Preliminary Proceedings

(Please do not distribute)

PPDP 2025

10-11 September 2025

University of Calabria, Rende, Italy

Folding Narrowing for the Analysis of Mutual Exclusion Protocols

Raúl López-Rueda
VRAIN, Universitat Politècnica de València
Valencia, Spain
rloprue@upv.es

Duong Dinh Tran
Japan Advanced Institute of Science and Technology
Nomi, Japan
duongtd@jaist.ac.jp

Canh Minh Do
Japan Advanced Institute of Science and Technology
Nomi, Japan
canhdo@jaist.ac.jp

Santiago Escobar
VRAIN, Universitat Politècnica de València
Valencia, Spain
sescobar@upv.es

Kazuhiro Ogata
Japan Advanced Institute of Science and Technology
Nomi, Japan
ogata@jaist.ac.jp

Abstract

Folding Narrowing is a symbolic analysis technique used in various areas of computer science. Maude-NPA has already demonstrated the effectiveness and potential of this technique in protocol analysis, particularly in the cryptographic domain. However, a thorough investigation into how protocols should be specified to make folding narrowing effective has not yet been conducted. The key to enabling this technique lies in ensuring the finiteness of the search space, which requires the search graph to collapse onto itself. Achieving this demands a careful design of the sorts and rules of the specified system. In many cases, such a design even allows for proving properties without relying on lemmas that other techniques typically require. In this work, we show how to prove the mutex property for the Test-and-Set (TAS), Qlock, and Anderson protocols, providing a detailed explanation of the reasoning needed to ensure the folding of the search space—without the use of lemmas.

CCS Concepts

• **Theory of computation** → **Rewrite systems**; **Logic and verification**; **Abstraction**; **Verification by model checking**; *Concurrent algorithms*; • **Software and its engineering** → *Formal software verification*; • **Security and privacy** → **Logic and verification**; Mathematical foundations of cryptography; *Cryptanalysis and other attacks*; • **Computing methodologies** → *Concurrent algorithms*.

Keywords

Folding narrowing, TAS, Qlock, Anderson, Maude, Mutual exclusion

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Raúl López-Rueda, Duong Dinh Tran, Canh Minh Do, Santiago Escobar, and Kazuhiro Ogata. 2025. Folding Narrowing for the Analysis of Mutual Exclusion Protocols. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Narrowing [Escobar et al. 2006] is a symbolic analysis technique capable of generating a search space from a rewrite system and an initial state, by analyzing the possibility of reaching a given target state. The solutions returned by narrowing are valuable for reasoning about the system's properties, as well as for identifying critical or invalid states, vulnerabilities, or for verifying the system's correct behavior, among other applications.

The most remarkable aspect of narrowing is that it is a symbolic analysis technique that goes beyond applying rewrite rules to fully instantiated terms, as is done in term rewriting. It also allows for the use of logical variables and employs unification to identify states that satisfy a target state, whether in a partially or fully instantiated form. For this reason, narrowing can be seen as an extension or generalization of rewriting. As a result, it becomes a very powerful technique, especially for functional or logic languages (or multi-paradigm, as in the case of Maude [Escobar 2014, 2018]). It is particularly useful for deducing unknown values and solving equations in systems specified through rewriting modulo axioms using equations and rules.

A particularly interesting feature of the unification used in Maude's narrowing [Clavel et al. 2025] is that it operates modulo equational axioms such as associativity (A), commutativity (C), and identity (U). This makes the unification algorithm a powerful tool, employed by protocol analysis tools such as Tamarin [Meier et al. 2013] and AKISS [Chadha et al. 2016]. Moreover, Maude-NPA [Escobar et al. 2007], among other protocol analysis tools, uses narrowing as its main symbolic analysis technique. Specifically, it leverages Maude's narrowing, as it is implemented in the Maude language. This technique is used to explore the search space through a backward search: it starts from a target state described in strand syntax and attempts to determine whether it can be reached from a valid initial protocol state. This illustrates the potential of narrowing, for instance,

when searching for vulnerabilities when analyzing cryptographic protocols.

Narrowing can generate a finite search space. In such cases, reasoning about this search space can lead to significant conclusions, such as proving certain security properties of a system or protocol. In the case of an invariant, if a pattern describing the negation of the invariant is unreachable using narrowing from a generic state (with variables), then the invariant is automatically proven for all possible instances. This allows, for example, and in contrast to other techniques such as theorem proving, to prove properties over data structures with an arbitrary number of items.

Unfortunately, the search space generated by narrowing is often infinite. This can occur because there is an infinite number of instances of the initial term (with variables) producing an infinite number of increasingly longer and longer finite sequences of either a rule or an equation application. In the case of narrowing with the rules of a Maude rewrite theory, folding narrowing [Bae et al. 2013; Bae and Meseguer 2014; Escobar and Meseguer 2007] may still be able to transform an infinite state system into a finite one. In the case of narrowing with the equations of a Maude functional module, folding *variant* narrowing [Escobar et al. 2008, 2010, 2012] is able to provide a finite number of all the possible normal forms of a term if the theory satisfies the *finite variant property* [Comon-Lundh and Delaune 2005] (FVP). Other narrowing extensions have been developed, with the aim of improving its performance or restricting its search space. One such extension involves the use of irreducibility constraints [Escobar and Meseguer 2019; López-Rueda et al. 2022].

Folding narrowing leverages the concept of state equivalence to transform an initially infinite search space into a finite one in many cases. One state folds into another one if the former is an instance modulo axioms and variant equations of the latter. Ideally, the final result is a finite search graph on which properties can be verified. What makes this particularly interesting is that narrowing is a highly general analysis technique. Therefore, if the search space can be folded and made finite, it enables a wide range of powerful applications.

The property of mutual exclusion or *mutex* [Dijkstra 1965] is essential for concurrency. This property ensures that at most one entity can access the critical resource at any time, even though the entities may be executed simultaneously. It is a crucial property in concurrent systems, as it prevents conflicts and ensures the integrity of shared resource access. In certain communication protocols, a violation of this property can allow attackers to impersonate legitimate participants, reuse credentials, or even intercept messages containing sensitive or private information. These messages can be exploited in various ways, including being modified to deceive other participants.

Focusing on authentication protocols, the mutex property ensures that two participants do not assume an exclusive role simultaneously. A clear example of this is the possession of a unique secret key during a key-exchange session. In distributed environments and scenarios involving untrusted agents, a mutex violation can lead to interleaving attacks, allowing attackers to manipulate the execution of protocols, which is a significant security compromise. In synchronization protocols, mutex ensures that at most one process can access a critical section at any given time. That is, two or more processes cannot access the critical section simultaneously. If this were to happen, the executions could exhibit erratic behavior, leading to inconsistent or unexpected results, which, in turn, may cause data corruption. All of these issues represent a clear security risk. Therefore, verifying the mutex property is essential to maintaining the protocol's robustness and preventing vulnerabilities that attackers could exploit.

In this work, our goal is to demonstrate that folding narrowing is a truly powerful technique to verify the mutex property in three synchronization protocols: TAS [Herlihy and Shavit 2012], Qlock [Ogata and Futatsugi 2013] and Anderson [Anderson 1990]. This study sets a precedent for the application of this advanced symbolic analysis technique to other protocols and systems. The technique is useful not only for these types of protocols, but also for others that operate with different characteristics and in other domains. Verifying the mutex property in these three protocols using the most general possible states ensures that the property will hold for any instance of them.

In previous work, some authors of this paper were able to prove properties such as mutual exclusion for TAS [Riesco and Ogata 2018], Qlock [Tran et al. 2021; Tran and Ogata 2022] and Anderson [Tran and Ogata 2020] with some lemmas required using *CafeOBJ* [Diaconescu and Futatsugi 1998], a proof assistant and a sibling language of Maude, although the techniques employed are very different from those used in this work.

In Section 2, we provide some background. In Section 3, we present the specification of the three case studies used in this work. In Section 4, after performing a preliminary check to ensure that processes can reach the critical section without any issue, we try to prove the mutex property for the three defined protocols. In Section 5, we discuss related work, explore potential avenues for future research, and assess the conclusions of this work.

2 Preliminaries

In this section, we provide background on rewrite theories, narrowing, and Maude. We follow the classical notation and terminology from [TeReSe 2003] for term rewriting, and from [Meseguer 1992] for rewriting logic and order-sorted notions. Note that all the Maude specifications included in this paper satisfy the required executability conditions presented below.

2.1 Rewrite theories

We assume an order-sorted signature Σ with a poset of sorts $(S, \leq)$. The poset $(S, \leq)$ of sorts for Σ is partitioned into equivalence classes, called *connected components*, by the equivalence relation $(\leq \cup \geq)^+$. We assume that each connected component $[s]$ has a *top element* under $\leq$, denoted $\top_{[s]}$ and called the *top sort* of $[s]$. This involves no real loss of generality, since if $[s]$ lacks a top sort, it can be easily added.

We assume an S-sorted family $\mathcal{X} = \{\mathcal{X}_s\}_{s \in S}$ of disjoint variable sets with each $\mathcal{X}_s$ countably infinite. $\mathcal{T}_\Sigma(\mathcal{X})_s$ is the set of terms of sort s, and $\mathcal{T}_{\Sigma,s}$ is the set of ground terms of sort s. We write $\mathcal{T}_\Sigma(\mathcal{X})$ and $\mathcal{T}_\Sigma$ for the corresponding order-sorted term algebras. Given a term t, $Var(t)$ denotes the set of variables in t.

A *substitution* $\sigma \in \mathcal{S}ubst(\Sigma, \mathcal{X})$ is a sorted mapping from a finite subset of $\mathcal{X}$ to $\mathcal{T}_\Sigma(\mathcal{X})$. Substitutions are written as $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where the domain of σ is $Dom(\sigma) = \{X_1, \ldots, X_n\}$

and the set of variables introduced by terms $t_1, \ldots, t_n$ is written $Ran(\sigma)$. The identity substitution is *id*. Substitutions are homomorphically extended to $\mathcal{T}_\Sigma(\mathcal{X})$. The application of substitution σ to a term t is denoted by $t\sigma$ or $\sigma(t)$.

A Σ-*equation* is an unoriented pair $t = t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_s$ for some sort $s \in S$. Given Σ and a set E of Σ-equations, order-sorted equational logic induces a congruence relation $=_E$ on terms $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$ (see [Meseguer 1998]). Throughout this paper we assume that $\mathcal{T}_{\Sigma,s} \neq \emptyset$ for every sort s, because this affords a simpler deduction system. We write $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ and $\mathcal{T}_{\Sigma/E}$ for the corresponding order-sorted term algebras modulo the congruence closure $=_E$, denoting the equivalence class of a term $t \in \mathcal{T}_\Sigma(\mathcal{X})$ as $[t]_E \in \mathcal{T}_{\Sigma/E}(\mathcal{X})$.

An *equational theory* (Σ, E) is a pair with Σ an order-sorted signature and E a set of Σ-equations. An equational theory (Σ, E) is *regular* if for each $t = t'$ in E, we have $Var(t) = Var(t')$. An equational theory (Σ, E) is *linear* if for each $t = t'$ in E, each variable occurs only once in t and in t'. An equational theory (Σ, E) is *sort-preserving* if for each $t = t'$ in E, each sort s, and each substitution σ, we have $t\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_s$ iff $t'\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_s$. An equational theory (Σ, E) is *defined using top sorts* if for each equation $t = t'$ in E, all variables in $Var(t)$ and $Var(t')$ have a top sort.

An *E-unifier* for a Σ-equation $t = t'$ is a substitution σ such that $t\sigma =_E t'\sigma$. For $Var(t) \cup Var(t') \subseteq W$, a set of substitutions $CSU_E^W(t = t')$ is said to be a *complete* set of unifiers for the equality $t = t'$ modulo E away from W iff: (i) each $\sigma \in CSU_E^W(t = t')$ is an E-unifier of $t = t'$; (ii) for any E-unifier ρ of $t = t'$ there is a $\sigma \in CSU_E^W(t = t')$ such that $\sigma|_W \sqsupseteq_E \rho|_W$ (i.e., there is a substitution η such that $(\sigma\eta)|_W =_E \rho|_W$); and (iii) for all $\sigma \in CSU_E^W(t = t')$, $Dom(\sigma) \subseteq (Var(t) \cup Var(t'))$ and $Ran(\sigma) \cap W = \emptyset$.

A *rewrite rule* is an oriented pair $l \to r$, where $l \notin \mathcal{X}$ and $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_s$ for some sort $s \in S$. An *(unconditional) order-sorted rewrite theory* is a triple (Σ, E, R) with Σ an order-sorted signature, E a set of Σ-equations, and R a set of rewrite rules. The set R of rules is *sort-decreasing* if for each $t \to t'$ in R, each $s \in S$, and each substitution σ, $t'\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_s$ implies $t\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_s$.

The rewriting relation on $\mathcal{T}_\Sigma(\mathcal{X})$, written $t \to_R t'$ or $t \to_{p,R} t'$ holds between t and t' iff there exist $p \in Pos_\Sigma(t)$, $l \to r \in R$ and a substitution σ, such that $t|_p = l\sigma$, and $t' = t[r\sigma]_p$. The relation $\to_{R/E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $=_E; \to_R; =_E$. The transitive (resp. transitive and reflexive) closure of $\to_{R/E}$ is denoted $\to^+_{R/E}$ (resp. $\to^*_{R/E}$). A term t is called $\to_{R/E}$-irreducible (or just R/E-irreducible) if there is no term t' such that $t \to_{R/E} t'$. For $\to_{R/E}$ confluent and terminating, the irreducible version of a term t is denoted by $t\downarrow_{R/E}$.

A relation $\to_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is defined as: $t \to_{p,R,E} t'$ (or just $t \to_{R,E} t'$) iff there are a non-variable position $p \in Pos_\Sigma(t)$, a rule $l \to r$ in R, and a substitution σ such that $t|_p =_E l\sigma$ and $t' = t[r\sigma]_p$. Reducibility of $\to_{R/E}$ is undecidable in general since E-congruence classes can be arbitrarily large. Therefore, R/E-rewriting is usually implemented [Jouannaud and Kirchner 1986] by R, E-rewriting under some conditions on R and E such as confluence, termination, and coherence.

We call (Σ, B, E) a *decomposition* of an order-sorted equational theory $(\Sigma, E \cup B)$ if B is regular, linear, sort-preserving, defined using top sorts, and has a finitary and complete unification algorithm, which implies that B-matching is decidable, and the equations E oriented into rewrite rules $\overrightarrow{E}$ are *convergent*, i.e., confluent, terminating, and strictly coherent [Meseguer 2017] modulo B, and sort-decreasing.

Given a decomposition (Σ, B, E) of an equational theory, (t', θ) is an E, B-*variant* [Comon-Lundh and Delaune 2005; Escobar et al. 2012] (or just a variant) of term t if $t\theta\downarrow_{E,B} =_E t'$ and $\theta\downarrow_{E,B} =_E \theta$. A *complete set of E, B-variants* [Escobar et al. 2012] (up to renaming) of a term t is a subset, denoted by $[[t]]_{E,B}$, of the set of all E, B-variants of t such that, for each E, B-variant (t', σ) of t, there is an E, B-variant $(t'', \theta) \in [[t]]_{E,B}$ such that $(t'', \theta) \sqsupseteq_{E,B} (t', \sigma)$, i.e., there is a substitution ρ such that $t' =_B t''\rho$ and $\sigma|_{Var(t)} =_B (\theta\rho)|_{Var(t)}$. A decomposition (Σ, B, E) has the *finite variant property* (FVP) [Escobar et al. 2012] (also called a *finite variant decomposition*) iff for each Σ-term t, a complete set $[[t]]_{E,B}$ of its most general variants is finite.

In what follows, the set G of equations will in practice be $G = E \uplus B$ and will have a decomposition (Σ, B, E).

2.2 Narrowing

Definition 2.1 (Reachability goal). [Meseguer and Thati 2007] Given an order-sorted rewrite theory (Σ, G, R), a *reachability goal* is defined as a pair $t \overset{?}{\to}^*_{R/G} t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_s$. It is abbreviated as $t \overset{?}{\to}^* t'$ when the theory is clear from the context; t is the *source* of the goal and t' is the *target*. A substitution σ is a R/G-*solution* of the reachability goal (or just a solution for short) iff there is a sequence $\sigma(t) \to_{R/G} \sigma(u_1) \to_{R/G} \cdots \to_{R/G} \sigma(u_{k-1}) \to_{R/G} \sigma(t')$.

A set Γ of substitutions is said to be a *complete set of solutions* of $t \overset{?}{\to}^*_{R/G} t'$ iff (i) every substitution $\sigma \in \Gamma$ is a solution of $t \overset{?}{\to}^*_{R/G} t'$, and (ii) for any solution ρ of $t \overset{?}{\to}^*_{R/G} t'$, there is a substitution $\sigma \in \Gamma$ more general than ρ modulo G, i.e., $\sigma|_{Var(t)\cup Var(t')} \sqsupseteq_G \rho|_{Var(t)\cup Var(t')}$.

This provides a tool-independent semantic framework for symbolic reachability analysis of protocols under algebraic properties. Note that we have removed the condition $Var(r) \subseteq Var(l)$ for rewrite rules $l \to r \in R$ and thus a solution of a reachability goal must be applied to all terms in the rewrite sequence. If the terms t and t' in a goal $t \overset{?}{\to}^*_{T/G} t'$ are ground and rules have no extra variables in their right-hand sides, then goal solving becomes a standard rewriting reachability problem. However, since we allow terms t, t' with variables, we need a mechanism more general than standard rewriting to find solutions of reachability goals. *Narrowing* with R modulo G generalizes rewriting by performing *unification* at non-variable positions instead of the usual matching modulo G. Specifically, narrowing instantiates the variables in a term by a G-unifier that enables a rewrite modulo G with a given rule of R and a term position.

Definition 2.2 (Narrowing modulo G). [Meseguer and Thati 2007] Given an order-sorted rewrite theory (Σ, G, R), the *narrowing* relation on $\mathcal{T}_\Sigma(\mathcal{X})$ modulo G is defined as $t \rightsquigarrow_{\sigma,R,G} t'$ (or $t \overset{\sigma}{\rightsquigarrow} t'$ if R, G is understood) iff there is $p \in Pos_\Sigma(t)$, a rule $l \to r$ in R such that $Var(t) \cap (Var(l) \cup Var(r)) = \emptyset$, and $\sigma \in CSU_G^V(t|_p = l)$ for a

set V of variables containing $Var(t)$, $Var(l)$, and $Var(r)$, such that $t' = \sigma(t[r]_p)$.

The reflexive and transitive closure of narrowing is defined as $t \rightsquigarrow^*_{\sigma,R,G} t'$ iff either $t = t'$ and $\sigma = id$, or there are terms $u_1, \ldots, u_n$, $n \geq 1$, and substitutions $\sigma_1, \ldots, \sigma_{n+1}$, such that $t \rightsquigarrow_{\sigma_1,R,G} u_1 \rightsquigarrow_{\sigma_2,R,G} u_2 \cdots u_n \rightsquigarrow_{\sigma_{n+1},R,G} t'$ and $\sigma = \sigma_1 \cdots \sigma_{n+1}$.

Soundness and completeness of narrowing with rules R modulo the equational theory G for solving reachability goals are proved in [Jouannaud and Kirchner 1986; Meseguer and Thati 2007] for order-sorted *topmost* rewrite theories, i.e., rewrite theories were all the rewrite steps happen at the top of the term.

Narrowing is a powerful technique; however, in many cases, its search space turns out to be infinite. To address this issue, several techniques have been developed. One such approach is folding narrowing [Escobar and Meseguer 2007]. This technique performs a symbolic state space reduction. It works by removing a newly explored symbolic state v' if it $E \cup B$-matches a previously explored state v, and replacing any transition leading to v' with a transition to v instead. This reduction can make the entire search space finite, thus enabling complete verification of invariants in certain infinite-state systems.

2.3 Maude

Maude is a formal language and tool designed for specification and analysis, based on rewriting logic. Below, we provide a brief overview of its syntax and refer the reader to [Clavel et al. 2025, 2007] for a more comprehensive understanding.

Sorts and subsorts (types and subtypes) are declared in Maude using the keywords `sort` or `sorts` and `subsort` or `subsorts`. Function symbols are declared using the syntax `op f : s_1 ...s_n -> s [atts]`, where $s_1, \ldots, s_n$ are the argument sorts, s is the target sort, and `atts` represents optional attributes. Operators can define custom syntax by using underscores '`_`' to indicate argument positions (e.g., `_and_`). Constructors, which define the elements of a sort, are marked with the `ctor` attribute. The attributes `assoc`, `comm`, and `id: t` specify that an operator is associative, commutative, and has t as its identity element, respectively.

In Maude, the following forms are used to declare equations, conditional equations, rewrite rules, and conditional rewrite rules:

- `eq u = v` [*atts*]
- `ceq u = v if` ψ [*atts*]
- `rl [l]: q => r` [*atts*]
- `crl [l]: q => r if` ψ [*atts*]

Equations relevant for variant-based equational unification are tagged with the `variant` attribute, while rules intended for narrowing are tagged with the `narrowing` attribute. It should be noted that Maude's built-in narrowing is not capable of handling conditional rules and equations.

Variables in equations and rules are either declared explicitly with `var` and `vars`, or introduced inline using the form `name:sort`.

To structure specifications and promote reuse, Maude allows importing existing modules into new ones. This is done using the keywords `including`, `extending`, or `protecting`, depending on the intended level of visibility and modification. For instance, `protecting` ensures that the imported module's definitions cannot be changed, while `extending` allows adding new equations or rules to its operators.

3 Case Studies

This section provides a detailed description of the Maude specifications for the mutex protocols with increasing complexity analyzed in this work. Section 3.1 presents the specification of the TAS protocol. Section 3.2 introduces the Qlock protocol, which builds upon the TAS specification, reusing some components while incorporating several modifications and extensions. Section 3.3 describes the Anderson protocol, which also reuses parts of the previous specifications and introduces additional changes. The key insights for proving an invariant are: (i) the initial state (with variables) of a narrowing reachability problem must already satisfy the invariant for all its instances, (ii) the target state describes the negation of the invariant as generally as possible, (iii) there is no limit on the depth of the narrowing-based search space, but (iv) in order for the search space to be folded, any finite sequence of rule applications is actually the repetition of a loop in the folded search space.

