: TSO [24, 26, 27], Java [25] and C/C++ [23]. These methods are non-compositional in the sense that verifying a transformation requires considering the trace set of the entire program—there is no abstraction of the context. We abstract both the sequential and concurrent context and thereby support automated verification. The above methods also model transformations as rewrites on program executions, whereas we treat them directly as modifications of program syntax; the latter corresponds more closely to actual compilers. Finally, these methods all require considerable proof effort; we build an automated verification tool.

Our tool is a sound verification tool – that is, transformations are verified for all context and all executions of unbounded size. Several tools exist for testing

(not verifying) program transformations on axiomatic memory models by searching for counter-examples to correctness, e.g., [16] for GCC and [8] for LLVM. Alloy was used by [28] in a testing tool for comparing memory models – this includes comparing language-level constructs with their compiled forms.

10 Conclusions

We have proposed the first fully abstract denotational semantics for an axiomatic relaxed memory model, and using this, we have built the first tool capable of automatically verifying program transformation on such a model. Our theory lays the groundwork for further research into the properties of axiomatic models. In particular, our definition of the denotation as a set of histories and our context reduction should be portable to other axiomatic models based on happens-before, such as those for hardware [1].

Acknowledgements. Thanks to Jeremy Jacob, Viktor Vafeiadis, and John Wickerson for comments and suggestions. Dodds was supported by a Royal Society Industrial Fellowship, and undertook this work while faculty at the University of York. Batty is supported by a Lloyds Register Foundation and Royal Academy of Engineering Research Fellowship.

References

1. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* **36**(2), 7:1–7:74 (2014)
2. Anderson, J.H., Moir, M.: Universal constructions for multi-object operations. In: *Symposium on Principles of Distributed Computing (PODC)*, pp. 184–193 (1995)
3. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: *Symposium on Principles of Programming Languages (POPL)*, pp. 235–248 (2013)
4. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: Vitek, J. (ed.) *ESOP 2015. LNCS*, vol. 9032, pp. 283–307. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_12
5. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: *Symposium on Principles of Programming Languages (POPL)*, pp. 55–66 (2011)
6. Brookes, S.: Full abstraction for a shared-variable parallel language. *Inf. Comput.* **127**(2), 145–163 (1996)
7. Burckhardt, S., Musuvathi, M., Singh, V.: Verifying local transformations on relaxed memory models. In: *International Conference on Compiler Construction (CC)*, pp. 104–123 (2010)
8. Chakraborty, S., Vafeiadis, V.: Validating optimizations of concurrent C/C++ programs. In: *International Symposium on Code Generation and Optimization (CGO)*, pp. 216–226 (2016)

9. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006). https://doi.org/10.1007/11691372_19
10. Dodds, M., Batty, M., Gotsman, A.: Compositional verification of compiler optimisations on relaxed memory (extended version). CoRR, [arXiv:1802.05918](https://arxiv.org/abs/1802.05918) (2018)
11. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **12**(3), 463–492 (1990)
12. Jackson, D.: *Software Abstractions - Logic Language and Analysis*, Revised edn. MIT Press, Cambridge (2012)
13. Jagadeesan, R., Petri, G., Riely, J.: Brookes is relaxed, almost! In: International Conference on Foundations of Software Science and Computational Structures (FOSSACS), pp. 180–194 (2012)
14. Jeffrey, A., Riely, J.: On thin air reads towards an event structures model of relaxed memory. In: Symposium on Logic in Computer Science (LICS), pp. 759–767 (2016)
15. Kang, J., Hur, C.-K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Symposium on Principles of Programming Languages (POPL), pp. 175–189 (2017)
16. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Symposium on Principles of Programming Languages (POPL), pp. 649–662 (2016)
17. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: Conference on Programming Language Design and Implementation (PLDI), pp. 618–632 (2017)
18. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: a general-purpose higher-order relational constraint solver. In: International Conference on Software Engineering (ICSE), pp. 609–619 (2015)
19. Morisset, R., Pawan, P., Zappa Nardelli, F.: Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In: Conference on Programming Language Design and Implementation (PLDI), pp. 187–196 (2013)
20. Pichon-Pharabod, J., Sewell, P.: A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In: Symposium on Principles of Programming Languages (POPL), pp. 622–633 (2016)
21. Poetzl, D., Kroening, D.: Formalizing and checking thread refinement for data-race-free execution models. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 515–530 (2016)
22. The C++ Standards Committee: *Programming Languages – C++* (2011). ISO/IEC JTC1 SC22 WG21
23. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In: Symposium on Principles of Programming Languages (POPL), pp. 209–220 (2015)
24. Vafeiadis, V., Zappa Nardelli, F.: Verifying fence elimination optimisations. In: International Conference on Static Analysis (SAS), pp. 146–162 (2011)
25. Ševčík, J., Aspinall, D.: On validity of program transformations in the Java memory model. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 27–51. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70592-5_3
26. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: Symposium on Principles of Programming Languages (POPL), pp. 43–54 (2011)
27. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: CompCertTSO: a verified compiler for relaxed-memory concurrency. *J. ACM* **60**(3), 22:1–22:50 (2013)

28. Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: Symposium on Principles of Programming Languages (POPL), pp. 190–204 (2017)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Author Index

- Abdulla, Parosh Aziz 442
Aguado, Joaquín 86
Aguirre, Alejandro 214

Bagnall, Alexander 561
Barthe, Gilles 117, 214
Batty, Mark 1027
Batz, Kevin 186
Bi, Xuan 3
Bichsel, Benjamin 145
Birkedal, Lars 214, 475
Bizjak, Aleš 214
Brunet, Paul 856

Charguéraud, Arthur 533
Chatterjee, Krishnendu 739
Chen, Tzu-Chun 799
Clebsch, Sylvan 885

Devriese, Dominique 475
Dodds, Mike 1027
Doko, Marko 357
Drossopoulou, Sophia 885

Eilers, Marco 502
Espitau, Thomas 117
Esteves-Verissimo, Paulo 619
Eugster, Patrick 799

Foster, Jeffrey S. 653
Franco, Juliana 885

Gaboardi, Marco 117, 214
García-Pérez, Álvaro 912
Garg, Deepak 214
Gehr, Timon 145
Goharshady, Amir Kafshdar 739
Gommerstadt, Hannah 771
Gotsman, Alexey 912, 1027
Grégoire, Benjamin 117
Guéneau, Armaël 533

Hamin, Jafar 415
Hicks, Michael 653

Hitz, Samuel 502
Hobor, Aquinas 385
Hsu, Justin 117
Hu, Raymond 799
Hupel, Lars 999

Jabs, Julian 60
Jacobs, Bart 415
Jagadeesan, Radha 968
Jia, Limin 771
Jonsson, Bengt 442

Kaminski, Benjamin Lucien 186
Kappé, Tobias 856
Karachalias, Georgios 327
Katoen, Joost-Pieter 186
Kobayashi, Naoki 711

Lahav, Ori 357, 940
Le, Xuan-Bach 385

Mardziel, Piotr 653
Matheja, Christoph 186
Matsuda, Kazutaka 31
Mendler, Michael 86
Merten, Samuel 561
Meshman, Yuri 912
Moore, Brandon 589
Müller, Peter 502, 683

Nipkow, Tobias 999

Oliveira, Bruno C. d. S. 3, 272
Ostermann, Klaus 60

Pédrot, Pierre-Marie 245
Peña, Lucas 589
Pfenning, Frank 771
Pichon-Pharabod, Jean 357
Pottier, François 533
Pouzet, Marc 86
Pretnar, Matija 327

- Raad, Azalea [940](#)
Rahli, Vincent [619](#)
Riely, James [968](#)
Roop, Partha [86](#)
Rosu, Grigore [589](#)
Ruef, Andrew [653](#)
- Saleh, Amr Hany [327](#)
Schrijvers, Tom [327](#)
Sergey, Ilya [912](#)
Silva, Alexandra [856](#)
Simpson, Alex [300](#)
Skorstengaard, Lau [475](#)
Stewart, Gordon [561](#)
Strub, Pierre-Yves [117](#)
Svendsen, Kasper [357](#)
- Tabareau, Nicolas [245](#)
Toninho, Bernardo [827](#)
Trinh, Cong Quy [442](#)
Tsukada, Takeshi [711](#)
- Urban, Caterina [683](#)
- Vafeiadis, Viktor [357](#), [940](#)
Vechev, Martin [145](#)
Velner, Yaron [739](#)
Viering, Malte [799](#)
Vitek, Jan [885](#)
Völz, Marcus [619](#)
von Hanxleden, Reinhard [86](#)
Voorneveld, Niels [300](#)
Vukotic, Ivana [619](#)
- Wang, Meng [31](#)
Watanabe, Keiichi [711](#)
Wei, Shiyi [653](#)
Wrigstad, Tobias [885](#)
- Xie, Ningning [3](#), [272](#)
- Yoshida, Nobuko [827](#)
- Zanasi, Fabio [856](#)
Ziarek, Lukasz [799](#)