3.1 The Test-and-Set (TAS) Protocol in Maude

The first case study that we use for this work is the TAS protocol [Herlihy and Shavit 2012]. This protocol is used in concurrent programming to ensure mutual exclusion when accessing shared resources. As shown in Algorithm 1, TAS uses an atomic operation, `test&set`, with a given `locked` variable, which performs the following: it sets `locked` to `true` and returns the previous value of `locked`. When a process wants to enter the critical section, it reads the value of `locked`. If the value is `false`, the process sets it to `true` and enters the critical section. If, on the other hand, the value is `true`, then the process simply keeps trying to access the critical section, that is, waiting for `locked` to be `false`. Note that in the context of process synchronization, this blocking behavior is usually implemented using a `wait` instruction.

Algorithm 1 TAS Protocol (Pseudocode)

```
1. Loop
2.    "Remainder Section"
3.    repeat while test&set(locked);
4.    "Critical Section"
5.    locked := false;
```

This algorithm can be specified relatively easily in Maude since processes can only be in two different locations: *remainder section* or *critical section*. Therefore, only three rewrite rules are needed, two that move a process from one location to the other and another one for the processes to loop in the remainder section.

To specify the protocol in Maude, it is necessary to carefully consider the data types and data structures to be used. The design of this part is essential to take advantage of folding narrowing, as we will need certain *sorts* of variables both when defining the rewrite rules and when making narrowing calls. Thus, in Listing 1, we define processes only based on locations. Process IDs are not required in this case, since the protocol specification does not make use of them.

In the module LOC, we declare the labels for the locations where a process can be, along with two constants representing the remainder section rs, and the critical section cs. Once the locations and identifiers are defined, we can define the structure that determines what a process is in the PROC module. This is the point at which the problem gains greater complexity and demands a more nuanced approach.

```
fmod LOC is
  sorts RLoc CLoc Loc .
  subsorts RLoc CLoc < Loc .
  op rs : -> RLoc [ctor] .
  op cs : -> CLoc [ctor] .
endfm

fmod PROC is
  protecting LOC .
  sorts RProc CProc Proc .
  subsorts RProc CProc < Proc .
  op < p:_ > : Loc -> Proc [ctor] .
  op < p:_ > : RLoc -> RProc [ctor] .
  op < p:_ > : CLoc -> CProc [ctor] .
endfm
```

Listing 1: Modules defining processes for TAS in Maude.

To define narrowing problems that start from a pattern sufficiently generalized yet specific enough to adequately represent the property to be verified, it is crucial to carefully design the sorts and subsorts. To achieve this, we make use of the order-sorted features provided by Maude, which offer greater control over the structure and behavior of the terms. Strategically defining the hierarchy of sorts helps restrict the search space enough to direct the narrowing towards a concrete and relevant exploration. In many cases, this can even result in the search space being folded, ensuring that the narrowing command has a finite execution time. Consequently, these sorts play a key role in writing rewrite rules precisely to determine when each rule should be applied, and in other cases to use variables for the initial or target states that are more or less general. In this case, we distinguish between two types of processes: those in rs and the other ones in cs. Each will belong to a different sort, but they will share a common supersort.

It is also necessary to define the data structure that will contain the processes. In this case, the most convenient way is to use multisets of processes, as, thanks to the axioms handled by Maude, defining equations and rewrite rules that use multisets is particularly straightforward. Note that narrowing uses unification with axioms, so using associativity, commutativity, and identity (all three properties are present in sets) is not an issue. Obviously, the use of these axioms may increase execution time in more complex scenarios, but in some others, like this one, using them simplifies things. Listing 2 defines the multiset structure used to represent processes.

Similarly, as we had defined processes in rs and processes in cs, this idea naturally extends to multisets of processes. In this way, we declare a sort RProcSet to represent multisets of processes in rs and a sort CProcSet to represent multisets of processes in cs. This will be very useful later for using variables that belong to these sorts. Additionally, we declare a sort MTProcSet, which is a subsort

```
fmod PROC-SET is
    protecting PROC .
    sorts MTProcSet RProcSet CProcSet ProcSet .
    subsorts MTProcSet < RProcSet CProcSet < ProcSet .
    subsort RProc < RProcSet .
    subsort CProc < CProcSet .
    subsort Proc < ProcSet .
    op mt : -> MTProcSet [ctor] .
    op __ : MTProcSet MTProcSet -> MTProcSet
            [ctor assoc comm id: mt] .
    op __ : ProcSet ProcSet -> ProcSet [ctor ditto] .
    op __ : RProcSet RProcSet -> RProcSet [ctor ditto] .
    op __ : CProcSet CProcSet -> CProcSet [ctor ditto] .
endfm
```

Listing 2: Module defining sets of processes for TAS in Maude.

of all other multisets and represents empty multisets. The constant mt is used as the identity element, and processes are concatenated using a space to form a multiset.

Once all auxiliary modules are defined and the sorts, subsorts and data structures properly structured, we can specify the main module, which will define the system rewrite rules (in this case, TAS). Listing 3 shows the code required for this.

```
mod TAS is
  protecting PROC-SET .
  sort Config .
  op {_|`locked:_} : ProcSet Bool -> Config [ctor] .

  vars B B' : Bool .
  vars RPROC RPROC' RPROC'' : RProc .
  vars CPROC CPROC' : CProc .
  var RPSET : RProcSet .
  var PSET : ProcSet .

  rl [enter] :
       {< p: rs > PSET | locked: false} =>
       {< p: cs > PSET | locked: true} [narrowing] .

  rl [exit] :
       {< p: cs > PSET | locked: true} =>
       {< p: rs > PSET | locked: false} [narrowing] .
endm
```

Listing 3: Main module for the TAS protocol specification in Maude.

We define a sort Config that will represent the global state or configuration of the system at each moment. This configuration is composed of a multiset of processes and a Boolean value, as can be seen in the declared operator. The Boolean value indicates the value of the global lock at each moment. After this, we define some variables before using them, as well as others that are not used in this module but are necessary for the narrowing calls later in Section 4.1.

We also define three rewrite rules, as mentioned at the beginning of the section. Note that the narrowing attribute in the rewrite rules indicates that these rules are used for narrowing, not for rewriting. If

the locked value is `false`, the second rule (`enter`) moves a process in `rs` to `cs` and sets the locked value to `true`. The third rule (`exit`) performs the inverse operation, that is, it moves a process from `cs` to `rs`. In addition, it sets the locked value to `false`.

Note that we have leveraged Maude's order-sorted capabilities to design a specific hierarchy of sorts and subsorts. The mutex property applies only within a (possibly unbounded) multiset of processes in `rs`. This is typically described in other approaches, such as theorem proving, by providing either a predicate restricting the possible instances of a variable representing processes or a lemma that ensures some property for the instances of a variable for processes. Defining a sort `RProcSet` allows us to start from a general state, i.e., an arbitrary number of processes, while maintaining a restricted scope (only processes in `rs`). Furthermore, our specification does not require a lemma to enforce that when a process is in `cs`, the locked value is `true`. Instead, the concrete specification of the rewrite rules is sufficient to enforce this constraint.

3.2 The Qlock Protocol in Maude

The second case study that we use for this work is the Qlock protocol [Ogata and Futatsugi 2013]. This protocol is based on a queue-based synchronization mechanism designed to improve the efficiency of locks in multiprocessor systems. As shown in Algorithm 2, instead of waiting in a loop, as in TAS, Qlock organizes processes into a queue, so that each process waits for its turn in an ordered manner. In this way, the use of shared memory is minimized. When a process leaves the critical section, the next process in the queue is allowed to enter that section. This significantly reduces latency and resource consumption. Compared to other traditional locks in multithreaded or multi-core systems, Qlock is more scalable and efficient.

Algorithm 2 Qlock Protocol (Pseudocode)

1. **Loop**
2. **"Remainder Section"**
3. `rs: enq(queue, i);`
4. `ws: repeat until top(queue) = i;`
5. **"Critical Section"**
6. `cs: deq(queue);`

This algorithm is more complex to specify in Maude, and several iterations were required to achieve a functional specification for using folding narrowing. In this case, we do not only need a new location for the processes, but as a result, several additional sorts are necessary to handle it. In addition, a new rule is needed to account for the operation of processes that are in `ws` (waiting section). Qlock uses `ws` as an intermediate section between `cs` and `rs`. Processes in this location will wait to enter `cs`. Thus, processes in `rs` will enqueue their identifiers into the queue and move to `ws`.

Listing 4 shows the modifications to the modules previously defined in Listing 1 to align with the Qlock specification. We add a `PID` module to specify the process IDs. Note that the actual process identifiers are irrelevant for the protocol and we simply use constants `a`, `b`, `c`. In the `LOC` module, we introduce a new sort, `WLoc`, to represent the newly used location. In addition, we introduce the sort `RWLoc`, which serves as a supersort representing processes

```
fmod LOC is
  ...
  sorts WLoc CLoc .
  subsorts RLoc WLoc < RWLoc .
  subsorts RWLoc < Loc .
  op ws : -> WLoc [ctor] .
endfm

fmod PID is
    sort Pid .
    ops a b c : -> Pid .
endfm

fmod PROC is
  ...
  protecting PID .
  sorts WProc RWProc .
  subsorts RProc WProc < RWProc .
  subsorts RWProc < Proc .
  op < p[_]:_ > : Pid Loc -> Proc [ctor] .
  op < p[_]:_ > : Pid RLoc -> RProc [ctor] .
  op < p[_]:_ > : Pid WLoc -> WProc [ctor] .
  op < p[_]:_ > : Pid CLoc -> CProc [ctor] .
endfm
```

Listing 4: Extensions made to the process definition modules to adapt them to Qlock in Maude.

located in either `rs` or `ws`. In other words, this new sort represents non-critical locations. A new constant, `ws`, is also required for the new location. Similarly, in the `PROC` module, we introduce the sort `WProc` for processes in `ws` and the sort `RWProc`, which combines non-critical processes (in `rs` and `ws`). Furthermore, we add the definition of a process in `ws`.

As in the previous case, a data structure that contains the processes and assists in their management is also required. Listing 5 presents the additions and modifications made to Listing 2 (specifically in the `PROC-SET` module) to achieve this.

```
fmod PROC-SET is
  ...
  sorts RWProcSet .
  subsorts RProcSet WProcSet < RWProcSet .
  subsorts RWProcSet CProcSet < ProcSet .
  subsorts MTProcSet < WProcSet .
  subsort WProc < WProcSet .
  op __ : WProcSet WProcSet -> WProcSet [ctor ditto] .
  op __ : RWProcSet RWProcSet -> RWProcSet [ctor ditto] .

  eq [idem] : P:Proc P:Proc = P:Proc .
endfm
```

Listing 5: Extensions made to the sets of processes definition modules to adapt them to Qlock in Maude.

In this case, processes require an ID for the protocol to function as intended. Consequently, we no longer use multisets of processes, but rather sets. That is, process repetitions within sets are not allowed. To enforce this, we define an equation `idem` that removes

duplicate processes from a given multiset. Note that we are not using the `variant` attribute for the equation. If we were to specify the equation as `variant`, narrowing would be performed modulo this idempotency property, i.e., in equivalence classes containing an infinite number of repeated processes while the rules assume that only one of them exists. In other words, the equation would have the opposite effect of what we aim for. However, by using a non-variant equation, narrowing assumes irreducibility in the terms reduced by the equation. As a result, the equation is used solely for simplification, eliminating redundant identical processes, which is exactly what we want.

We introduce a new sort, `WProcSet`, which represents sets of processes in `ws`. Furthermore, we introduce a sort `RWProcSet` that serves as an intermediate subsort between the general sets `ProcSet` and the sets of processes in `rs` and `ws` (`RProcSet` and `WProcSet`). It is also necessary to add two new operators to specify the sets of the new sorts.

In this example, a queue is used to manage the order of processes waiting to access `cs`. There are many ways to implement this in Maude, but we have decided to implement a basic queue, as shown in Listing 6.

```
fmod QUEUE is
  protecting PID .
  sort Queue .
  subsort Pid < Queue .
  op empq : -> Queue [ctor] .
  op _|_ : Queue Queue -> Queue [ctor assoc id: empq] .
  op deq : Queue -> Queue .
  eq deq(empq) = empq [variant] .
  eq deq(P:Pid | Q:Queue) = Q:Queue [variant] .
endfm
```

Listing 6: Module defining a queue of process IDs in Maude.

We introduce the operator `deq` for queues, which is responsible for dequeuing the head and returning the rest of the queue. Note that `deq` is defined using *variant* equations, which will be used during variant-based equational unification.

Once all the auxiliary modules have been defined and the sorts, subsorts, and data structures properly organized, we can specify the main module, which encapsulates the rewrite rules. The corresponding code is presented in Listing 7.

Once again, there are variables that are not necessary within the module itself, but will be used later in the narrowing executions. The configuration now consists of a set of processes and a queue that manages the order of the processes requested to enter `cs`. In addition, now four rewrite rules are used. The `want` rule models the behavior of any process when it is in `rs` and wants to enter `cs`. The process will move to `ws` and its process ID is added to the end of the queue, whether the queue was empty or not. The `try` rule allows a process to move from `ws` to `cs`. This occurs only when the process ID is at the head of the queue. That is a protocol requirement, so the formal specifications follow that requirement. However, according to other methodologies and technologies, this would typically need to be proven as a lemma. But in this case, it is not necessary, as the specification of the protocol in Maude, through the rule definitions,

```
mod QLOCK is
  protecting QUEUE .
  protecting PROC-SET .
  sort Config .
  op {_|`queue:_} : ProcSet Queue -> Config [ctor] .

  var P : Pid .
  vars Q Q' : Queue .
  vars RWPROC RWPROC' RWPROC'' : RWProc .
  vars CPROC CPROC' : CProc .
  var PROC : Proc .
  var RPSET : RProcSet .
  var RWPSET : RWProcSet .
  var PSET : ProcSet .

  rl [want] :
     {< p[P]: rs > PSET | queue: Q } =>
     {< p[P]: ws > PSET | queue: (Q | P)} [narrowing] .

  rl [repeat] :
       {< p[P]: ws > PSET | queue: Q } =>
       {< p[P]: ws > PSET | queue: Q } [narrowing] .

  rl [try] :
      {< p[P]: ws > PSET | queue: (P | Q) } =>
      {< p[P]: cs > PSET | queue: (P | Q) } [narrowing] .

  rl [exit] :
      {< p[P]: cs > PSET | queue: Q } =>
      {< p[P]: rs > PSET | queue: deq(Q) } [narrowing] .
endm
```

Listing 7: Main module for the Qlock protocol specification in Maude.

inherently includes this information. As a result, no lemmas are necessary. The `exit` rule removes a process from `cs`. At the same time, it performs the dequeue operation on the queue. Again, we use the rule `repeat` to specify a looping process when it is in `ws`. Due to the nondeterminism of the rules in Maude, in this case the process may enter the loop even when it is already at the front of the queue. This represents an abstraction of a process that remains in a similar state. The corresponding branch of the search space is inconsequential, since the new state obtained by applying that rule will be folded into a previously explored state.

As happens with TAS, defining the sorts and subsorts of Qlock hierarchically allows us to avoid the use of certain lemmas required by other techniques. An example is the restriction of an initial state when executing narrowing problems. Instead of using lemmas and theorem proving to constrain valid terms, we leverage Maude's order-sorted capabilities to create a type hierarchy that achieves a similar effect. In this case, we restrict the initial states to those with processes in `rs` or `ws`. If the initial state contains two processes in `cs`, it would directly violate the mutex property.

3.3 The Anderson Protocol in Maude

The third case study used in this work is the Anderson protocol [Anderson 1990]. Unlike Qlock, which relies on a dynamically growing queue and shared memory access to manage process ordering,

Anderson's protocol employs a circular queue with statically allocated positions, providing a more memory-efficient and predictable structure. As shown in Algorithm 3, the queue is implemented as a Boolean array of fixed size equal to the number of processes. Each process is assigned a specific slot in the array by atomically incrementing a global index variable. Once assigned, a process waits until it is granted access according to its position in the array. When the process enters the critical section, other processes are effectively blocked until it exits and signals the next one in line. One key advantage of Anderson's protocol over Qlock is its reduced contention on shared memory: since each process spins on a distinct location in the array, it avoids the cache invalidation overhead typically seen in queue-based locks like Qlock. This results in improved performance, especially under high contention, and makes the Anderson protocol particularly well-suited for systems with a large number of processors.

Algorithm 3 Anderson Protocol (Pseudocode)

```
1. Loop
2.    "Remainder Section"
3.    rs: place[i] := fetch&incmod(next, N);
4.    ws: repeat until array[place[i]];
5.    "Critical Section"
6.    array[place[i]] := false;
        array[(place[i] + 1)%N] := true;
```

Since we are using folding narrowing, the distribution of positions in the array through a global variable poses a problem. This is because, in order to ensure the folding of the search space, it is necessary to use a sufficiently general initial state, as shown later in Section 4. However, such a state may be instantiated into an unintended configuration where multiple processes start with the same index value for the array. A way to prevent this is by using induction, which is not supported by narrowing. Our solution is to use an abstraction of the protocol, which we describe next.

Listing 8 shows the definition of the processes in our definition of the Anderson protocol. Note that the defined module is importing the same `LOC` module from Listing 4.

```
fmod PROC is
  protecting LOC .
  sorts RProc WProc RWProc CProc TProc Proc .
  subsorts RProc WProc < RWProc .
  subsorts RWProc CProc < TProc < Proc .
  op < p:_ > : Loc -> Proc [ctor] .
  op < p:_ > : RLoc -> RProc [ctor] .
  op < p:_ > : WLoc -> WProc [ctor] .
  op < p:_ > : CLoc -> CProc [ctor] .
  op [_] : Proc -> TProc [ctor] .
endfm
```

Listing 8: Module defining processes for Anderson in Maude.

Since process identifiers are no longer required, each process is now represented solely by its current location. As in the Qlock model, a process can be in any of the previously defined locations. However, we introduce a new operator that encloses a process in square brackets. This operator is used to model when a process is granted permission to enter the critical section—that is, it corresponds to the case in the original queue where the array position assigned to that process holds the value `true`. Additionally, we need to define a new sort `TProc` representing those kind of processes.

To simulate the behavior of the circular queue, we combine the above with the definition of process lists, which preserve the order of the processes. This definition is provided in Listing 9.

We define general process lists, as well as sublists that contain only processes in the remaining, waiting, or critical sections. Additionally, we introduce a list definition that allows elements to be in the remaining or waiting state. These list structures are particularly useful when defining generic initial states for narrowing. For the identity we use the `empty` constant.

```
fmod PROC-LIST is
  protecting PROC .
  sort ProcList .
  sorts RsProcList WProcList RWProcList CsProcList .
  subsorts RsProcList WProcList < RWProcList .
  subsorts RWProcList CsProcList < ProcList .
  subsort RsProc < RsProcList .
  subsort WProc < WProcList .
  subsort CsProc < CsProcList .
  subsort Proc < ProcList .
  op empty : -> ProcList [ctor] .
  op __ : ProcList ProcList -> ProcList
         [ctor assoc id: empty] .
  op __ : RsProcList RsProcList -> RsProcList
         [ctor ditto] .
  op __ : WProcList WProcList -> WProcList
         [ctor ditto] .
  op __ : CsProcList CsProcList -> CsProcList
         [ctor ditto] .
  op __ : RWProcList RWProcList -> RWProcList
         [ctor ditto] .
endfm
```

Listing 9: Module defining lists of processes for Anderson in Maude.

With all auxiliary modules defined and the sorts, subsorts, and data structures properly structured, the main module—responsible for specifying the system's rewrite rules—can now be introduced. The complete specification is shown in Listing 10. In this case, we declare again additional variables that will be used later in the narrowing problems.

Our specification of the Anderson protocol assumes that, initially, there is a process of type `TProc`, that is, a process encapsulated with square brackets. This simulates the original initial state, in which only one position in the Boolean array is set to `true`. As a consequence, we need to duplicate the `want` rule so that it can be applied to processes both with and without square brackets. The rules `want-1` and `want-2` allow any process to transition at any time from location `rs` to location `ws`. The `fetch&incmod` operation is not needed, as its behavior is abstracted through the use of a process list that determines execution order.

After this transition, a process may enter the critical section only if it is encapsulated within square brackets—representing, in the original version, a position in the array set to `true`—via the `enter` rule. If the process is not encapsulated by this operator, it is interpreted as holding a `false` value in its assigned array position and must therefore wait until another process sets it to `true`. Finally, the `exit-1` and `exit-2` rules remove the process from the critical section and signal the next process in the circular queue—i.e., the one that would have obtained the `next` position—to indicate that it may now enter the critical section. This signaling is modeled by transferring the square brackets from the current process to the next one in the circular queue. This abstraction simulates the effect of setting the current process's array position to `false` and the next one's to `true`.

```
mod ANDERSON is
  protecting PROC-LIST .
  sort Config .
  op {_} : ProcList -> Config [ctor] .

  vars PLIST PLIST' PLIST'' PLIST''' : ProcList .
  vars RWPLIST RWPLIST' RWPLIST'' : RWProcList .
  var RSPLIST : RProcList .
  vars RWPROC RWPROC' : RWProc .
  vars CPROC CPROC' : CProc .
  vars RPROC RPROC' RPROC'' : RProc .

  rl [want-1] :
     {PLIST < p: rs > PLIST'} =>
     {PLIST < p: ws > PLIST'} [narrowing] .

  rl [want-2] :
     {PLIST [< p: rs >] PLIST'} =>
     {PLIST [< p: ws >] PLIST'} [narrowing] .

  rl [try] :
     {PLIST [< p: ws >] PLIST'} =>
     {PLIST [< p: cs >] PLIST'} [narrowing] .

  rl [exit-1] :
     {PLIST [< p: cs >] < p: ws > PLIST'} =>
     {PLIST < p: rs > [< p: ws >] PLIST'} [narrowing] .

  rl [exit-2] :
     {< p: ws > PLIST [< p: cs >]} =>
     {[< p: ws >] PLIST < p: rs >} [narrowing] .
endm
```

Listing 10: Main module for the Anderson protocol specification in Maude.

It should be noted that, to correctly model the behavior of a circular queue, the `exit` rule has been duplicated to account for the case in which the current process is the last one in the list—ensuring that the first process is signaled in such cases.

Having completed the necessary specifications, we are now prepared to execute the Anderson protocol through narrowing.

4 Proving the Mutual Exclusion Property with Folding Narrowing

In this section, we prove the mutex property for the protocols previously specified in Maude. We first perform preliminary checks to ensure that processes can reach `cs`, confirming that the protocols are well defined. Note that in all the narrowing problems in this section, we use folding narrowing. If we use narrowing without the folding technique for any of these examples, many of the executions will run indefinitely, as the specification of both protocols allows processes to enter and exit `cs` as many times as they wish. For this reason, we will not make comparisons between using and not using folding, as not using it will not provide any useful results. This, in itself, serves as strong evidence of how important this technique is for narrowing: it often transforms infinite search spaces into finite ones.

The mutex property is proven by showing that no counterexamples exist in each case. That is, by using folding narrowing, we will attempt, starting from an initial state, to find solutions for target states in which there are at least two processes in `cs`. If we find solutions, it means that the mutex property has been violated. If we do not find solutions, it means that the mutex property is proven to hold. Obviously, it is necessary to ensure that the search space is finite to prove the property in this way.

In Section 4.1, we use folding narrowing to prove the mutex property for the case of TAS protocol. In Section 4.2, we do the same for the Qlock protocol. In Section 4.3, we prove it for the Anderson protocol.

4.1 Mutual Exclusion for the TAS Protocol

4.1.1 Proving `cs` reachable. The first thing we want to check with folding narrowing is whether, starting from an initial multiset of processes (all of which are in `rs`), we can reach states in which there is at least one process in `cs`. The problem of folding narrowing in Maude to check this would be as follows.

```
{fold} vu-narrow in TAS :
{RPSET | locked: false} =>* {CPROC PSET | locked: B} .

Solution 1
state: {@1:RProcSet < p: cs > | locked: true}
accumulated substitution:
RPSET --> @1:RProcSet < p: rs >
variant unifier:
PSET --> %1:RProcSet
CPROC --> < p: cs >
B --> true
@1:RProcSet --> %1:RProcSet

No more solutions.
```

Note that in the initial state, in addition to starting from a multiset of processes in `rs`, we specify that the locked value must be `false`. In the target state, in principle, we do not care about its value, so we use a variable. What really matters to us is whether we can reach `cs` in that target state. Only one solution is obtained, where a process is in `cs`, and the remaining (arbitrary number of) processes are represented by a variable for a multiset of processes in `rs`. All other solutions that could be obtained are instances of this one,

so folding narrowing automatically omits them. The narrowing execution returns, in addition to the solution state, some more useful information: the accumulated substitution required to reach it from the initial state, and the unifier used to obtain the solution. From this point onward, we omit this additional information for the sake of brevity.

4.1.2 Proving mutex with three processes. We can try to prove the mutex property for a specific number of initial processes. This approach is closest to rewriting, but still more general, since we are using free variables. In the following, we use a folding narrowing problem in which we prove the mutex property for three processes in the system. To achieve this, we use an initial state containing three processes in the remaining section, represented by variables. For the target state, we use two variables representing processes in `cs`, along with an additional variable for the remaining set. In other words, the target state corresponds to a situation in which two processes are simultaneously in the critical section.

```
{fold} vu-narrow in TAS :
    {RPROC RPROC' RPROC'' | locked: false} =>*
    {CPROC CPROC' PSET | locked: B'} .

No solution.
```

Since we do not obtain any solution, the mutex property is proven by showing that no counterexamples exist when there are three processes in the system.

4.1.3 Proving mutex for infinite processes. We can repeat the previous operation for any fixed number of processes. However, what we are truly interested in is proving the mutex property for an arbitrary number of processes. To achieve this, we introduce the variables in the initial state as follows.

```
{fold} vu-narrow in TAS :
{RPSET | locked: B} =>*
{CPROC CPROC' PSET | locked: B'} .

No solution.
```

In this case, instead of directly using a specific number of processes as the starting point, we use a variable representing an indefinite number of processes in `rs`, as we did earlier to check if a process can enter `cs`. In the target state, we search for two processes that are in critical section at the same time, together with any process set variable. Since we do not obtain any solution, the mutex property is now proven for an indefinite number of initial processes.

Note that we are using variables for all processes and even for the global lock value. This is not particularly useful in this example, beyond seeking a greater generalization, but it can be very useful in other examples to significantly simplify the search or even to reduce the search space through folding. In fact, it will be very important in the next case study.

4.1.4 Explaining the result. To better understand the underlying dynamics, Figure 1 shows the resulting search graph obtained by executing the narrowing procedure employed to validate the mutex property in this protocol.

As shown in Figure 1, two states, s_1 and s_2, are generated from the initial state by applying the `repeat` and `enter` rules, respectively. Both states include a variable representing an arbitrary number of processes in `rs`. Additionally, s_1 contains an extra process in `rs`—the one that has performed the loop—while s_2 includes an extra process in `cs`, which is the one that has entered the critical section. However, s_1 is clearly an instance of the initial state s_0 and is therefore folded into it. From s_2, a new state s_3 will emerge, which moves the process in `cs` back to `rs`. State s_3 is also clearly an instance of s_0 and will likewise be folded into it. The states enclosed by a dotted line are not considered during the narrowing-based search for solutions; however, we include them in the figure to provide a clearer understanding of the folding technique.

4.2 Mutual Exclusion for the Qlock Protocol

4.2.1 Proving `cs` reachable. In the case of Qlock, simply checking whether any given process is capable of reaching `cs` is already an interesting process for folding narrowing. In the following problem shown, we attempt to do this starting from a set of processes in `rs` and with an initially empty queue.

```
{fold} vu-narrow in Qlock :
{RPSET | queue: empq} =>* {CPROC PSET | queue: Q} .
```

When executing the problem, the computation turns out to be infinite, or in other words, the search space is not folding completely. The problem lies, on one hand, in the use of an initially empty queue in the initial state, as during the execution of the protocol, the queue can grow infinitely if we start with an indefinite number of processes. This means that there will be states where the queue is always longer than in the previous ones, so they will not be instances of any of the previous states (they cannot fold into any of them). On the other hand, if we try to run the same problem but using an arbitrary initial queue, we will see that the execution will still be infinite. This is because we are starting from a set of an arbitrary number of processes in `rs`, and as the protocol progresses, an indefinite number of them may move to `ws`. These states with infinite combinations of processes in the two locations will not be able to fold into the initial state, so we need to generalize even further. This is where the sort that unifies processes in the non-critical locations in general comes into play.

```
{fold} vu-narrow in Qlock :
{RWPSET | queue: Q} =>* {CPROC PSET | queue: Q'} .

Solution 1
state: {@1:RWProcSet < p[@2:Pid]: cs > | queue: @2:Pid
       | @3:Queue}
...
No more solutions.
```

In this case, we will obtain, as in TAS, one general solution: one process is in `cs` and an indefinite number of processes are in `rs` and `ws`. Note that in the obtained solution, the identifier of the process in `cs` appears at the top of the queue. This will always be the case in valid states, and while other tools require proving this property with a lemma, in our case such a lemma is unnecessary: the specification itself ensures that states where this does not hold are excluded. Now, the search space is finite, due to the use of a

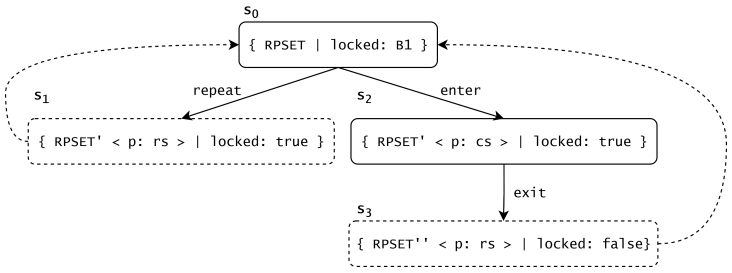


Figure 1: Search graph generated by the folding narrowing to prove the mutex property in the TAS protocol.

more generic sort variable and an arbitrary initial queue. Obviously, this initial state may contain instances of invalid states, but since the two solutions shown are generic (they use variables), we can also instantiate them to valid states, ensuring that the protocol behaves as expected.

4.2.2 Proving mutex with three processes. To prove the mutex property, we can once again start with a fixed number of initial processes.

```
{fold} vu-narrow in Qlock :
{RWPROC RWPROC' RWPROC'' | queue: empq} =>*
{CPROC CPROC' PSET | queue: Q} .

No solution.
```

This is again an approach closer to rewriting. By using a finite number of processes, we can even use an initially empty queue, and the search space can still fold. Since no solutions are obtained, the mutual exclusion property is proven to hold for the case in which three processes are present in the system. This analysis could be repeated for any fixed number of processes.

4.2.3 Proving mutex with infinite processes. If we want to prove the mutex property for an arbitrary number of processes, it is necessary to do something similar to what we did when testing whether a process can reach `cs`.

```
{fold} vu-narrow in Qlock :
{RWPSET | queue: Q} =>* {CPROC CPROC' PSET | queue: Q'} .

No solution.
```

Note that we again use an arbitrary initial queue and a set of an indefinite number of processes that can be in `rs` or `ws`. This initial state can instantiate invalid states (i.e., several processes are in `ws` but the queue is `empty`), but as long as we ensure that the search space is finite and do not obtain any solutions, the mutex property is proven by showing that no counterexamples exist in any case.

4.2.4 Explaining the result. To better understand the underlying dynamics, Figure 2 presents the resulting search graph obtained by executing the narrowing problem used to validate the mutex property in this protocol.

In this case, states s_1, s_2, and s_3 are generated from the initial state by applying the rules `want`, `try`, and `repeat`, respectively. On the one hand, states s_1 and s_3 contain processes in `rs` and `ws`, and are therefore instances of the initial state s_0, which means they can be folded onto it. On the other hand, state s_2 contains a process in `cs`. From s_2, several additional states may arise, which are not explicitly shown in the diagram due to space limitations. This is because some rules can be reapplied to this state, as there may be other processes in `rs` or `ws` within the set variable associated with the process in `cs`. In any case, all such states will be folded onto s_2 itself. The state s_4, which corresponds to the situation in which the process exits the `cs` phase, is explicitly shown. This state can also be folded onto s_0. As in Figure 1, only states without a dotted line are considered candidates for narrowing solutions.

4.3 Mutual Exclusion for the Anderson Protocol

4.3.1 Proving `cs` reachable. The proof that, in this example, it is possible to reach the critical section serves as a good example to illustrate how this abstraction operates under folding narrowing. In the problem shown below, the initial state is defined as a list of processes in `rs`, with the first process additionally encapsulated in square brackets. This models the original initial state, in which exactly one position in the Boolean array is set to `true`, while all others are set to `false`.

```
{fold} vu-narrow in ANDERSON :
{ [RPROC] RSPLIST } =>* { PLIST [CPROC] PLIST' } .
```

In this case, the computation becomes infinite once again, as observed in previous examples. The solution is similar: we must extend the initial list variable to include processes in both `rs` and `ws`, thereby creating a more generic state capable of collapsing other states. Since we are working with a list instead of a set and do not assume commutativity, it is necessary to use two initial list variables: one preceding the process enclosed in square brackets, and another following it. This allows for a generalization of the initial state, enabling us to start with any position of the virtual queue set to true and the rest to false. This representation is particularly useful for folding. Additionally—and although not required for folding in this case—we may apply a similar generalization to the variable representing the encapsulated process, further increasing the generality of the problem.

```
{fold} vu-narrow in ANDERSON :
{ RWPLIST [RPROC] RWPLIST' } =>*
{ PLIST [CPROC] PLIST' } .

Solution 1
state: {%1:RWProcList [< p: cs >] %2:RWProcList}
...
No more solutions.
```

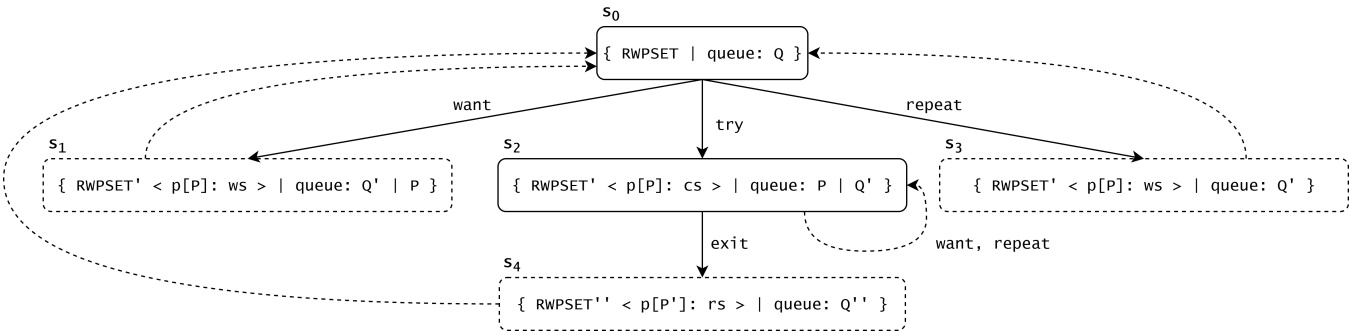


Figure 2: Search graph generated by the folding narrowing to prove the mutex property in the Qlock protocol.

With these modifications, the search space becomes finite, as folding narrowing is able to fully fold the computation. In this case, even with the bigger number of rules compared to previous examples, narrowing only returns a possible solution, which subsumes the rest. The folding mechanism detects all those that are equivalent to the one returned and collapses them accordingly. This reachability problem demonstrates that a process can indeed reach the critical section without issue.

4.3.2 Proving mutex with three processes. To verify the mutual exclusion property, we may begin once again by considering a system with only three processes.

```
{fold} vu-narrow in ANDERSON :
{ [RPROC] RPROC' RPROC'' } =>*
{ PLIST [CPROC] PLIST' [CPROC] PLIST'' } .

No solution.
```

This approach remains relatively close to standard rewriting; however, the true advantage of narrowing lies in its capacity for generalization. In any case, we use variables to represent the processes, which introduces a certain degree of generality. The initial state must include one process encapsulated in square brackets—emulating the condition in which the process finds its assigned position in the original array set to `true`. No solutions are obtained, which indicates that the mutual exclusion property holds for the case with three processes. While this analysis could be repeated for any fixed number of processes, we instead proceed to verify the property for an arbitrary number of processes.

4.3.3 Proving mutex for infinite processes. To verify the mutual exclusion property for an arbitrary number of processes, we must follow a strategy similar to the one used in the two previous examples. However, this case is somewhat more complex. Nevertheless, thanks to the well-defined type hierarchy and the structure of the rules, the corresponding narrowing problem remains relatively easy to specify.

```
{fold} vu-narrow in ANDERSON :
{ RWPLIST [RPROC] RWPLIST' } =>*
{ PLIST [CPROC] PLIST' [CPROC'] PLIST'' } .

No solution.
```

In this case, we again use two initial lists of processes that may be in either `rs` or `ws`. This is necessary because, as previously mentioned, without such generalization the search space cannot be effectively folded. Additionally, we specify a process encapsulated in square brackets, which simulates the position set to `true` in the original array. Given the way our specification of the protocol is defined, the initial state cannot be instantiated into invalid configurations. As the target state, we specify a configuration in which at least two processes are in `cs`. Since no solution is found, the mutual exclusion property is thus proven to hold for our Anderson protocol specification with an arbitrary number of processes.

In this example, it is interesting to observe that if the initial state includes at least two processes encapsulated within square brackets, we do obtain a general solution in which the target state contains two processes simultaneously in the critical section.

```
{fold} vu-narrow in ANDERSON :
    { RWPLIST [RPROC] RWPLIST' [RPROC'] RWPLIST''} =>*
    { PLIST' [< p: cs >] PLIST'' [< p: cs >] PLIST''' } .

Solution 1
state: {%1:RWProcList [< p: cs >] %2:RWProcList
       [< p: cs >] %3:RWProcList}
...
No more solutions.
```

Clearly, such an initial state is invalid, as it simulates a configuration in the original protocol where two positions in the Boolean array would initially be set to `true`. By the protocol's definition, this is not allowed: there must always be exactly one position initially set to `true`.

4.3.4 Explaining the result. For the Anderson protocol, the graph generated by the folding narrowing procedure is significantly larger. This increase in complexity is due to the use of lists, which yield many more unification possibilities than sets. In contrast, when using sets, commutativity combined with variables allows many equivalent states to be unified into a single one. The search graph generated by narrowing when proving the mutex property for this example is shown in Figure 3.

In this case, six new states are generated from the initial state. States s_1 and s_2 are obtained by applying the rule `want-1`. In both cases, an arbitrary process without square brackets is moved from `rs` to `ws`. In state s_3, the same operation is performed for the process

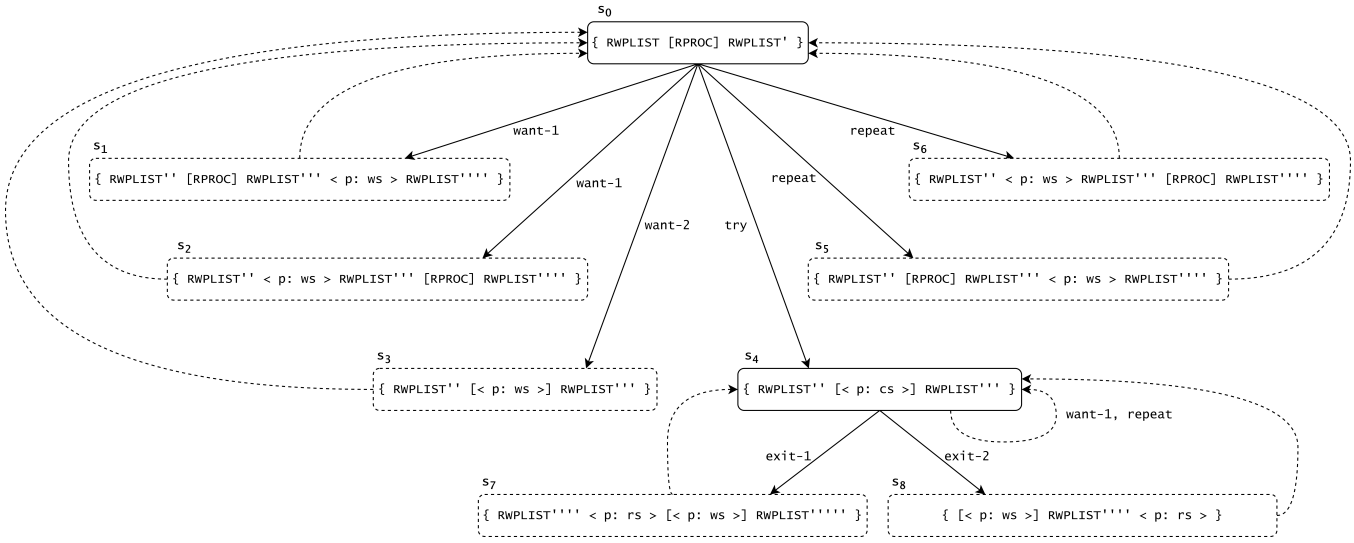


Figure 3: Search graph generated by the folding narrowing to prove the mutex property in the Anderson protocol.

with square brackets, using the rule `want-2`. States s_5 and s_6 result from applying the rule `repeat` to a process in `ws` that is unable to enter the critical section. State s_4 corresponds to the case where the process with square brackets enters the critical section via the `try` rule. From this state, the rules `want-1` and `repeat` can still be applied to other processes, but the resulting states fold back onto s_4 itself. Additionally, the rules `exit-1` and `exit-2` may be applied to exit the critical section. The rule used depends on whether the process is the last one in the list, that is, whether the modulo operation on the circular queue must be performed. As in the previous examples, the states that do not contain a process in the critical section—namely, s_1, s_2, s_3, s_5, and s_6—fold back onto the initial state s_0, as they are instances of it. Similarly, states s_7 and s_8, which are generated from state s_4, are folded onto it.

5 Related Work and Conclusions

In this work, we have shown how folding narrowing in Maude is an effective technique for verifying the mutex property in certain protocols. Other tools based on Maude's unification with associativity, commutativity, and identity, such as Tamarin [Meier et al. 2013] or AKiss [Chadha et al. 2016], are also used for similar purposes. Additionally, for protocol analysis, Maude-NPA [Escobar et al. 2007] employs narrowing in Maude and adopts the Dolev–Yao attacker model to symbolically represent attacks. Beyond these, various approaches such as model checking, theorem proving, and abstract interpretation have been used over the years to analyze protocol properties (see [Basin et al. 2018]).

Using CafeOBJ, a sibling language of Maude, some authors of this paper have previously verified the mutex property on TAS, Qlock, another Anderson variant, and other mutual exclusion protocols [Tran et al. 2021]. That work required conjecturing lemmas, proving them, and using them to establish the desired properties. In contrast, folding narrowing, by defining sorts and subsorts and leveraging Maude's order-sorted capabilities, restricts terms and states to avoid lemmas in mutex property analysis. As a result, executing the proofs becomes almost trivial, yielding compact and human-readable code.

In [Bae et al. 2024], the DM-Check invariant checker for Maude theories is presented. It is also based on folding narrowing but combined with theorem proving, namely the NuITP theorem prover [Durán et al. 2024]. This is necessary when there are inductive properties that cannot be handled by using sorts and subsorts.

A future direction is to apply this technique to prove mutex properties in other mutual exclusion protocols. Narrowing could also be used to verify additional properties within these or other protocols. We also plan to explore its use in the analysis of properties of post-quantum protocols.

Furthermore, our objective is to conduct a similar study using our canonical narrowing algorithm combined with SMT solving [López-Rueda et al. 2023] or canonical narrowing with conditional rules [López-Rueda and Escobar 2022]. This would facilitate a simpler analysis of synchronization protocols and extend to other protocols, such as time-based authentication. In particular, canonical narrowing may help reduce the complexity of reasoning in systems with richer equational theories or data types. Moreover, the integration with SMT solving opens up opportunities for automating proofs in domains where symbolic reasoning alone would be insufficient.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions.

This work was supported by Generalitat Valenciana Grant Number CIPROM/2022/6, by INCIBE's Chair funded by the EU-NextGenerationEU through the Spanish government's Plan de Recuperación, Transformación y Resiliencia Grant Number Chair Program, by MCIN/AEI/10.13039/501100011033 and ERDF A way of making Europe Grant Number PID2021-122830OB-C42, by NATO Science for Peace and Security Programme Grant Number G6133, and by JSPS KAKENHI Grant Number JP23K28060.

References

Thomas E. Anderson. 1990. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distributed Syst.* 1, 1 (1990), 6–16. doi:10.1109/71.80120

Kyungmin Bae, Santiago Escobar, Raúl López-Rueda, José Meseguer, and Julia Sapiña. 2024. Verifying Invariants by Deductive Model Checking. In *Rewriting Logic and Its Applications - 15th International Workshop, WRLA 2024, Luxembourg City, Luxembourg, April 6-7, 2024, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 14953)*, Kazuhiro Ogata and Narciso Martí-Oliet (Eds.). Springer, 3–21. doi:10.1007/978-3-031-65941-6_1

Kyungmin Bae, Santiago Escobar, and José Meseguer. 2013. Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. In *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands (LIPIcs, Vol. 21)*, Femke van Raamsdonk (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 81–96. doi:10.4230/LIPICS.RTA.2013.81

Kyungmin Bae and José Meseguer. 2014. Predicate Abstraction of Rewrite Theories. In *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8560)*, Gilles Dowek (Ed.). Springer, 61–76. doi:10.1007/978-3-319-08918-8_5

David A. Basin, Cas Cremers, and Catherine Meadows. 2018. Model Checking Security Protocols. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer, 727–762. doi:10.1007/978-3-319-10575-8_22

Rohit Chadha, Vincent Cheval, Ştefan Ciobâcă, and Steve Kremer. 2016. Automated Verification of Equivalence Properties of Cryptographic Protocols. *ACM Trans. Comput. Log.* 17, 4 (2016), 23. doi:10.1145/2926715

M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott. 2025. *Maude Manual (Version 3.5.1).* Technical Report. SRI International Computer Science Laboratory. Available at: http://maude.cs.illinois.edu.

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. 2007. *All About Maude: A High-Performance Logical Framework.* Lecture Notes in Computer Science, Vol. 4350. Springer. doi:10.1007/978-3-540-71999-1

Hubert Comon-Lundh and Stéphanie Delaune. 2005. The Finite Variant Property: How to Get Rid of Some Algebraic Properties. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3467)*, Jürgen Giesl (Ed.). Springer, 294–307. doi:10.1007/978-3-540-32033-3_22

Razvan Diaconescu and Kokichi Futatsugi. 1998. *CafeOBJ Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification.* AMAST Series in Computing, Vol. 6. World Scientific. doi:10.1142/3831

E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9 (Sept. 1965), 569. doi:10.1145/365559.365617

Francisco J. Durán, Santiago Escobar, José Meseguer, and Julia Sapiña. 2024. NuITP: An Inductive Theorem Prover for Equational Program Verification. In *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024, Milano, Italy, September 9-11, 2024*, Alessandro Bruni, Alberto Momigliano, Matteo Pradella, Matteo Rossi, and James Cheney (Eds.). ACM, 6:1–6:11. doi:10.1145/3678232.3678236

Santiago Escobar. 2014. Functional Logic Programming in Maude. In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi (Lecture Notes in Computer Science, Vol. 8373)*, Shusaku Iida, José Meseguer, and Kazuhiro Ogata (Eds.). Springer, 315–336. doi:10.1007/978-3-642-54624-2_16

Santiago Escobar. 2018. Multi-paradigm Programming in Maude. In *Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018, Held as a Satellite Event of ETAPS, Thessaloniki, Greece, June 14-15, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11152)*, Vlad Rusu (Ed.). Springer, 26–44. doi:10.1007/978-3-319-99840-4_2

Santiago Escobar, Catherine Meadows, and José Meseguer. 2007. Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures (Lecture Notes in Computer Science, Vol. 5705)*, Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri (Eds.). Springer, 1–50. doi:10.1007/978-3-642-03829-7_1

Santiago Escobar and José Meseguer. 2007. Symbolic Model Checking of Infinite-State Systems Using Narrowing. In *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4533)*, Franz Baader (Ed.). Springer, 153–168. doi:10.1007/978-3-540-73449-9_13

Santiago Escobar and José Meseguer. 2019. Canonical Narrowing with Irreducibility Constraints as a Symbolic Protocol Analysis Method. In *Foundations of Security, Protocols, and Equational Reasoning - Essays Dedicated to Catherine A. Meadows (Lecture Notes in Computer Science, Vol. 11565)*, Joshua D. Guttman, Carl E. Landwehr, José Meseguer, and Dusko Pavlovic (Eds.). Springer, 15–38. doi:10.1007/978-3-030-19052-1_4

Santiago Escobar, José Meseguer, and Ralf Sasse. 2008. Variant Narrowing and Equational Unification. In *Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008 (Electronic Notes in Theoretical Computer Science, Vol. 238)*, Grigore Rosu (Ed.). Elsevier, 103–119. doi:10.1016/J.ENTCS.2009.05.015

Santiago Escobar, José Meseguer, and Prasanna Thati. 2006. Narrowing and Rewriting Logic: from Foundations to Applications. In *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain, November 16-17, 2006 (Electronic Notes in Theoretical Computer Science, Vol. 177)*, Francisco Javier López-Fraguas (Ed.). Elsevier, 5–33. doi:10.1016/J.ENTCS.2007.01.004

Santiago Escobar, Ralf Sasse, and José Meseguer. 2010. Folding Variant Narrowing and Optimal Variant Termination. In *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6381)*, Peter Csaba Ölveczky (Ed.). Springer, 52–68. doi:10.1007/978-3-642-16310-4_5

Santiago Escobar, Ralf Sasse, and José Meseguer. 2012. Folding variant narrowing and optimal variant termination. *J. Log. Algebraic Methods Program.* 81, 7-8 (2012), 898–928. doi:10.1016/J.JLAP.2012.01.002

Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Jean-Pierre Jouannaud and Hélène Kirchner. 1986. Completion of a Set of Rules Modulo a Set of Equations. *SIAM J. Comput.* 15, 4 (1986), 1155–1194.

Raúl López-Rueda and Santiago Escobar. 2022. Canonical Narrowing for Variant-Based Conditional Rewrite Theories. In *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13478)*, Adrián Riesco and Min Zhang (Eds.). Springer, 20–35. doi:10.1007/978-3-031-17244-1_2

Raúl López-Rueda, Santiago Escobar, and José Meseguer. 2022. An Efficient Canonical Narrowing Implementation for Protocol Analysis. In *Rewriting Logic and Its Applications - 14th International Workshop, WRLA@ETAPS 2022, Munich, Germany, April 2-3, 2022, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13252)*, Kyungmin Bae (Ed.). Springer, 151–170. doi:10.1007/978-3-031-12441-9_8

Raúl López-Rueda, Santiago Escobar, and Julia Sapiña. 2023. An efficient canonical narrowing implementation with irreducibility and SMT constraints for generic symbolic protocol analysis. *J. Log. Algebraic Methods Program.* 135 (2023), 100895. doi:10.1016/J.JLAMP.2023.100895

Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 696–701. doi:10.1007/978-3-642-39799-8_48

José Meseguer. 1992. Conditioned Rewriting Logic as a United Model of Concurrency. *Theor. Comput. Sci.* 96, 1 (1992), 73–155. doi:10.1016/0304-3975(92)90182-F

José Meseguer. 1998. Membership algebra as a logical framework for equational specification. In *Recent Trends in Algebraic Development Techniques*, Francesco Parisi Presicce (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–61.

José Meseguer. 2017. Strict coherence of conditional rewriting modulo axioms. *Theor. Comput. Sci.* 672 (2017), 1–35. doi:10.1016/j.tcs.2016.12.026

José Meseguer and Prasanna Thati. 2007. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation* 20, 1-2 (2007), 123–160.

Kazuhiro Ogata and Kokichi Futatsugi. 2013. Compositionally Writing Proof Scores of Invariants in the OTS/CafeOBJ Method. *J. Univers. Comput. Sci.* 19, 6 (2013), 771–804. doi:10.3217/JUCS-019-06-0771

Adrián Riesco and Kazuhiro Ogata. 2018. Prove it! Inferring Formal Proof Scripts from CafeOBJ Proof Scores. *ACM Trans. Softw. Eng. Methodol.* 27, 2 (2018), 6:1–6:32. doi:10.1145/3208951

TeReSe (Ed.). 2003. *Term Rewriting Systems.* Cambridge University Press, Cambridge.

Duong Dinh Tran, Dang Duy Bui, and Kazuhiro Ogata. 2021. Simulation-Based Invariant Verification Technique for the OTS/CafeOBJ Method. *IEEE Access* 9 (2021), 93847–93870. doi:10.1109/ACCESS.2021.3093211

Duong Dinh Tran and Kazuhiro Ogata. 2020. Formal verification of an abstract version of Anderson protocol with CafeOBJ, CiMPA and CiMPG. In *The 32nd International Conference on Software Engineering and Knowledge Engineering, SEKE 2020, KSIR Virtual Conference Center, USA, July 9-19, 2020*, Raúl García-Castro (Ed.). KSI Research Inc., 287–292. doi:10.18293/SEKE2020-064

Duong Dinh Tran and Kazuhiro Ogata. 2022. IPSG: Invariant Proof Score Generator. In *46th IEEE Annual Computers, Software, and Applications Conferenc, COMPSAC 2022, Los Alamitos, CA, USA, June 27 - July 1, 2022*, Hong Va Leong, Sahra Sedigh Sarvestani, Yuuichi Teranishi, Alfredo Cuzzocrea, Hiroki Kashiwazaki, Dave Towey, Ji-Jiang Yang, and Hossain Shahriar (Eds.). IEEE, 1050–1055. doi:10.1109/COMPSAC54236.2022.00164

Formalizing Weighted Pushdown Systems in Isabelle/HOL

Anders Schlichtkrull
Aalborg University
Copenhagen, Denmark
andsch@cs.aau.dk

Morten Konggaard Schou
Aalborg University
Aalborg, Denmark
mksc@cs.aau.dk

ABSTRACT

Pushdown systems are a fundamental formalism in computer science with applications in model checking and program analysis. As a generalization, weighted pushdown systems associate transitions in pushdown systems with weights, thus allowing one to calculate the cost of reaching configurations, where the cost is measured over an algebraic structure. Several model checkers and program analysis tools apply libraries for weighted pushdown system reachability. In this paper, we formalize weighted pushdown systems in Isabelle/HOL. Specifically, we formally prove the correctness of an algorithm for reachability in such systems and extract a verified implementation of the algorithm as a functional program. This requires us to formalize bounded idempotent semirings and sums over countably infinite sets of their elements, as well as saturation procedures that compute these sums. We use differential testing to compare our implementation with a state-of-the-art implementation called PDAAAL. Our testing revealed an error in PDAAAL which we have remedied.

CCS CONCEPTS

• **Theory of computation → Logic and verification**; *Quantitative automata.*

KEYWORDS

Weighted Pushdown System, Isabelle/HOL

ACM Reference Format:
Anders Schlichtkrull and Morten Konggaard Schou. 2025. Formalizing Weighted Pushdown Systems in Isabelle/HOL. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming.* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The classic formalism of pushdown systems extends finite transition systems with an unbounded stack and stack operations on the transitions. Of particular interest is regular sets of configurations, i.e. those sets of configurations that can be represented by a regular language where each member represents a configuration. The possibly infinite set of reachable configurations of a pushdown system from a regular set of configurations is itself a regular set [6] and can be efficiently computed by automata-theoretic algorithms [4, 14, 44].

PPDP '25, September 10–11, 2025, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

Pushdown systems have been used for modeling programs with recursive procedure calls, where the call sites are stored using the stack [44]. More recently, pushdown reachability is also used for verification of MPLS networks [19], in which packet headers contain a stack of labels used for forwarding.

As a natural extension, a weighted pushdown system (WPDS) assigns a weight to each pushdown rule. The weight domain must define an operation for combining the weights of each step along a trace in the system, and an operation for aggregating the weights of different traces. Furthermore, these operations must satisfy the algebraic rules of an idempotent semiring. The weighted generalization of pushdown reachability [39] asks for the aggregated weight of traces through the WPDS between some (sets of) start and end configurations. As an example, the min-plus semiring finds the shortest trace through the pushdown system.

Weighted pushdown systems allow more expressive analysis of programs, in particular interprocedural dataflow analysis [39]; and for network verification, WPDS is used for quantitative MPLS network verification [20].

Given the usage of weighted pushdown systems in the formal analysis and verification of computer programs and networks, it is important to ensure the correctness of both the algorithms and their implementations. It is well known that software often contains bugs, and bugs in model checkers might lead to erroneous results in their analysis and verification of other systems. To increase the trust in WPDS model checking, we formalize WPDSs and their generalized reachability problem in the proof assistant Isabelle/HOL [36], and we prove the correctness of an algorithm that solves this problem. Following the methodology of Schlichtkrull et al. [41], we further verify, in Isabelle/HOL, an executable implementation of the algorithm, and use this as a verified oracle for differential testing of the state-of-the-art C++ implementation PDAAAL [21].

Isabelle/HOL's logic is popularly understood as a combination of logic and typed functional programming. Its notation is also a combination of logical notation and functional programming notation which are both broadly known—we refer to a textbook [35] for an introduction. We include Isabelle/HOL notation directly in the paper, but we will sometimes simplify it to ease the readability.

The formal development corresponding to this submission is available at GitHub.[1]

1.1 Related work

The most similar work is the formalization of unweighted pushdown systems by Schlichtkrull et al. [41, 43]. In contrast to their work, our formalization here considers *weighted* pushdown systems. The work on dynamic pushdown networks by Lammich et al. [25, 26] contains a formalization of dynamic pushdown networks (DPN). In an unpublished technical report, Lammich describes the

[1] https://github.com/anderssch/LTS-formalization/tree/PPDP2025/WPDS

formalization of an executable pre* algorithm. Our work differs from this in that while dynamic pushdown networks are a generalization of pushdown systems, his formalization does not consider the weighted case.

The shortest path problem is essentially a weighted reachability problem. Algorithms for shortest path problems on finite graphs have been formalized in Isabelle/HOL such as Dijkstra's algorithm [27, 28, 37] and the Floyd-Warshall Algorithm [49]. These formalizations are not applicable to solving the reachability problem in our context, because the labelled transition system expressed by a pushdown system is infinite. They also cannot be used for the sub-problem of finding paths from the initial states to the final states of a weighted finite automaton (W-automaton) because we use a formulation of weighted reachability that is based on semiring weights too general for the mentioned formalizations of Dijkstra's algorithm and the Floyd-Warshall algorithm which require linearly ordered weights. Dijkstra's algorithm has also been verified in Mizar [8–10], HOL [15], Coq [33], ACL2 [34], and Key [23].

Model checkers typically have formula languages that are powerful enough to express reachability. In the literature, there is a small number of model checkers formalized in proof assistants, but these are not based on pushdown systems: Sprenger [45] formalized in Coq a checker of finite labelled transition systems for properties expressed in the modal μ-calculus. Esparza et al. [11, 12] formalized in Isabelle/HOL a model checker of programs in a guarded command language (with Booleans as the only datatype) for properties expressed in LTL, building on and improving a previous formalization by Schimpf, Merz and Smaus [40] of the central algorithm for converting LTL formulas to Büchi automata. Wimmer [46–48] and Wimmer and Lammich [50] formalized in Isabelle/HOL a model checker of (networks of) timed automata for properties expressed in a subset of (T)CTL. This includes a formalization of the reachability algorithm for timed automata, central in model checking, which was also formalized in PVS by Xu and Miao [51, 52].

Algorithms for finite automata [3, 5, 22, 29] have also been formalized and so have algorithms for context-free grammars [32, 38].

2 ISABELLE/HOL

Isabelle/HOL is a proof assistant, i.e. a computer program that allows its user to conduct proofs of theorems in computer science, mathematics and logic. The proof assistant checks the correctness of the proofs and can help the user by doing parts of the proofs automatically. Its language can be seen as a mix of typed functional programming and mathematical notation. We shall here introduce the parts of Isabelle/HOL needed to understand the present paper. As an example to illustrate this we will show the formalization of so called *P*-automata (Non-deterministic automata) from the aforementioned formalization of unweighted reachability in Pushdown Automata by Schlichtkrull, Schou, Srba and Traytel [41–43].

Being based on simple type theory the syntax of Isabelle/HOL's language is based on terms belonging to types. We write that term t has type τ as $t :: \tau$. If we for example have term $t_1 :: \tau_1$ and $t_2 :: \tau_2$ then $(t_1, t_2) :: \tau_1 \times \tau_2$ states that the pair (t_1, t_2) belongs to the type of pairs where the first projection is of type τ_1 and the second projection is of type τ_2. In general, types are built from type variables $'a, 'b, \ldots$ and type constructors applied postfix to tuples of types e.g. $('a, 'b)$ *prod*. For some constructors such as e.g. the product type *prod* a more conventional notation $'a \times 'b$ exists. Terms are built from variables $x, y, \ldots$, constants $\mathsf{c}, \mathsf{d}, \ldots$ lambda abstraction $\lambda x.\ t$ and function applications $f\ x$. E.g. given variables x and y we can apply the Pair constructor to construct Pair $x\ y$ which is the pair whose first projection is x and second projection is y. For this specific constructor the more conventional notation (x, y) exists as seen before. The function $\lambda x.\ t$ is the function that takes x and returns the value expressed by t. If x has type $'a$ and t has type $'b$ then $\lambda x.\ t$ belongs to $'a \Rightarrow 'b$, i.e. the type of functions with domain $'a$ and codomain $'b$.

Users of Isabelle/HOL can introduce new types as abbreviations of more complex types such as the *transition* type from [41]:

type_synonym $('state, 'label)$ *transition* = $'state \times 'label \times 'state$

Isabelle/HOL also supports algebraic datatypes. E.g. its library contains the list type:

datatype $'a$ *list* = Nil | Cons $'a$ ($'a$ *list*)

Where Nil :: $'a$ *list* (also written []) is the empty list and Cons :: $'a \Rightarrow 'a$ *list* $\Rightarrow 'a$ *list* is the list constructor. Applications of Cons such as Cons *a as* can also be written *a*#*as*.

Isabelle has a module system called locales. A locale can be seen as a named context consisting of fixed types, fixed constants and a number of assumptions. In the locale's context, the fixed types and constants can be used in types and terms, and its assumptions can be used to prove lemmas and theorems. Consider e.g. the LTS locale from [42] in Figure 1. This particular locale has the name LTS and fixes the constant transition_relation of type $('state, 'label)$ *transition set* based on fixed types $'state$ and $'label$. It does not have any assumptions. The locale defines a new constant trans_star using the **inductive_set** command. This defines trans_star as the least set satisfying the two formulas (terms of type *bool*) separated by | and defined using $\longrightarrow$ (implication). Note that Isabelle requires that the formulas have a specific form that ensures the existence of such a least set. In addition to implication the other usual logical connectives are also available ($\wedge$, $\vee$, $\longleftrightarrow$, $\neg$). One can also make inductive predicates with the command **inductive**.

Locales can also extend other locales. An example is the locale P_Automaton from [43] in Figure 1. This locale extends LTS with an extra type $'init$ representing initial states, a function Init $::'init \Rightarrow 'state$ mapping initial states into states, and a set of final states finals :: $'state\ set$. (Readers may be surprised to see that initial states are defined with a type, and final states with a subset, but this turns out to be convenient.) Notice additionally that the P-automaton locale specifies $'state :: finite$. This is the requirement that the $'state$ type belongs to the *finite* type class which ensures that $'state$ has a finite number of inhabitants. The locale defines constant lang_aut which gives the language of the P-automaton. In a P-automaton the language is a set of pairs where the left projection is an initial state and the right projection is the word read along a path starting in that initial state and ending in a final state. The language is defined using the **definition** command which allows for the definition of non-recursive functions and here introduces the constant lang_aut for the expression on the right. As can be seen Isabelle supports the existential quantifier and set

```
locale LTS =
  fixes transition_relation :: ('state, 'label) transition set
begin
inductive_set trans_star :: ('state × 'label list × 'state) set where
  (p, [], p) ∈ trans_star
| (p, γ, q') ∈ transition_relation ⟶
  (q', w, q) ∈ trans_star ⟶
  (p, γ#w, q) ∈ trans_star
end
locale P_Automaton = LTS transition_relation
  for transition_relation :: ('state::finite, 'label) transition set +
  fixes Init :: ('init::enum) ⇒ 'state
  and finals :: 'state set
begin
definition lang_aut :: ('init × 'label list) set where
  lang_aut = {(p, w) | ∃q ∈ finals. (Init p, w, q) ∈ trans_star}
end
```

Figure 1: The LTS **and** P_Automaton **locales from [42, 43].**

comprehensions. Isabelle/HOL provides also the universal quantification (∀). The function lang_aut has type $('init \times 'label\ list)\ set$. (This demonstrates the advantage of letting *'init* be a type – it ensures that already in the type it is captured that the language consists of pairs where the first projection is an initial state.) One can also use the **primrec** and **fun** commands to make definitions and these allow also for recursion and pattern matching. There are more definitions in both the LTS and P_automaton locale, but the ones shown are sufficient to understand the concepts. In addition to the explicit definitions of the constants inside a locale, Isabelle will automatically introduce parameterized versions outside the locale where each constant fixed by the locale becomes a parameter. Thus in the examples, we will automatically get function LTS.trans_star :: $('state::finite, 'label)\ transition\ set \Rightarrow ('state \times 'label\ list \times 'state)\ set$ as well as function P_Automaton.lang_aut :: $('state::finite, 'label)\ transition\ set \Rightarrow (('init::enum) \Rightarrow 'state) \Rightarrow ('init \times 'label\ list)\ set$.

In the formalization, we are using type classes for the algebraic structures. Like a locale, a type class can have a number of operators and define a set of operations, but in contrast to a locale, it only fixes one type, and in its declaration, the type is thus implicit. A semigroup could therefore be declared as:

```
class semigroup =
  fixes (⊗)
  assumes (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)
```

Since type classes only fix one type, each type class characterizes a class of types, e.g. *semigroup* characterizes those types for which ⊗ is defined and is associative. This also allows for the annotations for type variables with type classes. In a HOL expression we can annotate variables with type variables belonging to type classes e.g. $x :: 'a::semigroup$ and then rely on the existence of the type classes operators and the assumptions.

We use the type of FinFuns [30] which are total functions that are constant except for on a finite set of points. The type $'a \Rightarrow_f 'b$ is the type of such functions from elements of type *'a* to type *'b*.

The FinFun datatype has many useful qualities: (1) In Isabelle's logic FinFuns behave like ordinary functions (though with the syntax $f\ \$\ a$ for the application of FinFun f on the value a), while in Isabelle's code generator they are treated essentially as association lists making code generation work out-of-the-box. (2) Using FinFuns ensures that exactly one weight is associated with each transition. This ensures that if both type *'state* and type *'label* are finite types, then so is type $('state, 'label)\ transition \Rightarrow_f 'weight$. (3) The FinFun datatype comes with an operator $:= for updates where $ts(t\ :=\ d)$ is the FinFun that gives the same weights to all transitions as ts except for t which is given weight d. (4) The FinFun datatype comes with operators for composing FinFuns and functions into FinFuns. We use the composition operators $∘, ∘$ and FinFun_Diag, which can be explained by the following formulas

$$g_1 \circ\$\ f_1 = \text{Abs\_finfun}\ (\lambda x.\ g_1\ (f_1\ \$\ x))$$
$$g_2\ \$\circ\ f_2 = \text{Abs\_finfun}\ (\lambda x.\ g_2\ \$\ (f_2\ x))$$
$$\forall x.\ (\text{finfun\_Diag}\ f_3\ g_3)\ \$\ x = (f_3\ \$\ x,\ g_3\ \$\ x)$$

where

$$g_1 :: 'b \Rightarrow 'c, \quad f_1 :: 'a \Rightarrow_f 'b,$$
$$g_2 :: 'b \Rightarrow_f 'c, \quad f_2 :: 'a \Rightarrow 'b$$
$$f_3 :: 'a \Rightarrow_f 'b, \quad g_3 :: 'a \Rightarrow_f 'c$$

and Abs_finfun h is the conversion of a function h of type $'a \Rightarrow 'b$ to a $'a \Rightarrow_f 'b$ such that $\forall x.\ h\ x = (\text{Abs\_finfun}\ h)\ \$\ x$ under the requirement that h is constant except for on a finite set of points. For the first equation that requirement necessarily holds, but for the second one it depends on f and thus needs to be proved for $∘ to be applicable.

In Isabelle **theorem**s and **lemma**s are HOL expressions of type *bool* that have been formally proved. All proofs in Isabelle are passed through its kernel which is built on a small set of sound inference rules.

3 WEIGHT-GENERALIZED REACHABILITY

Our formalization of weighted pushdown systems is based on a generalization of the labelled transition system (LTS), where the labels form a dioid, i.e. an idempotent semiring. This is an algebraic structure $(D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ over the set D, where

(1) $(D, \oplus, \mathbf{0})$ is a commutative monoid with identity $\mathbf{0} \in D$,
(2) $\oplus$ is idempotent, i.e. $a \oplus a = a$ for all $a \in D$,
(3) $(D, \otimes, \mathbf{1})$ is a monoid with identity $\mathbf{1} \in D$,
(4) $\otimes$ distributes over $\oplus$, i.e. $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$ and $(a \oplus b) \otimes c = a \otimes c \oplus b \otimes c$ for all $a, b, c \in D$, and
(5) $\mathbf{0}$ is an annihilator for $\otimes$, i.e. $\mathbf{0} \otimes a = \mathbf{0} = a \otimes \mathbf{0}$ for all $a \in D$.

Recall that a monoid consists of an associative binary operator and a neutral element that is both the left and right identity of the operator. In a commutative monoid, the binary operator is also commutative.

An example of a dioid is the min-plus semiring over natural numbers extended with infinity: $(\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$. This dioid can be used to represent the length of the shortest trace in an LTS. Another example is the language dioid with union and language concatenation: $(\mathcal{P}(A^*), \cup, \cdot, \emptyset, \{\varepsilon\})$ for some alphabet A.

The $(D, \otimes, \mathbf{1})$ monoid is used for combining the weights of the transitions along a path in the generalized LTS. As an example, the

string monoid (with concatenation and empty string) corresponds to the standard LTS. We define in Isabelle/HOL the reflexive, transitive closure of a monoid-labelled relation r:

inductive_set monoid_rtrancl ($\_^{\odot}$) **where**
  $(a, \mathbf{1}, a) \in r^{\odot}$
| $(a, w, b) \in r^{\odot} \longrightarrow (b, \ell, c) \in r \longrightarrow (a, w \otimes \ell, c) \in r^{\odot}$

Here the first rule labels the reflexive part of the closure with the identity element $\mathbf{1}$, and the second rule inductively defines the weight-generalized transitive closure of r using the $\otimes$ operator to combine weights.

Using Isabelle's module mechanism called locales [2], we fix the relation r of the transition system, and in this context, we define the syntax $c =\ell\Rightarrow^* c'$ to mean $(c, \ell, c') \in r^{\odot}$. These definitions allow us to express that there is a trace through the transition system from c to c' with the weight ℓ.

To formalize the combined weight of all traces from c to c', we use the $(D, \oplus, \mathbf{0})$ monoid. We define the weight-generalized reachability between two sets of states C and C' in the transition system as:

definition weight_reach_set ($\Sigma\langle \_ \Rightarrow^* \_ \rangle$) **where**
  $\Sigma\langle C \Rightarrow^* C' \rangle = \sum\{\ell \mid \exists c \in C.\ \exists c' \in C'.\ c =\ell\Rightarrow^* c'\}$

where intuitively $\sum W$ is the sum, using the $\oplus$ operator, of all elements in W. Since the set of traces through a WPDS may be infinite (though countable), formally defining this $\sum$ in Isabelle/HOL turned out to be a challenge. We address this challenge in Section 3.1.

In weight_reach_set, the reachability is with respect to two sets of states. But we can generalize further and introduce a similar notion on two functions C and C' from states into weights, where the weight $\mathbf{0}$ corresponds to the state not being in the set:

definition weight_reach **where**
  weight_reach $C\ C' = \sum\{(C\ c) \otimes \ell \otimes (C'\ c') \mid c =\ell\Rightarrow^* c'\}$

If we think of the weights as "costs" then we can understand $C\ c$ as a starting cost of c and $C'\ c'$ as an arrival cost in c'.

We additionally define weight_pre_star $C\ c$ as the combined cost of paths starting at c and with C assigning them arrival costs.

definition weight_pre_star **where**
  weight_pre_star $C\ c = \sum\{l \otimes (C\ c') \mid \ell\ c'.\ c =\ell\Rightarrow c'\}$

3.1 Bounded Dioids and Countably Infinite Sums

To guarantee termination of the algorithms for weighted pushdown systems, Reps et al. [39] require the dioid to have a bounded sum, i.e. the partial order defined by $a \leq b$ iff $a \oplus b = a$ must have no infinite descending chains. This requirement is also what enables us to define the sum $\sum W$ over a possibly infinite set W. To formalize this requirement in Isabelle/HOL, we represent chains as functions $f : \mathbb{N} \to D$ from the natural numbers to the weight domain.

We formulate the assumption for the type class *bounded_dioid* (along with the dioid assumptions) as $\nexists f.\ \forall i.\ f\ (i+1) < f\ i$, where the strict order relation is defined by $a < b$ iff $a \oplus b = a \wedge a \neq b$.

For the min-plus dioid over natural numbers, this order is the standard order on numbers, and the requirement holds since any strictly decreasing sequence of natural numbers will eventually reach 0. For the language dioid, the order corresponds to the superset relation, where $f(i) = \{\varepsilon, a\}^i$ is an example of an infinite descending chain (for the superset order). Hence, the language dioid is not bounded.

An algorithm for unbounded dioids is presented in [24]. However, in this paper, we focus on the simpler case of bounded dioids as defined above.

From the assumption that there are no infinite descending chains, we prove that the sum of the elements of a series $f : \mathbb{N} \to D$ converges to a limit L in a finite number of steps:

lemma eventually_stable_sum:
  $\exists L\ N.\ \forall n \geq N.\ f\ 0 \oplus f\ 1 \oplus \cdots \oplus f\ n = L$

In our Isabelle code, the sum is described using Finite_Set.fold. This lemma enables us to use the existing formalization of sums of series from the Isabelle/HOL library. As a technical requirement to work with the formalization of sums of series, we also require the weight domain to be an instance of the type class discrete_topology; however, the requirements of this type class can be trivially satisfied, so it does not limit the applicability of the theory.

The last step for defining the sum $\sum W$ from the previous section, is to convert the countable (but possibly infinite) set W to a series $f : \mathbb{N} \to D$, where $\forall a.\ a \in W \longleftrightarrow \exists i.\ f\ i = a$. For the special case of the empty set, we define $\sum \emptyset = \mathbf{0}$. For non-empty sets, such a series exists when W is countable, and by using Hilbert choice—one of the axioms of Isabelle/HOL—we obtain such a series, on which the sum is defined.

As a useful lemma, we prove that for any countable set of weights W, there is a finite subset $W' \subseteq W$ with the same sum $\sum W' = \sum W$. We use this connection to finite sums to prove various properties of the bounded countable sum, such as the distributivity of $\otimes$ over the sum.

An existing Isabelle/HOL formalization of dioids [1] unfortunately defines the partial order $\leq$ in the reverse direction of [39]. To avoid confusion, we follow the definition in [39] and hence we cannot reuse the dioid formalization in [1].

4 WEIGHTED PUSHDOWN SYSTEMS

After having established the weight domain and generalization of LTS, we can present our formalization of WPDS as a generalization of the PDS theory presented in [41].

A weighted pushdown system (WPDS) over a weight dioid $(D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a tuple (P, Γ, Δ) where P is a finite set of control locations, Γ is a finite stack alphabet, and the set of rules Δ is a finite subset $\Delta \subseteq (P \times \Gamma) \times D \times (P \times \Gamma^*)$. We write a rule as $(p, \gamma)\ -d\hookrightarrow\ (p', w)$ when $((p, \gamma), d, (p', w)) \in \Delta$. Without loss of generality, we assume $|w| \leq 2$, where $w = \varepsilon$ is a pop rule, $|w| = 1$ is a swap rule, and $|w| = 2$ is a push rule.

In the Isabelle/HOL formalization shown in Figure 2, we use type variables *'ctr_loc* and *'label* to represent control locations and stack symbols, and we use the type class *finite* to restrict that only types with finitely many elements can instantiate the type variables. The *'weight* type variable is restricted to the type class *bounded_dioid*, so only types that have been proven to satisfy the assumptions of bounded dioids can be used as the weight domain.

Similar to the formalization in [41], we define a datatype *op* for the pop, swap and push rules, and a function |b| that converts them into a corresponding label list. The keyword **type_synonym** is used to introduce *w_rule* as an alias for the type expression for

```
datatype 'label op = pop | swap 'label | push 'label 'label
type_synonym ('ctr_loc, 'label, 'weight) w_rule =
  ('ctr_loc × 'label) × 'weight × ('ctr_loc × 'label op)
type_synonym ('ctr_loc, 'label) conf =
  'ctr_loc × 'label list
locale WPDS =
  fixes Δ :: ('ctr_loc::finite, 'label::finite,
              'weight::bounded_dioid) w_rule set
begin
primrec lbl where
  lbl pop = [] | lbl (swap γ) = [γ] | lbl (push γ γ') = [γ, γ']
abbreviation is_rule (_ −_↪ _) where
  (p, γ) −d↪ (p', w) ≡ ((p, γ), d, (p', w)) ∈ Δ
inductive_set transition_rel
  (p, γ) −d↪ (p', w) ⟶
  ((p, γ#w'), d, (p', (lbl w)@w')) ∈ transition_rel
interpretation dioidLTS transition_rel .
end
```

Figure 2: Types and locale for weighted pushdown systems

weighted pushdown rules. In the context of the WPDS locale, where we have fixed the set of rules, the is_rule abbreviation allows us to use the rule syntax introduced above.

The semantics of a WPDS is described by a dioid-labelled transition system, where the states in the LTS are configurations $c \in P \times \Gamma^*$ in the WPDS, i.e. pairs of a control location and a stack. For each rule in the WPDS $(p, \gamma) \xrightarrow{d} (p', w)$, there are transitions in the LTS $(p, \gamma w') \overset{d}{\Rightarrow} (p', ww')$ for all $w' \in \Gamma^*$. In Isabelle/HOL, we instantiate the dioidLTS locale with a transition relation defined in this way. In the rest of the paper, transition arrows without subscripts indicating otherwise will refer to transitions in a WPDS. Now the weight-generalized reachability problem of the WPDS is for two sets of configurations C and C' to calculate weight_reach_set C C' where the definition of weight_reach_set is inherited from the dioidLTS locale as presented in Section 3.

4.1 W-automata

Whereas reachability for unweighted pushdown systems is defined between sets of configurations, for WPDS we use a weighted generalization of such a set, which is a map from configurations to weights. Furthermore, in unweighted pushdown systems, we were particularly interested in the regular sets of configurations, i.e. those sets of configurations that can be represented by a regular language where each member represents a configuration. These can be represented by finite automata, and in the weighted case, we can similarly represent the aforementioned maps as so-called W-automata. A W-automaton is essentially a finite automaton in which transitions are not only labelled with the input symbol to be read but also labelled with a weight. In Section 2 we explained the FinFuns, and we shall now use them to model W-automata.

We use the type synonym $('state, 'label)\ transition = 'state \times 'label \times 'state$ for the type of transitions in Isabelle/HOL. In our formalization, a W-automaton is based on three types: $'state$ which is the type of states, $'label$ which is the type of labels and $'weight$ which is the type of the weights. A W-automaton then consists of: (1) A FinFun of the type $('state, 'label)\ transition \Rightarrow_f 'weight$ assigning weights to all transitions. (2) A set of type $'state\ set$ consisting of the final states. In some of the automata in this paper, all states are initial states. For these, there is thus no reason to introduce a set of initial states. For other automata, there will be initial states and non-initial states. For these we will have types $'init$ and $'noninit$ and then make a datatype representing both:

```
datatype ('init, 'noninit) state =
            Init ('init) | Noninit ('noninit)
```

For automata with non-initial states, we will instantiate the $'state$ type variable with $('init, 'noninit)\ state$.

Representing the transitions of a W-automaton as a FinFun is advantageous due to the qualities discussed in Section 2 and additionally this representation means that we can avoid making the needless distinction between edges having weight **0** and being absent from the set of transitions, as we build the FinFun by updates to the constant function (K\$**0**) that has the weight **0** for all transitions.

We introduce a useful variant of \$:= that combines \$:= with an addition:

```
abbreviation finfun_add_update (_(_ $⊕= _)) where
  f(a $⊕= b) = f(a $:= (f $ a) ⊕ b)"
```

While we prefer the FinFun representation of the transitions, the $('state, ('label\ list \times 'weight))\ transition\ set$ representation is also convenient as we shall see and thus we provide a function for making the conversion:

```
definition wts_to_monoidLTS (⟦_⟧)where
  ⟦ts⟧ = {(p, ([γ], d), q) | ts $ (p, γ, q) = d}
```

This set is convenient because its elements are of a monoid type, namely the type $'label\ list \times 'weight$ with monoid operator $\lambda(gs, d)\ (gs', d').\ (gs@gs', d \otimes d')$ where @ is list append and $\otimes$ is the multiplication operator for weights. Note that Isabelle can construct this monoid operator automatically because we have instantiated its monoid class for lists and pairs.

Given the weighted transition relation ts of a W-automaton, we can use $\llbracket ts \rrbracket$ to interpret the monoidLTS locale. This gives us access to the notions defined in the monoidLTS locale. For clarity, we will in this paper label arrows related to transitions in W-automata, such that for $(p, w, d, q) \in \llbracket ts \rrbracket$ the transition is denoted $p \overset{(w,d)}{\Rightarrow}_{\mathsf{Aut}} q$.

Given the set of transitions ts of a W-automaton, we are now able to take its weighted transitive closure as it is shown here: $\llbracket ts \rrbracket^{\odot}$. Its members are tuples $(p, (w, d), q)$ where such a tuple represents the existence of a path through the W-automaton starting in p, ending in q, reading word w and having the weight d.

The following definition is also convenient:

```
definition wts_to_weightLTS (⟦_⟧w) where
  ⟦ts⟧w = {(p, d, q) | ∃γ. ts $ (p, γ, q) = d}
```

As said, a W-automaton will map any configuration (p, w) to a weight. For an automaton with only initial states, we let $'state$ be $'ctr\_loc$ and then the W-automaton maps each (p, w) to the summed weight of all paths that start in state p, read w, and arrive at a final state. We therefore define the language of such a W-automaton as:

```
definition language where
  language ts finals =
    λ(p, w). ∑{d | ∃q ∈ finals. (p, (w, d), q) ∈ ⟦ts⟧^⊙}
```

For a W-automaton with non-initial states we will use the aforementioned $('init, 'noninit)$ *state*, where we for $'init$ take $'ctr\_loc$. Again, each (p, w) will be mapped to the summed weight of all paths that start in p, read w, and arrive at a final state, so we define language_ts0 as:

abbreviation language_ts0 **where**
  language_ts0 = $(\lambda(p, w).$ language (Init $p, w))$

5 PROBLEMS

As advertised we ultimately want to solve the weight-generalized reachability problem. This will be done by a number of problem reductions from a weight-generalized predecessor problem for finite sets of control locations; we explain the involved problems in this section.

The first problem concerns a weighted pushdown system W and a set of control locations *finals*. The problem concerns the combined cost of all paths that start from a configuration (p, w), consume the entire stack w and end in a configuration with an empty stack and a control location in *finals*.

Problem A

Types:
  $'ctr\_loc$: The type used for the control locations.
  $'weight$: The weight type.
  $'label$: The label type.

Input:
  Δ: A finite set of weighted pushdown rules
  finals: A set of control locations

Output:
  The weighted transition relation *ts* of a W-automaton with $'ctr\_loc$ as its type of states, *finals* as its set of final states, and such that for all (p, w):
  (language *ts finals*) (p, w) =
    $\Sigma\langle \{(p, w)\} \Rightarrow^* \{(q, []) \mid q \in finals\} \rangle$

The second problem concerns a WPDS W and a regular set of final configurations. The problem concerns the combined cost of all paths that start from a configuration (p, w). In the problem the set of final configurations is regular and expressed by a W-Automaton.

Problem B

Types:
  $'ctr\_loc$: The type used for the control locations and initial states in the W-automata.
  $'weight$: The weight type.
  $'label$: The label type.
  $('ctr\_loc, 'noninit)$ *state*:
    The type used for the W-automata states.

Input:
  ts: the weighted transition relation of a W-automaton.
  finals: the set of that W-automaton's final states.

Output:
  The weighted transition relation ts' of a W-automaton with type $'ctr\_loc$ for initial states, *finals* as set of final states and such that for all (p, w):
    (language ts' *finals*) (p, w) =
      weight_pre_star (language *ts finals*) (p, w)

The third problem concerns a WPDS W and two regular sets of final configurations. This is the reachability problem we are ultimately interested in, namely the *weight-generalized reachability problem*:

Problem C

Types:
  $'ctr\_loc$: The type used for the control locations and initial states in the P-automata.
  $'weight$: The weight type.
  $'label$: The label type.
  $('ctr\_loc, 'noninit)$ *state*:
    The type used for the P-automata states.

Input:
  ts: The (unweighted) transition relation for a P-automaton.
  ts': The (unweighted) transition relation for a P-automaton.

Output:
  The weight w such that
  $w = \Sigma\langle$ (P_Automaton.lang_aut *ts Init finals*) $\Rightarrow^*$
    (P_Automaton.lang_aut ts' *Init finals'*) $\rangle$

In the following sections, we show the formalized algorithms that solve these problems.

6 SOLVING PROBLEM A

We shall now explain the central Pre* saturation algorithm which solves Problem A.

The Pre* algorithm is a saturation algorithm that takes a set of control locations *finals*. It then builds a W-Automaton by iteratively increasing the weight of transitions in the W-Automaton that fulfill a criterion. The initial W-automaton has type $'ctr\_loc$ as type of states, all states are considered initial states, $\emptyset$ is its transition relation and *finals* is its set of final states.

The algorithm terminates when the W-automaton is saturated, i.e. no more transitions fulfilling the criteria can be added. We formalize a step of the algorithm as a relation, that relates two W-automata if the latter can be obtained from the former by increasing the weight of a transition that fulfills the criterion (i.e. the conjunction of the three assumptions in the following implication):

inductive pre_star_rule **where**
$(p, \gamma)\ -d\hookrightarrow\ (p', w) \longrightarrow$
$(p', (\text{lbl}\ w, d'), q) \in [\![ts]\!]^{\odot} \longrightarrow$
$(ts\ \$\ (p, \gamma, q) \oplus (d \otimes d')) \neq ts\ \$\ (p, \gamma, q) \longrightarrow$
pre_star_rule $ts\ ts((p, \gamma, q)\ \$\oplus{=}\ d \otimes d')$

We are interested in saturation, i.e. the point at which no more steps of the rule can be done. We characterize this with two predicates:

definition saturated **where**
saturated $rule\ val \longleftrightarrow (\nexists val'.\ rule\ val\ val')$

definition saturation **where**
saturation $rule\ val\ val' \longleftrightarrow rule^{**}\ val\ val' \wedge$
saturated $rule\ val'$

Where $rule^{**}$ is the reflexive transitive closure of the relation $rule :: 'a \Rightarrow 'a \Rightarrow \text{bool}$.

We want to prove that the algorithm indeed solves the problem and introduce a useful notion of the soundness of an automaton. We call an automaton A sound, if for each $(p, ([\gamma], d), p') \in [\![A]\!]$ its weight d overestimates $\Sigma\langle \{(p, [\gamma])\} \Rightarrow^* \{(p', [])\} \rangle$:

definition sound **where**
sound $A \longleftrightarrow$
$(\forall p\ p'\ \gamma\ d.\ (p, ([\gamma], d), p') \in [\![A]\!] \longrightarrow$
$d \geq \Sigma\langle \{(p, [\gamma])\} \Rightarrow^* \{(p', [])\} \rangle)$

It is clear that if A does not have any transitions then it is sound. Furthermore, soundness is an invariant of pre_star_rule:

lemma pre_star_rule_sound:
assumes sound A
assumes pre_star_rule $A\ A'$
shows sound A'

As a consequence, when the saturation algorithm has saturated the automaton, the resulting automaton is sound.

This together with some results about soundness gives us the following result:

lemma saturation_pre_star_geq_pred_weight:
assumes saturation pre_star_rule $(K\$\mathbf{0})\ A$
shows language A *finals* $(p, v) \geq$
$\Sigma\langle \{(p, v)\} \Rightarrow^* \{(q, []) \mid q \in finals\} \rangle$

Thus we can see that A overestimates $\Sigma\langle \{(p, v)\} \Rightarrow^* \{(q, []) \mid q \in finals\} \rangle$ for any (p, v). What we have so far is two components of a classical invariant proof namely that the invariant is initially satisfied and that it is preserved by the algorithm. The last thing we need to establish is a property of the termination that together with the invariant ensures correctness. This property is that A also underestimates the weights:

lemma saturation_pre_star_leq_pred_weight:
assumes saturation pre_star_rule $(K\$\mathbf{0})\ A$
shows language A *finals* $(p, v) \leq$
$\Sigma\langle \{(p, v)\} \Rightarrow^* \{(q, []) \mid q \in finals\} \rangle$

From this clearly follow correctness:

theorem correctness:
assumes saturation pre_star_rule $(K\$\mathbf{0})\ A$
shows language A *finals* $(p, v) =$
$\Sigma\langle \{(p, v)\} \Rightarrow^* \{(q, []) \mid q \in finals\} \rangle$

7 SOLVING PROBLEM B

We shall now explain the algorithm for solving Problem B.

locale WPDS_with_W_automata = WPDS Δ
for
$\Delta :: ('ctr\_loc::finite, 'label::finite, 'weight::bounded\_dioid)$
$w\_rule\ set$
and $ts0 :: (('ctr\_loc, 'noninit :: finite)\ state, 'label, 'weight)$
$w\_transitions$
+ **assumes** $\forall p\ \gamma\ q.\ \text{is\_Init}\ q \longrightarrow ts0\ \$\ (p, \gamma, q) = \mathbf{0}$
assumes finite Δ

The first assumption states that there are no transitions into initial states in the W-automaton – or more precisely that such transitions have weight $\mathbf{0}$. This is without loss of generality, as an automaton with such transitions can be transformed to an automaton without them that has the same language; however, we have not formalized this fact.

The problem is solved by a reduction to the weight-generalized predecessor problem for finite sets of control locations, and thus the algorithm will implement the reduction and then use the Pre* algorithm to solve the reduced problem.

The reduction is from [39] and we formalize it here. We explain the idea of this reduction. The reduction takes as input the transition relation *ts0* of a W-automaton, its set of final states *finals* and a WPDS. The states of this W-automaton are of type $('ctr\_loc :: finite, 'noninit :: finite)\ state$.

The reduction builds a new WPDS, which instead of the type $'ctr\_loc$, has the type $('ctr\_loc, 'noninit)\ state$ for control locations. The reduction lifts all WPDS rules from the original WPDS into this type by using the Init constructor:

definition init_rules **where**
init_rules =
$\{((\text{Init}\ p, \gamma), d, (\text{Init}\ p', w)) \mid (p, \gamma)\ -d\hookrightarrow\ (p', w)\}$

Additionally the reduction introduces some new rules that code the W-automaton into the WPDS:

definition pop_ts0_rules **where**
pop_ts0_rules = $\{((p, \gamma), d, (q, \text{pop})) \mid ts0\ \$\ (p, \gamma, q) = d\}$

We collect these rules in a single set of rules:

definition Δ_{ts0} **where**
$\Delta_{ts0} = \text{init\_rules} \cup \text{pop\_ts0\_rules}$

We introduce the abbreviation $\text{pre\_star\_rule}_{ts0}$ for the version of pre_star_rule based on the rules in Δ_{ts0}:

abbreviation $\text{pre\_star\_rule}_{ts0} = \text{WPDS.pre\_star\_rule}\ \Delta_{ts0}$

We can now show correctness:

lemma pre_star_correctness':
assumes saturation $\text{pre\_star\_rule}_{ts0}\ (K\$\mathbf{0})\ A$
shows (language A) (Init p, w) =
weight_pre_star language_ts0 (p, w)

7.1 Comparison

The paper by Reps et al. [39] that our work builds on generalizes the unweighted case, which was initially explored by Büchi [6]. Therefore, our algorithm and the proof of its correctness are variants of the algorithm and proof presented by Reps et al. Our theory does, however, differ from theirs in several ways in order to accommodate

formalization. We shall here explain the approach of Reps et al. [39], contrast it with our approach, and argue for our decisions. In their paper, Reps et al. explain the coding of W-automata into WPDS that we have formalized above. They then show essentially three approaches to solving Problem B:

(i) by encoding to an abstract grammar problem (a generalization of context-free grammar productions)
   - This codes the WPDS and the W-automaton into an abstract grammar problem. The encoding of the W-automaton is similar to its encoding into a WPDS.

(ii) a saturation procedure that starts from the W-automaton given as input in Problem B, and which in each step will either add a transition or increase weights of already existing transitions.
   - As the initial W-automaton in the saturation process is the one from the problem formulation, it is not coded into the WPDS. The procedure is based on a saturation rule.

(iii) an imperative implementation of the saturation process
   - As this implements the saturation process it also starts from the W-automaton provided in the problem and does not code it into the WPDS.

Our approach differs from this in three ways:

- We do not consider abstract grammar problems at all.
- Like Reps et al. we use the coding of W-automata into WPDS. But in our definition of the saturation procedure, we assume that the initial W-automata is already coded into the WPDS, which allows our saturation procedure to solve Problem A instead of Problem B.
- Instead of an imperative implementation, we provide a functional implementation.

We do not consider abstract grammar problems because this would, in some sense, be a detour from our goal of having a formalized executable implementation—for that to become executable, we would also need to formalize and prove correct an algorithm for solving abstract grammar problems. In the paper by Reps et al. they are building on non-formalized mathematics and can thus refer to results about abstract grammar problems without such concerns, and from a didactic perspective, it also helps the reader understand their work, but it would not help our formalization.

The saturation rule pre_star_rule we have is similar to that of Reps et al. but there are differences. Firstly, our use of FinFun ensures that we do not have the distinction between transitions having weight **0** and transitions not occurring in the W-automata. This simplifies the saturation rule and also means that in our proofs, we do not need to handle these equivalent cases separately. Additionally, our saturation procedure solves Problem A, because we are relying on the W-automaton from Problem B's input being coded into the WPDS. An advantage of this approach is that during the saturation procedure, the built W-automaton preserves a very intuitive invariant: each transition in the W-automaton has an over-approximation of a weight corresponding to a specific path through the WPDS. Had we not relied on the encoding, this invariant would also have to take into account the W-automaton given in Problem B.

8 SOLVING PROBLEM C

In this section we will explain how we formalized an algorithm for solving Problem C. This is again done by a problem reduction, this time to the weight-generalized predecessor problem for regular sets of configurations. The reduction crucially relies on two calculations. Namely a weighted generalization of the famous intersection automaton construction and an algorithm for summing up all "accepting paths" through a graph.

8.1 Intersection algorithm

We explain here the *intersection algorithm*. The intersection algorithm takes two W-automata, *ts1* and *ts2* and creates a product W-automaton that simulate the "simultaneous" run of *ts1* and *ts2*. In the product W-automaton the transitions will be of the form $((p1, q1), (\gamma, d), (p2, q2))$. We lift first and second projections from product states to production transitions:

fun fst_trans **where**
  fst_trans $((p1, q1), \gamma, (p2, q2)) = (p1, \gamma, p2)$

fun snd_trans **where**
  snd_trans $((p1, q1), \gamma, (p2, q2)) = (q1, \gamma, q2)$

We then define FinFuns that map each $((p1, q1), (\gamma, d), (p2, q2))$ to respectively the weight in *ts1* of its first projection in and weight in *ts2* of its second projection.

definition fst_weight **where**
  fst_weight *ts1* = *ts1* $\$\circ$ fst_trans

definition snd_weight **where**
  snd_weight *ts2* = *ts2* $\$\circ$ snd_trans

After this we can build the FinFun mapping each product transition to the pair consisting of these two weights:

definition pair_weight **where**
  pair_weight *ts1 ts2* =
    finfun_Diag (fst_weight *ts1*) (snd_weight *ts2*)

And lastly we can multiply these weights:

definition w_inters (_ $\cap_w$ _) **where**
  $ts_1 \cap_w ts_2 = (\lambda(d, d').\ d \otimes d') \circ\$$ (pair_weight ts_1 ts_2)

The function w_inters thus does exactly the desired product construction. Given two sets of final states *finals1* and *finals2*, the set of final states in the product automaton is simply *finals1*×*finals2*. We expect the W-automata to have the same sets of initial states *init* and then the initial states in the product are $\{(p, p) \mid p \in init\}$. Consider e.g. the automata in Figure 3 and Figure 4 with the intersection automaton shown in Figure 5.

What remains to be proved is that paths through the automaton have the desired properties. We prove the construction correct and complete:

lemma w_inters_sound_wts_to_monoidLTS:
  assumes binary_aut *ts1*
  assumes $((p1, q1), d, (p2, q2)) \in [\![ts1 \cap_w ts2]\!]^{\odot}_w$
  shows $\exists w\ dp\ dq.$
    $(p1, (w, dp), p2) \in [\![ts1]\!]^{\odot} \wedge$
    $(q1, (w, dq), q2) \in [\![ts2]\!]^{\odot} \wedge$
    $dp \otimes dq = d$

lemma w_inters_complete_wts_to_weightLTS:
assumes binary_aut *ts1*
assumes $(p1, (w, dp), p2) \in [\![ts1]\!]^{\odot}$
assumes $(q1, (w, dq), q2) \in [\![ts2]\!]^{\odot}$
shows $((p1, q1), dp \otimes dq, (p2, q2)) \in [\![ts1 \cap_w ts2]\!]^{\odot}_w$

The lemma has the additional requirement that *ts1* is a binary W-automaton, i.e. all transitions have weight **0** or **1**. We now demonstrate why this is required by an example. Consider again the automata in Figure 3 and Figure 4 with the intersection automaton shown in Figure 5. In $\mathrm{ts}_{\mathrm{ex1}}$ we can find the following path:

$$p_2 \ =x, d_{23}\Rightarrow p_3 \ =x, d_{34}\Rightarrow p_4 \ =y, d_{45}\Rightarrow p_5$$

In $\mathrm{ts}_{\mathrm{ex2}}$ we can find the following path:

$$p_2 \ =x, d_{22}\Rightarrow p_2 \ =x, d_{22}\Rightarrow p_2 \ =y, d_{26}\Rightarrow p_6$$

The product of the weights of these paths is

$$d_{23} \otimes d_{34} \otimes d_{45} \otimes d_{22} \otimes d_{22} \otimes d_{26}$$

and they read both the word xxy. The product automaton can simulate these two runs:

$$(p_2, p_2) \ =x, d_{23} \otimes d_{22}\Rightarrow (p_3, p_2) \ =x, d_{34} \otimes d_{22}\Rightarrow$$
$$(p_4, p_2) \ =y, d_{45} \otimes d_{26}\Rightarrow (p_5, p_6)$$

and the path's weight is

$$d_{23} \otimes d_{22} \otimes d_{34} \otimes d_{22} \otimes d_{45} \otimes d_{26}.$$

The problem is that in general it may be that the two weighs $d_{23} \otimes d_{34} \otimes d_{45} \otimes d_{22} \otimes d_{22} \otimes d_{26}$ and $d_{23} \otimes d_{22} \otimes d_{34} \otimes d_{22} \otimes d_{45} \otimes d_{26}$ are different. The reason is that we are not assuming commutativity. However, in the case where $\mathrm{ts}_{\mathrm{ex1}}$ is binary the two weights are equal, because then either one of d_{23}, d_{34} and d_{45} is **0** and then

$$d_{23} \otimes d_{34} \otimes d_{45} \otimes d_{22} \otimes d_{22} \otimes d_{26} = \mathbf{0} = d_{23} \otimes d_{22} \otimes d_{34} \otimes d_{22} \otimes d_{45} \otimes d_{26}$$

or all of d_{23}, d_{34} and d_{45} are **1** and then

$$d_{23} \otimes d_{34} \otimes d_{45} \otimes d_{22} \otimes d_{22} \otimes d_{26} =$$
$$\mathbf{1} \otimes \mathbf{1} \otimes \mathbf{1} \otimes d_{22} \otimes d_{22} \otimes d_{26} =$$
$$d_{22} \otimes d_{22} \otimes d_{26} =$$
$$\mathbf{1} \otimes d_{22} \otimes \mathbf{1} \otimes d_{22} \otimes \mathbf{1} \otimes d_{26} =$$
$$d_{23} \otimes d_{22} \otimes d_{34} \otimes d_{22} \otimes d_{45} \otimes d_{26}.$$

As we shall see, when we apply the intersection algorithm to solve the weight-generalized reachability problem we will indeed apply it to a W-automaton and a binary W-automaton, and thus we can make this assumption.

8.2 Automaton path sum algorithm

We explain in this section some algorithms that calculate the sum of the weights of multiple paths through a W-automaton. The first algorithm will take as input a Finfun S which assigns initial weights to all configurations and also take the transition relation transition_relation of a W-automaton. To any path $c =\ell\Rightarrow^* c'$ through transition_relation we can associate the weight $S\ \$\ c \otimes \ell$ i.e. the initial weight of c in S multiplied by the cost of reaching c'. We will then for a fixed c' calculate the summed weight over all such paths for all c, i.e. $\sum\{S\ \$\ c \otimes \ell \mid c =\ell\Rightarrow^*_{\mathrm{Aut}} c'\}$. And in fact, we will calculate this for each c' and store the result in a FinFun.

The algorithm is a saturation algorithm. The algorithm maintains the invariant that it maps each state to the sum of the aforementioned weight for a subset of all paths from initials into that state. The algorithm starts from S and in each step makes an update based on some transition in the W-automaton. This is captured by the following rule:

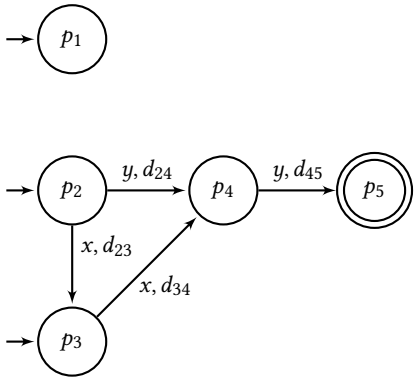


Figure 3: A W-automaton $\mathrm{ts}_{\mathrm{ex1}}$.

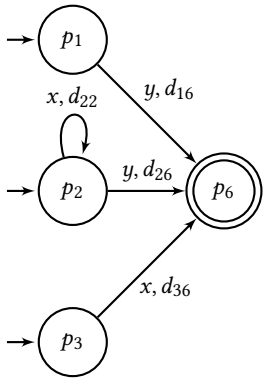


Figure 4: A W-automaton $\mathrm{ts}_{\mathrm{ex2}}$.

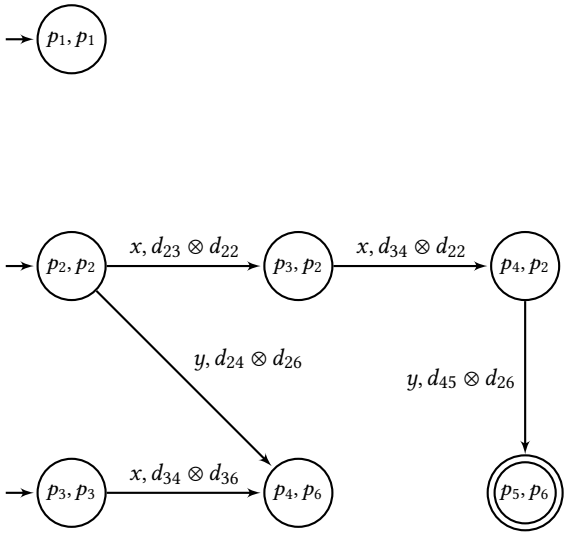


Figure 5: The product automaton $\mathrm{ts}_{\mathrm{ex1}} \cap_w \mathrm{ts}_{\mathrm{ex2}}$. States not reachable from initial states are omitted in this figure.

inductive weight_reach_rule **where**
$(p, d, q) \in$ transition_relation $\longrightarrow$
$S \$ q \oplus (S \$ p \otimes d) \neq S \$ q \longrightarrow$
weight_reach_rule S $S(q \$\oplus= S \$ p \otimes d)$

The first assumption picks a transition from the transition relation of the W-automaton, and the second assumption ensures that the potential update will actually have an effect.

We prove the correctness of the algorithm:

theorem weight_reach_saturation_correct:
assumes saturation weight_reach_rule S S'
shows $S' \$ c' = \sum\{S \$ c \otimes \ell \mid c =\ell\Rightarrow^*_{\mathsf{Aut}} c'\}$

This theorem simply states that the result S' of the algorithm indeed maps each configuration c' to the expected weight. However, this algorithm is not exactly what we need. What we are instead interested in is an algorithm that takes as inputs a Finfun S which assigns "initial" weights to all configurations, a Finfun S'' which assigns "final" weights to all configurations and also take the transition relation transition_relation of a W-automaton. Then we simply want to calculate weight_reach (($) S) (($) S'') where ($) S is the conversion of S from a $'state \Rightarrow_{\mathsf{f}} 'weight$ to a $'state \Rightarrow 'weight$.

The algorithm for this simply saturates S using weight_reach_rule and then calculates a sum over all states. This can be done because a W-Automaton has finitely many states.

theorem weight_reach_saturation_sum_correct:
assumes saturation weight_reach_rule S S'
shows $\sum\{(S' \$ c') \otimes (S'' \$ c') \mid c'.\ \mathsf{True}\} =$
weight_reach (($) S) (($) S'')

The theorem says that if we start from S and run the saturation algorithm resulting in S', then by doing a finite number of multiplications and a finite sum we can calculate the weighted reachability from ($) S to ($) S'. The reason for the finiteness is simply that the automaton has finitely many states.

8.3 Combining the algorithms

We will now present the solution to Problem C. As promised, it works by combining the algorithms from the previous sections. Firstly, we introduce a predicate representing the encoding of a W-automaton into the WPDS (as explained in Section 7) combined with the application of the saturation algorithm (explained in Section 6):

definition pre_star_saturation **where**
pre_star_saturation $ts0$ $ts0_{sat}$ =
saturation
(WPDS_with_W_automata.pre_star_rule' Δ $ts0$)
($K\$0$)
$ts0_{sat}$

Then the solution and its correctness are formalized in the following lemma, where the three assumptions and the left-hand side of the proven equation formalize how the algorithms are combined, and the right-hand side of the proven equation expresses the weight-generalized reachability problem.

theorem pre_star_correctness_full:
assumes $\forall p\ \gamma\ q.$ is_Init $q \longrightarrow (p, \gamma, q) \notin ts'$
assumes pre_star_saturation Δ (ts_to_wts ts') ts'_{sat}
assumes $prod\_ts$ = (ts_to_wts ts) $\cap_{\mathsf{w}}$ ts'_{sat}
assumes $prod\_fins = finals \times finals'$
shows
$\sum\{d.\ \exists p\ w.$
$d =$ dioidLTS.language $prod\_ts$ $prod\_fins$ $((p, p), w)$
$\wedge$ is_Init $p\} =$
$\Sigma\langle$ P_Automaton.lang_aut ts Init $finals \Rightarrow^*$
P_Automaton.lang_aut ts' Init $finals' \rangle$

We have already seen the first of the assumptions in the locale WPDS_with_W_automata, namely that initially in ts' there should be no transitions into initial states. The second to fourth assumptions can be seen as steps in an algorithm. The second assumption represents the encoding of ts' first into a W-automaton and then into Δ followed by the application of the saturation algorithm on the resulting WPDS. This results in the automaton ts'_{sat}. The third assumption defines $prod\_ts$ as the intersection automaton using the algorithm from Section 8.1. The fourth assumption defines $prod\_finals$ as the final states of the intersection automaton. The left-hand side of the proven equation represents calculating the sum over all paths for which we gave an algorithm in Section 8.2. And lastly the right-hand side of the equation is the specification of the problem.

9 EXECUTABLE CODE

As stated in Isabelle's code generation tutorial [17], the **datatype** and **definition**/**primrec**/**fun** declarations form the core of a functional programming language. Isabelle's code generator [17, 18] builds on this by translating Isabelle functions based on these to functional languages (SML, OCaml, Haskell, Scala). Thus, obtaining verified executable implementations from Isabelle can be done by building functions and types from these commands. This is the most straightforward way to obtain executable implementations from Isabelle/HOL. In fact, Isabelle's **value** command relies on this: the given expression is converted to a functional program that is then executed. While it is possible to verify also imperative code in Isabelle using e.g. the Imperative HOL framework [7] doing that is more involved and a larger undertaking.

Note that not all definitions and function declarations in Isabelle/HOL can be used for code generation. In particular, definitions that use the Hilbert choice operator, which obtains a value satisfying a given predicate if such a value exists, need to be accompanied by a constructive algorithm that calculates this value. By proving the equivalence between the (often simpler) definition and the constructive algorithm, verified code can be generated.

The main result of this paper is the definition of an executable function run_WPDS_reach that we prove to be equal to the non-constructive definition of weight_reach_set for a WPDS:

theorem WPDS_reach_exec_correct:
assumes $\forall p\ \gamma\ q.$ is_Init $q \longrightarrow (p, \gamma, q) \notin ts'$
shows run_WPDS_reach Δ ts ts' $finals$ $finals'$ =
Some ($\Sigma\langle$ P_Automaton.lang_aut ts Init $finals \Rightarrow^*$
P_Automaton.lang_aut ts' Init $finals' \rangle$)

The result of run_WPDS_reach is wrapped in the optional type, since the function starts by checking the assumption in the above theorem and returns the value None if it is not satisfied.

Solving the weight-generalized reachability problem for a WPDS involves two saturation procedures: one over the WPDS using pre_star_rule and another over the product W-automaton using

weight_reach_rule. To handle both cases, we formalize a framework for implementing weighted saturation procedures. Given an executable step function over a bounded dioid, the executable saturation function is implemented using while_option, a functional version of a while loop. For a predicate b, function c and state s, while_option b c s computes the state (wrapped in the optional type) Some $(c \ldots (c\ (c\ s)))$ not satisfying b with the minimal number of applications of c, or None if no such state exists. From the properties of bounded dioids, we prove that the saturation will eventually terminate; hence, we can extract the resulting state from the optional type. Moreover, we prove that this implementation corresponds to the saturation procedure for a saturation rule that takes one step at a time.

```
locale step_saturation =
  fixes step :: 'a::bounded_dioid ⇒ 'a
  assumes step s ≤ s
begin
definition
  saturation_exec = the ∘ while_option (λs. step s ≠ s) step
definition step_rule s s' ≡ step s = s' ∧ s ≠ s'
lemma saturation_step_exec:
  saturation step_rule s (saturation_exec s)
end
```

The implementations of the step function for both pre_star and weight_reach do not correspond one-to-one to the respective saturation rules. They compute a set of currently applicable updates to the state and apply them all in one step. We prove, for the more general case of a step function that adds the sum of a non-empty subset of applicable rule updates to the state, that the algorithm will result in the same saturated state, even though the intermediate states of the saturation may differ.

Interestingly, this framework enables us to formalize and prove the correctness of a parallel version of the pre_star algorithm that partitions the set of WPDS rules and runs saturation in parallel, where in each step, the parallel processes run a saturation with the rules in the corresponding partition, and then the resulting W-automata are summed up to a combined W-automaton, which is used for the next step. These large parallel steps are then repeated until saturation. This algorithm can be implemented in a map-reduce framework; however, the practical speed-up compared to the single-threaded algorithm will depend on the structure of the WPDS, i.e., how well the WPDS rules can be split into partitions with a low coupling between partitions.

10 DIFFERENTIAL TESTING

As an application of our formalization, we use the executable code as a verified oracle for differential testing of the C++ library PDAAAL [21], a state-of-the-art library for weighted pushdown systems. Differential testing [13, 16, 31] compares the output of executing different implementations of an algorithm on the same input problems. When results differ between the implementations, this indicates a bug in one of them. We use a formally verified oracle in the differential testing to find and fix bugs in the more efficient C++ implementation, and hence increase the trust in the correctness of the implementation. The advantage of using a verified implementation for differential testing is that for all problem instances, where the tools agree on the answer, it can be trusted to be correct, whereas multiple unverified implementations can agree on the same wrong answer.

Schlichtkrull et al. [41] applied this technique for pushdown system reachability (without weights) with a formally verified oracle and discovered three bugs in the C++ implementation PDAAAL. We extend this work by performing differential testing of the weighted pushdown system algorithms that is implemented in PDAAAL against our executable Isabelle/HOL formalization.

We run two sets of tests. In the first set of tests, we perform differential testing on numerous randomly generated instances of weighted pushdown systems. This differential testing against our Isabelle/HOL formalization led to the discovery of a bug in PDAAALs implementation of pre*, where a data structure that has a weight value for each state in the W-automaton was initialized to a value corresponding to $\mathbf{0}$ in the dioid for all states, but specifically for the accepting states in the W-automaton it should be initialized to the $\mathbf{1}$ value from the dioid. We have corrected this error in the source code of PDAAAL.

In the second set of tests, we exhaustively generate all (1177) WPDS with up to two states and two labels using the integer weights 1 and 2. These are combined with the corresponding (468) initial W-automata with up to four states and the (148) final W-automata with up to three states, resulting in just over 80 million problem instances for weighted pushdown reachability. For all the exhaustive small problem instances, the differential testing succeeded.

11 CONCLUSION

With the present paper we have contributed a formalization of a weight-generalized reachability algorithm for weighted pushdown systems including an executable implementation. It required us to formalize two saturation algorithms (pre* and sums over paths in a weighted automaton) as well as an encoding of W-automata into a WPDS and an automaton intersection algorithm. Differential testing revealed an error in PDAAAL which we remedied. Other works have also done differential testing between verified tools and the corresponding state-of-the-art tools. Experience has shown that it is possible to find errors with this methodology and the present paper provides further corroboration of this.

ACKNOWLEDGMENTS

This work was supported by the S4OS Villum Investigator Grant nr. 37819 from Villum Fonden.

REFERENCES

[1] Alasdair Armstrong, Georg Struth, and Tjark Weber. 2013. Kleene Algebra. *Archive of Formal Proofs* (January 2013). https://isa-afp.org/entries/Kleene_Algebra.html, Formal proof development.
[2] Clemens Ballarin. 2014. Locales: A Module System for Mathematical Theories. *J. Autom. Reason.* 52, 2 (2014), 123–153. https://doi.org/10.1007/s10817-013-9284-7
[3] Stefan Berghofer and Markus Reiter. 2009. Formalizing the Logic-Automaton Connection. In *TPHOLs 2009 (LNCS, Vol. 5674)*. Springer, 147–163. https://doi.org/10.1007/978-3-642-03359-9_12
[4] Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *CONCUR 1997 (LNCS, Vol. 1243)*. Springer, 135–150. https://doi.org/10.1007/3-540-63141-0_10
[5] Thomas Braibant and Damien Pous. 2012. Deciding Kleene Algebras in Coq. *Log. Methods Comput. Sci.* 8, 1 (2012). https://doi.org/10.2168/LMCS-8(1:16)2012
[6] J Richard Büchi. 1964. Regular canonical systems. *Archiv für mathematische Logik und Grundlagenforschung* 6, 3-4 (1964), 91–111. https://doi.org/10.1007/

BF01969548
[7] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. 2008. Imperative Functional Programming with Isabelle/HOL. In *TPHOLs 2008 (LNCS, Vol. 5170)*. Springer, 134–149. https://doi.org/10.1007/978-3-540-71067-7_14
[8] Jing-Chao Chen. 2003. Dijkstra's Shortest Path Algorithm. *Formaliz. Math.* 11, 3 (2003), 237–247. https://fm.mizar.org/2003-11/fm11-3.html
[9] Jing-Chao Chen. 2003. Dijkstra's shortest path algorithm. *Journal of formalized mathematics* 15, 9 (2003), 237–247.
[10] Jing-Chao Chen. 2006. Formalization of Dijkstra's Shortest Path Algorithm. *Mechanized Mathematics and Its Applications* 5, 1 (2006), 237–247.
[11] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. 2013. A Fully Verified Executable LTL Model Checker. In *CAV 2013 (LNCS, Vol. 8044)*. Springer, 463–478. https://doi.org/10.1007/978-3-642-39799-8_31
[12] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. 2014. A Fully Verified Executable LTL Model Checker. *Archive of Formal Proofs* (May 2014). https://isa-afp.org/entries/CAVA_LTL_Modelchecker.html, Formal proof development.
[13] Robert B. Evans and Alberto Savoia. 2007. Differential testing: a new approach to change detection. In *ESEC-FSE 2007*. ACM, 549–552. https://doi.org/10.1145/1287624.1287707
[14] Alain Finkel, Bernard Willems, and Pierre Wolper. 1997. A Direct Symbolic Approach to Model Checking Pushdown Systems. In *INFINITY 1997 (ENTCS, Vol. 9)*. Elsevier, 27–37.
[15] Michael J. C. Gordon, Joe Hurd, and Konrad Slind. 2003. Executing the Formal Semantics of the Accellera Property Specification Language by Mechanised Theorem Proving. In *CHARME 2003 (LNCS, Vol. 2860)*. Springer, 200–215. https://doi.org/10.1007/978-3-540-39724-3_19
[16] Alex Groce, Gerard J. Holzmann, and Rajeev Joshi. 2007. Randomized Differential Testing as a Prelude to Formal Verification. In *ICSE 2007*. IEEE Computer Society, 621–631. https://doi.org/10.1109/ICSE.2007.68
[17] Florian Haftmann and Lukas Bulwahn. 2025. Code generation from Isabelle/HOL theories. *Part of the Isabelle documentation: https://isabelle.in.tum.de/doc/classes.pdf* (2025).
[18] Florian Haftmann and Tobias Nipkow. 2010. Code Generation via Higher-Order Rewrite Systems. In *FLOPS 2010 (LNCS, Vol. 6009)*. Springer, 103–117. https://doi.org/10.1007/978-3-642-12251-4_9
[19] Jesper Stenbjerg Jensen, Troels Beck Krøgh, Jonas Sand Madsen, Stefan Schmid, Jiří Srba, and Marc Tom Thorgersen. 2018. P-Rex: Fast Verification of MPLS Networks with Multiple Link Failures. In *CoNEXT*. ACM, 217–227. https://doi.org/10.1145/3281411.3281432
[20] Peter Gjøl Jensen, Dan Kristiansen, Stefan Schmid, Morten Konggaard Schou, Bernhard Clemens Schrenk, and Jirí Srba. 2020. AalWiNes: a fast and quantitative what-if analysis tool for MPLS networks. In *CoNEXT 2020*. ACM, 474–481. https://doi.org/10.1145/3386367.3431308
[21] Peter G. Jensen, Stefan Schmid, Morten K. Schou, and Jiří Srba. 2022. PDAAAL: A Library for Reachability Analysis of Weighted Pushdown Systems. In *ATVA 2022 (LNCS, Vol. 13505)*. Springer, 225–230. https://doi.org/10.1007/978-3-031-19992-9_14
[22] Dongchen Jiang and Wei Li. 2018. The verification of conversion algorithms between finite automata. *Sci. China Inf. Sci.* 61, 2 (2018), 028101:1–028101:3. https://doi.org/10.1007/s11432-017-9155-x
[23] Volker Klasen. 2010. *Verifying Dijkstra's Algorithm with KeY*. Diplomarbeit. University of Koblenz and Landau. https://d-nb.info/1110894511/34
[24] M. Kühnrich, S. Schwoon, J. Srba, and S. Kiefer. 2009. Interprocedural Dataflow Analysis over Weight Domains with Infinite Descending Chains. In *FOSSACS'09 (LNCS, Vol. 5504)*. Springer-Verlag, 440–455.
[25] Peter Lammich. 2009. Formalization of Dynamic Pushdown Networks in Isabelle/HOL. https://www21.in.tum.de/~lammich/isabelle/dpn-document.pdf
[26] Peter Lammich, Markus Müller-Olm, and Alexander Wenner. 2009. Predecessor Sets of Dynamic Pushdown Networks with Tree-Regular Constraints. In *CAV 2009 (LNCS, Vol. 5643)*. Springer, 525–539. https://doi.org/10.1007/978-3-642-02658-4_39
[27] Peter Lammich and Tobias Nipkow. 2019. Proof Pearl: Purely Functional, Simple and Efficient Priority Search Trees and Applications to Prim and Dijkstra. In *ITP 2019 (LIPIcs, Vol. 141)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:18. https://doi.org/10.4230/LIPICS.ITP.2019.23
[28] Peter Lammich and Tobias Nipkow. 2019. Purely Functional, Simple, and Efficient Implementation of Prim and Dijkstra. *Archive of Formal Proofs* (June 2019). https://isa-afp.org/entries/Prim_Dijkstra_Simple.html, Formal proof development.
[29] Peter Lammich and Thomas Tuerk. 2012. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *ITP 2012 (LNCS, Vol. 7406)*. Springer, 166–182. https://doi.org/10.1007/978-3-642-32347-8_12
[30] Andreas Lochbihler. 2009. Formalising FinFuns - Generating Code for Functions as Data from Isabelle/HOL. In *TPHOLs 2009 (LNCS, Vol. 5674)*. Springer, 310–326. https://doi.org/10.1007/978-3-642-03359-9_22
[31] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107.
[32] Yasuhiko Minamide. 2007. Verified Decision Procedures on Context-Free Grammars. In *TPHOLs 2007 (LNCS, Vol. 4732)*. Springer, 173–188. https://doi.org/10.1007/978-3-540-74591-4_14
[33] Anshuman Mohan, Wei Xiang Leow, and Aquinas Hobor. 2021. Functional Correctness of C Implementations of Dijkstra's, Kruskal's, and Prim's Algorithms. In *CAV 2021 (LNCS, Vol. 12760)*. Springer, 801–826. https://doi.org/10.1007/978-3-030-81688-9_37
[34] J Strother Moore and Qiang Zhang. 2005. Proof Pearl: Dijkstra's Shortest Path Algorithm Verified with ACL2. In *TPHOLs 2005 (LNCS, Vol. 3603)*. Springer, 373–384. https://doi.org/10.1007/11541868_24
[35] Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics - With Isabelle/HOL*. Springer. https://doi.org/10.1007/978-3-319-10542-0
[36] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9
[37] Benedikt Nordhoff and Peter Lammich. 2012. Dijkstra's Shortest Path Algorithm. *Archive of Formal Proofs* (January 2012). https://isa-afp.org/entries/Dijkstra_Shortest_Path.html, Formal proof development.
[38] Marcus Vinícius Midena Ramos, José Carlos Bacelar Almeida, Nelma Moreira, and Ruy José Guerra Barretto de Queiroz. 2016. Formalization of the pumping lemma for context-free languages. *J. Formaliz. Reason.* 9, 2 (2016), 53–68. https://doi.org/10.6092/issn.1972-5787/5595
[39] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming* 58, 1-2 (2005), 206–263. https://doi.org/10.1016/j.scico.2005.02.009
[40] Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. 2009. Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. In *TPHOLs 2009 (LNCS, Vol. 5674)*. Springer, 424–439. https://doi.org/10.1007/978-3-642-03359-9_29
[41] Anders Schlichtkrull, Morten Konggaard Schou, Jiri Srba, and Dmitriy Traytel. 2022. Differential Testing of Pushdown Reachability with a Formally Verified Oracle. In *FMCAD 2022*. TU Wien Academic Press, 369–379. https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_44
[42] Anders Schlichtkrull, Morten Konggaard Schou, Jiří Srba, and Dmitriy Traytel. 2023. Labeled Transition Systems. *Archive of Formal Proofs* (October 2023). https://isa-afp.org/entries/Labeled_Transition_Systems.html, Formal proof development.
[43] Anders Schlichtkrull, Morten Konggaard Schou, Jiří Srba, and Dmitriy Traytel. 2023. Pushdown Systems. *Archive of Formal Proofs* (October 2023). https://isa-afp.org/entries/Pushdown_Systems.html, Formal proof development.
[44] Stefan Schwoon. 2002. *Model checking pushdown systems*. Ph. D. Dissertation. Technical University Munich, Germany. https://d-nb.info/96638976X/34
[45] Christoph Sprenger. 1998. A Verified Model Checker for the Modal μ-calculus in Coq. In *TACAS '98 (LNCS, Vol. 1384)*. Springer, 167–183. https://doi.org/10.1007/BFB0054171
[46] Simon Wimmer. 2016. Formalized Timed Automata. In *ITP 2016 (LNCS, Vol. 9807)*. Springer, 425–440. https://doi.org/10.1007/978-3-319-43144-4_26
[47] Simon Wimmer. 2016. Timed Automata. *Archive of Formal Proofs* (March 2016). https://isa-afp.org/entries/Timed_Automata.html, Formal proof development.
[48] Simon Wimmer. 2019. Munta: A Verified Model Checker for Timed Automata. In *FORMATS 2019 (LNCS, Vol. 11750)*. Springer, 236–243. https://doi.org/10.1007/978-3-030-29662-9_14
[49] Simon Wimmer and Peter Lammich. 2017. The Floyd-Warshall Algorithm for Shortest Paths. *Archive of Formal Proofs* (May 2017). https://isa-afp.org/entries/Floyd_Warshall.html, Formal proof development.
[50] Simon Wimmer and Peter Lammich. 2018. Verified Model Checking of Timed Automata. In *TACAS 2018 (LNCS, Vol. 10805)*. Springer, 61–78. https://doi.org/10.1007/978-3-319-89960-2_4
[51] Qingguo Xu and Huaikou Miao. 2006. Formal Verification Framework for Safety of Real-Time System based on Timed Automata Model in PVS. In *IASTED Conf. on Software Engineering*. IASTED/ACTA Press, 107–112.
[52] Qingguo Xu and Huaikou Miao. 2009. Manipulating Clocks in Timed Automata Using PVS. In *SNPD '09*. IEEE Computer Society, 555–560. https://doi.org/10.1109/SNPD.2009.69

A Reflection on Task-Oriented Programming

Mart Lubbers
Radboud University
Institute for Computing and Information Sciences
Nijmegen, The Netherlands
mart@cs.ru.nl

Tim Steenvoorden
Open Universiteit
Computer Science
Heerlen, The Netherlands
tim.steenvoorden@ou.nl

Abstract

Task-oriented programming (TOP) is a declarative programming paradigm where the main building blocks are tasks. Tasks represent work and have an observable task value. Tasks are combined to form compositions of tasks. From this specification of work, a ready-for-work application can be derived automatically.

There are several implementations of task-oriented programming, for example iTask, an industry-grade TOP system for distributed web applications; and TopHat, a fully formalised task-oriented language. iTask and TopHat differ a lot in philosophy. The iTask language only has three complex super combinators from which every other combinator is derived. This makes it difficult to provide a formal semantics for them. In TopHat more complex combinators are built from a rich set of simple building blocks, core combinators. Consequently, defining a formal semantics is easier.

By definition, the super combinators of iTask are more expressive than TopHat, as they allow the programmer to use the full host language Clean to define the behaviour. Whereas in TopHat, one has to create the behaviour by combining simple core combinators. The contribution of the paper is threefold, we perform a qualitative and quantitative analysis of task combinator usage. From that, we identify gaps between iTask and TopHat. Finally we introduce a new combinator, *reflect* (◎) to bridge a gap.

CCS Concepts

• **Software and its engineering → Language features**; **Very high level languages**.

ACM Reference Format:
Mart Lubbers and Tim Steenvoorden. 2025. A Reflection on Task-Oriented Programming. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Task-oriented programming (TOP) is a relatively new programming paradigm to model workflows [Plasmeijer et al. 2012]. It is a declarative programming paradigm where the basic building blocks are *tasks*. Tasks are an abstract representation of work and only describe *what* work needs to be done, the *how* is derived from this specification. Tasks have an *observable task value*. I.e. during the execution of a task, other tasks can observe the progress of the task and make decisions accordingly. Task values are observed by other tasks using *task combinators*. There is a rich set of task combinators that allow the composition of tasks. For example, tasks can be composed sequentially or parallel to form complex workflow systems. Besides exposing the progress of a task *following* control flow, tasks can share information *across* control flow using *Shared Data Sources* (SDSs).

There are several implementations of TOP, for example iTask, an industry-grade TOP system for distributed web applications; TOPPYT, a TOP implementation in Python [Lijnse 2022, 2024]; mTask, a TOP language for microprocessors that integrates with iTask [Koopman et al. 2018; Lubbers 2023]; and TopHat, a fully formalised TOP language. The first and the last are the focus of this paper.

The iTask system is a TOP implementation that generates an interactive multi-user distributed web server that facilitates users to perform the specified work [Plasmeijer et al. 2007]. It is implemented as an embedded domain-specific language (eDSL) in the purely functional host language Clean [Brus et al. 1987; Plasmeijer et al. 2021].[1] It has a long history and the set of task combinators changed continuously throughout the years [Lijnse 2013a]. The philosophy behind iTask is that with three super combinators, all other combinators are derived. This means that there is only one sequential super combinator (`step`), one parallel super combinator (`parallel`), and one transform combinator (`transformError`). As a consequence, deriving new combinators is relatively easy, but understanding or changing the exact semantics of the super combinators is difficult, as they define a mixture of complex behaviours. Attempts to define a semantics have been made but always only on a subset of iTask [Koopman et al. 2011; Plasmeijer et al. 2012]. There are many documented case studies in literature and it is used in industry, resulting in a relatively large codebase of real-world TOP applications.

TopHat is a TOP implementation that is fully mathematically formalised [Steenvoorden et al. 2019a]. This formalisation is used to drive a symbolic execution engine on tasks which supports next-step hints generation [Naus and Steenvoorden 2020; Naus et al. 2019]. It is also the basis for reasoning about equivalence of tasks [Klijnsma and Steenvoorden 2022]. The design of task combinators in TopHat is exactly opposite of iTask. Instead of deriving all combinators from three complex super combinators, there is a rich set of core combinators from which more complex ones can be derived. Over the years, the set of core combinators of TopHat has been extended to cover more and more real-world workflow

[1] A guide to Clean for Haskell programmers can be found in Lubbers and Achten [2024].

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

patterns. For example, allowing users to dynamically spawn more tasks [Steenvoorden and Naus 2024].

Motivation. Comparing the expressive power of iTask and TopHat is difficult due to their philosophical differences and their implementation details. Still, we would like to identify the feature gaps between these two top implementations. Many features in iTask grew over time and their theoretical usefulness is not directly apparent. Basing our knowledge on real-world usage is therefore paramount. Then, we can make a well founded decision on extending TopHat and providing formal semantics for missing features.

Contributions. The research contribution of this paper is threefold. First, we identify combinator usage of real-world top programs by analysing thirteen iTask's published case studies, two of iTask's internal workflow applications, and one real-world industrial application. Then, using this information, we establish the gaps between iTask and TopHat. For each of these gaps, we discuss the impact on top code and the necessity to formalise these features in TopHat. We conclude that task value reflection, detaching of tasks, exceptions, and advanced parallel combinator usage are features of iTask that are currently not supported by TopHat. Finally, we narrow the gap between iTask and TopHat in two ways. We identify a derived combinator *censor* ($\lfloor t \rfloor$) and we add task value reflection to TopHat by extending the language with a new *reflect* ($\circledcirc$) core combinator. We give its complete semantics and derive the reflect combinators seen in the real-world iTask examples from it.

Structure. The remaining of this paper is structured as follows. First, in Section 2 we introduce top and its concepts by means of TopHat. We discuss the language, its components and combinators, and its semantics. Next, in Section 3, we describe our method and results for analysing real-world top applications. For each category of combinators, we describe similarities and differences between TopHat and iTask. Then, in Section 4, we dive into the usage and semantics of reflection combinators and extend TopHat with a new combinator supporting these workflow patterns. Section 5 discusses related work and Section 6 concludes our paper.

2 Task-oriented programming in TopHat

In this section we discuss the semantics of task-oriented programs by means of TopHat [Steenvoorden et al. 2019b]. TopHat is a formal specification of task-oriented programming, with a verified implementation in Idris[2] and a practical one in Haskell.[3] It specifies the semantics of basic task-oriented operations. The framework has been extended for symbolic execution of tasks [Naus et al. 2019], and next-step hint generation [Naus and Steenvoorden 2020]. Also, it is the foundation of proving equivalence of task definitions [Klijnsma and Steenvoorden 2022].

2.1 Host and task languages

The TopHat language consists of two two parts: the *host language* and the *task language*. TopHat's host language is the simply typed λ-calculus with *basic types* β such as booleans, integers, and strings, and product and sum types thereof. It also contains *addresses* a, which are values on the host layer and can only be manipulated on the task layers. Most importantly, our host language has no operation for general recursion, and addresses are restricted to only contain basic types, that is, no functions nor other addresses. This means, evaluation of λ terms is pure and total.

[2]https://github.com/timjs/tophat-proofs
[3]https://github.com/timjs/tophat-haskell

Editors

d	$::=$	$\square^v \beta \mid \boxminus^v b \mid \boxdot^v b$	– unvalued, valued, read-only
	$\mid$	$\boxplus^v a \mid \circledast^v a$	– shared, read-only

Tasks

t	$::=$	$d \mid \blacksquare v \mid \lightning$	– editor, done, fail
	$\mid$	$v_1 \bullet t_2 \mid t_1 \blacktriangleright v_2$	– transform, step
	$\mid$	$t_1 \bowtie t_2 \mid t_1 \blacklozenge t_2$	– pair, choose
	$\mid$	$\bowtie^v_{t_0} [\overline{t}]$	– pool
	$\mid$	**share** $b \mid a_1 := b_2$	– share, assign

Figure 1: Grammar of TopHat's task language.

On top of the simply typed λ-calculus, TopHat builds a task language. Its grammar is given in Figure 1. Terms v are *values* and b *basic values* of the host language. In the following paragraphs, we discuss the operators in the task language. For more details about types and expressions in the host language, we refer to previous work [Steenvoorden 2022].

2.2 Editors

Editors are the endpoints of tasks, used to interact with end users. They are an abstraction over input fields or widgets, as seen in web frameworks and gui toolkits. Editors are typed, which means that in an Int editor, users can really only fill in integers.

Editors come in multiple flavours. *Unvalued editors* $\square\beta$ do not contain a value yet. They need to be filled with a value of the appropriate type β. *Valued editors* $\boxminus b$ do contain a value b, which can be modified by users. *Read-only editors* $\boxdot b$ also contain a value b, but can only be viewed by users and not modified. We discuss editors on shared data in Section 2.4.

To distinguish editors at runtime, they are *named* by unique names v. Running programs receive *inputs* which are basic values together with the unique name of the editor ment to receive the value. In examples, we omit names on editors if they are not needed in that context.

Example 2.1 (Editors). The simple program $t_0 = \square^{k_1}$ String is a task to fill in data of type String. Users see this as an empty input field on their screen. Typing in the the input "Hello reader" in this field, sends the input k_1!"Hello reader" to program t_0. TopHat's semantics will then rewrite program t_0 to $t_1 = \boxminus^{k_1}$"Hello reader". This process of entering string data can continue indefinitely.

2.3 Combinators

TopHat's combinators join smaller task into bigger ones. Combinators come in two main forms: sequential and parallel.

The main sequential operator is a *step* from task t_1 to its continuation v_2, denoted by $t_1 \blacktriangleright v_2$. Here, v_2 is a pure function which calculates the next task to be performed based on its arguments. When this calculated task happens to be the *fail* task ($\lightning$), the step is not made and we stay working on t_1.

A task to check age restrictions, could be programmed as follows.

$$\square\textsc{Int} \blacktriangleright \lambda n.\ \textbf{if}\ n < 18\ \textbf{then}\ \lightning\ \textbf{else}\ \square\textsc{FormData}$$

This task indefinitely asks to enter an age, and only continues to the next task of filling out the form data if the entered number is higher than 18.

The transform combinator $v_1 \bullet t_2$ *transforms* the observable value of a task t_2 into something new by mapping the pure function v_1 over the value of t_2. As an example, take the task trafficLight below.

$$\textbf{type}\ \textsc{Light} = [\mathsf{Green}, \mathsf{Red}]$$
$$\textbf{let}\ \mathsf{trafficLight} = \lambda x.(\textbf{if}\ x\ \textbf{then}\ \mathsf{Green}\ \textbf{else}\ \mathsf{Red}) \bullet \square\textsc{Bool}$$

This initially does not have an observable value. It asks users to enter a boolean True or False, but returns one of the colours Green or Red by mapping a lambda over the task □Bool.

The parallel combinators also come in three forms: pair, choose, and pool. *Pairing* two task $t_1 \bowtie t_2$ let us work on both t_1 and t_2 interleaved. The observed value of both tasks is combined in a tuple if both are available, otherwise, it does not have a value. So, ⧈"Cat" ⋈ ⊟True has the observable value {"Cat", True}, but ⧈"Cat" ⋈ □Bool has no initial observable value.

Choosing between two tasks, $t_1 \blacklozenge t_2$, also means we can work on both tasks interleaved. However, the observed value is the value of t_1 if it is available, otherwise, it chooses the value of t_2. For example, □ Int ◆ ⊟ 42 immediately normalises to ⊟ 42, because □ Int does not have an observable value. If both values are unavailable, this also does not have a value.

Both pairing and choosing are *static* combinators, i.e. the number of tasks to work on is specified at development time and tasks cannot be added or deleted at runtime. This can be achieved by using *task pools*, denoted $\bowtie\!\triangleleft^{v}_{t_0}[\bar{t}]$. Task pools can receive inputs from users, therefore they need to be identifiable by a name v. They are parametrised by a *template task* t_0. Upon receiving an add-input, this template task is added to the task list $[\bar{t}]$. Task in this list can also be dynamically removed by sending a remove-input.

2.4 Sharing data

Note that, till now, data could only be passed from task to task sequentially: when groups of tasks finish, resulting values can be used to calculate continuations. This is too restrictive to describe general workflow systems, where parallel workflows need to react on data from each other. Therefore, data in TOP specifications can be shared.

Shared data is introduced by **share** b, which allocates the basic value b in memory and returns its address a. Using this a, multiple tasks can watch the same data. For example, *shared editors* ⊞a watch address a, show the data at a to end users, and allow them to change it. Similarly, *read-only shared editors* ◙a also watch an address a, but end users cannot change its data. The application itself can set addresses to any basic value using $a_1 := b_2$.

2.5 Observations

Tasks form syntax trees which can be *observed*. The most important observation on tasks is their current *value*. This is a partial function $\mathcal{V}$ from task trees to values. For example, unvalued editors like □Bool do not have a value, while valued editors like ⊟True have a value. Value observations are defined recursively on task trees. Notably steps never have a value, as we cannot tell what continuation it will evaluate to.

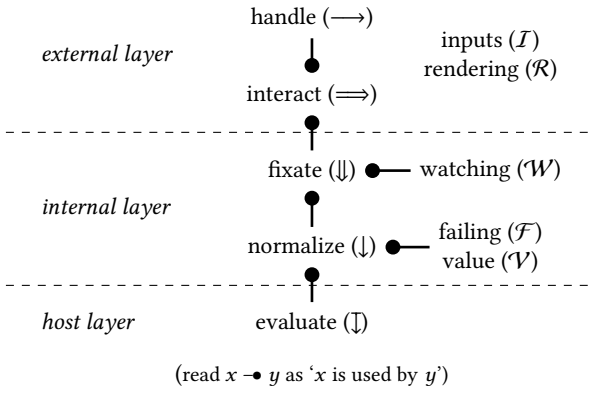


Figure 2: Overview of semantic layers, relations and functions in TopHat.

Tasks can be observed to be *failing* ($\mathcal{F}$) or not. The fail task ↯ is failing, as is the paring of fail tasks ↯ ⋈ ↯, these tasks cannot be worked on. However, the task ↯ ⋈ □Int is not failing, as the second element is a task that can be worked on. TopHat's semantics ensure steps are never made to failing tasks.

Other observations on tasks are the set of *watched addresses* ($\mathcal{W}$), and the possible *inputs* that can be send to a task ($\mathcal{I}$). Also, the *rendered* (graphical) user interface can be modelled as an observation ($\mathcal{R}$).

2.6 Semantic layers

TopHat semantics is defined in three layers: the host layer, the internal layer, and the external layer. These are depicted in Figure 2. At the bottom, there is the *host layer*, which evaluates pure lambda terms. On top of that, there are two *task layers* reasoning about the task language. The semantic arrows in the *internal layer* prepare a task for user interaction. The semantic arrows in the *external layer* do the actual handling of user inputs. At the right side of Figure 2, we can see the observations that play a role on that layer. Adding features to the task layer, should not alter the semantics of the host language and vice versa.

3 Analysing real-world TOP applications

TOP and iTask in particular have a very long history with many published case studies, examples, support applications and even some commercial use. In order to get some insight in real-world TOP combinator usage, we did a quantitative analysis of all real-world TOP applications or fragments. In this section we explain the method used to do this analysis, present our results, and for each category of combinators we describe similarities and differences between TopHat and iTask. We finalize the section with a short discussion of our findings.

Table 1: List of analysed real-world applications.

name	description
conf2009	a conference management system designed to teach students iTask at the AFP 2008 school [Plasmeijer et al. 2009].
itasks22009	a set of example programs bundled with the iTask 2 system [Lijnse and Plasmeijer 2009].
esmviz2011	a visualisation tool for testing a simplified version of iTask,written in iTask [Koopman et al. 2011].
gin2012	the frontend for GiN, a graphical interactive applications to design tasks [Henrix et al. 2012].
incidone2012	an incident report application for coordinating rescue situations [Lijnse et al. 2012].
trax2013	a single-player puzzle game designed to demonstrate the elegance en ergonomics of top [Achten 2013].
tonic2014	the fronted for Tonic, a static and dynamic visualisation tool for iTask tasks [Stutterheim et al. 2014].
ligretto2014	a multi-user card game designed to demonstrate the scalable vector graphics integration in iTask [Achten et al. 2014].
tasklets2015	a set of bigger examples for executing small tasks in the browser using *TaskLets* and *EditLets* [Domoszlai and Kozsik 2013; Domoszlai et al. 2014a; Domoszlai and Plasmeijer 2015].
shipadventure2017	an interactive fire-extinguishing game situated on a naval ship created to demonstrate task-oriented software development with a non-trivial case study [Stutterheim 2017].
serviceengineer2017	a case study to showcase the distributed extension of iTask. It is an application to manage and perform job allocation for service engineers [Oortgiese et al. 2017].
taxman2018	a workflow system for entering solar panel reimbursements, designed after specifications provided by the Dutch tax office [Stutterheim et al. 2018].
cws2023	a smart campus monitoring iot system prototype designed after specifications given by the University of Glasgow. It is written in both iTask and mTask, the latter runs on microcontrollers [Lubbers et al. 2023].
admin2024	a collection of task workflows for the iTask system to administrate the server itself [Plasmeijer et al. 2012].
examples2024	a snapshot of the set of example programs bundled with the iTask system [Plasmeijer et al. 2012].
top2024	an industrial deployed GIS application developed by TOP Software B.V. of around 110 kLOC.

3.1 Method

To gather top applications, we analysed little over 50 published papers on top, mainly about iTask but also a handful on mTask. Of those papers, we where able to recover 13 applications or example sets either from the provided artifact, by digging into the version control history of the iTask framework, or by asking the authors for a copy. Additionally, we extracted one application and an example set from the current version of the iTask framework.[4] Finally, we included the analysis of a commercial geographic information system (gis) application. This results in a total of 16 annalysed applications, which are all listed in Table 1.

The 15 published or publicly available real-world top applications have a total of 30 kLOC (source lines of code). The commercial application consists of around ±110 kLOC. This demonstrates that the programs at least have quite some developer effort that was put into them [Rosenberg 1997]. We do not draw conclusions on how realistic every application is based on the number of LOC, as that metric is not necessarily a realistic measure for that purpose [Alpernas et al. 2020]. The commercial application and the support applications of iTask on the other hand are realistic applications by definition because they are used by many users and are constantly adapting to current needs.

In order to analyse the combinator usage, we created a program that recursively scans all modules and keeps track of the combinator usage. It does so by parsing the module with the use of the Clean compiler and looks up every identifier to see whether it is a known combinator. The list of known combinators is compiled by hand and enumerates all known task combinators and their category.

[4]The iTask framework is found here: https://gitlab.com/clean-and-itasks/itasks-sdk.

Table 2: Distribution of combinator categories.

category	subcategory	num	% of parallel	% of total
sequential	—	2270	—	58.99
transform	—	825	—	21.44
parallel	or	282	45.41	7.55
	and	187	30.11	4.86
	detach	67	10.79	1.74
	reflect	39	6.28	1.01
	pool	38	6.12	0.99
	rest	8	1.29	0.21
	total	621	100.00	16.14
exception	—	20	—	0.54
total	—	3848	—	100.00

The direct occurrences of the iTask super combinator `parallel` (1 in the commercial and 19 in the rest) where hand checked on a case-by-case basis to determine if the use really required to use this super combinator or could be rewritten to use simpler derived combinators. Sometimes the super combinator `parallel` is used for ergonomics and the behaviour could be expressed otherwise. The analysis was ran locally on the applications where the source code was available to us. For the commercial application we asked the company to run the tool on their source code since it is proprietary.

3.2 Results

In total, 140 kLOC from of 395 modules was automatically analysed resulting in a total of 3848 uses of task combinators. These task

combinators are divided into four categories: sequential combination (Section 3.4), transformation of tasks (Section 3.5), parallel combination (Section 3.6), and exception handling (Section 3.7). The parallel combinator is furthermore split up into six categories: disjunctive (or) and conjunctive combination (and) (Section 3.6.1), detach behaviour (Section 3.6.2), reflect behaviour (Section 3.6.3), pool behaviour (Section 3.6.4), and the rest (Section 3.6.5). Table 2 shows the results of this analysis. The following sections details on the different aspects and combinators and how they relate to TopHat's.

3.3 Representation of task values and sdss

Before diving into the combinators, we first have to discuss the differences between task value representation in iTask and TopHat. In both systems, tasks have observable task values. However, in TopHat a task either has a value or not, in iTask a task value also has a stability. The definition for task values as used in iTask is presented in Listing 1. An unstable task value signifies that it can change in the future, a stable task value cannot but this is in no way guaranteed by the iTask system.

```
:: TaskValue a = NoValue | Value a Stability
:: Stability :== Bool
```

Listing 1: Task value data type in iTask.

Stability of tasks as used in iTask can in TopHat be mapped on the task conditions of Klijnsma and Steenvoorden [2022, §5] where stable is *finished steady*, unstable is *finished unsteady* and no value is *running*. In TopHat, stability can be simulated by yielding a tuple of the real task value and a Bool representing the stability.

Furthermore, the data type for sdss in iTask (simplified to `Shared a`) separates a read type, a write type and a parameter type [Domoszlai et al. 2014b]. Lijnse and Plasmeijer [2021] introduced editors with a separate read and write type as well but comparing this to TopHat is beyond the scope of this paper. In our examples, we assume that the read type and the write type are the same parameter.

3.4 Sequential combinators

Of all combinators, the vast majority (58.99%) are sequential combinators. All sequential combinators in iTask are derived from one super combinator, `step`, which signature is shown in Listing 2.[5] This combinator has two arguments: a left-hand side, the task to execute first; and a right-hand side, a task continuation. This construction allows the right hand side to *observe* the task value of the left-hand side and act upon it. There are four different types of continuations (also listed in Listing 2):

(1) `OnValue` is used to perform the step without user intervention, just by a predicate on the task value, we call this an *internal step*;
(2) `OnAction` is only triggered when the user performs an action, e.g. click on a button, still a predicate is used to determine whether the button is enabled or not, we call this an *external step*;
(3) `OnException`, and (4) `OnAllExceptions` are used to step when a certain exception occurs (see Section 3.7).

```
(>>*) :: (Task a)                // Current task
         [TaskCont a (Task b)]   // Continuation list
         → Task b

:: TaskCont a b
   = OnValue ((TaskValue a) → ?b)
   | OnAction Action ((TaskValue a) → ?b)
   | ∃.e: OnException (e → b) & iTask e
   | OnAllExceptions (String → b)
```

Listing 2: Super sequential combinator in iTask.

Besides the ability to catch exceptions, there seemingly is a difference between the iTask sequential combinator and TopHat's (►). Firstly, TopHat does not explicitly support external steps, only internal ones. External steps are simulated with normal editors. Pushing a button, for example, is simulated with:

$$\square\textsc{Unit} \blacktriangleright \lambda\{\}.\ t_{\text{cont}}$$

Secondly, in iTask one can create task continuations that continue even if the left-hand side task has no value. While this is quite a pathological case because the left-hand side task only gets one normalisation step, it is possible to implement this in TopHat as well, using the fact that ■Unit always has a value:

$$(t \blacklozenge \blacksquare\textsc{Unit}) \blacktriangleright \lambda\_.\ t_{\text{cont}}$$

In general, it is possible to expose the no value state of tasks in TopHat, by hoisting or exposing the task value as a sum type together using the fact that ◆ is left-biased:

$$\text{expose}\ t = (\text{Value} \bullet t) \blacklozenge \blacksquare\text{NoValue}$$

Furthermore, the way iTask prevents or guards a step is also a bit different but related. In iTask, the *Maybe*[6] type is used whereas in TopHat, failure is an observation of the resulting task using $\mathcal{F}$, i.e. $\lightning$.

3.5 Transform combinators

Transform combinators comprise around 21.44% of the entire combinator base. These combinators allow the programmer to apply a pure function to the task value of the task. In iTask, transform combinators are all derived from the super combinator `transformError` seen in Listing 3.[7] Not only does this function allow to transform the task value, it also allows you to inject exceptions (see Section 3.7).

```
transformError ::  // | Transformation function
   ((TaskValue a) → MaybeError TaskException (TaskValue b))
   (Task a)       // Current task
   → Task b
```

Listing 3: Super transform combinator in iTask.

[5] All types in iTask have a class collection `iTask` that contains derivable type classes for printing, storage, editors and equality. From now on we will omit these class constraints for readability of the type signatures.

[6] In Clean ? is the builtin strict *Maybe* type: `:: ? a = ?None | ?Just !a`

[7] The error type in Clean is defined as `:: MaybeError e a = Error e | Ok a`.

Other than the option to inject exceptions, the seeming difference between this super combinator and TopHat's (•) is the ability to transform no value into a value and the other way around. However, both can be expressed with TopHat as well. Using expose shown in Section 3.4, the existence of a task value is exposed. Then a transform function using the exposed value can be applied. The question is then how to lower the hoisted task value back into the task. Converting a NoValue to a task without a value can be done using a step. For convenience, as we will need it later, we define a new derived combinator for censor ($\lfloor t \rfloor$). Censoring a task t, means we *hide* its value for the outside world. In TopHat, we accomplish this by attempting a step to the fail task. This step is never made, because TopHat's semantics use $\lightning$ to signal an impossible step. Also, steps do not have a value, therefore censoring the value of t.

$$\lfloor t \rfloor = t \blacktriangleright \lambda\_.\ \lightning$$

Converting a hoisted task value back into a task value in general is not possible. However, it can be done in specific cases using a step (▶) in TopHat, but that freezes the value:

$$\text{freeze}\ t = t \blacktriangleright \lambda x.\ \textbf{case}\ x\ \textbf{of}\quad \begin{array}{ll} \text{Value}\ x & \mapsto x \\ \text{NoValue} & \mapsto \lightning \end{array}$$

3.6 Parallel combinators

16.14% of the combinators are parallel combinators and they typically appear high up in task trees. Parallel combinators perform the child tasks *at the same time*, but not necessarily truly in parallel. For example, in iTask, but also in TopHat, the user interfaces of the children are combined and presented to the user. Event handling is strictly sequential though and left biased.

All parallel combinators in iTask are implemented using the super combinator `parallel` (see Listing 4). This combinator is implemented as a function with two arguments. The first argument is an initial list of not just ordinary tasks but `ParallelTasks`. These are functions that, when given a `SharedTaskList`, produce ether a regular task (`Embedded`), or a detached task (`Detached`, see Section 3.6.2). This `SharedTaskList` can be used to observe the task values of all siblings and add, remove, or even replace tasks at will. The second argument is a list of task continuations that operate on task values of every task in the list and produce parallel tasks as continuations that are added to the task list as a new child.

Parallel tasks can use the `SharedTaskList` to manage the parallel combinator, e.g. add, remove or replace tasks. Moreover, the `SharedTaskList` is used to observe tasks. This allows for very dynamic behaviour that is similar to full control over a process table in an operating system [Plasmeijer et al. 2011].

The parallel super combinator is notoriously difficult to implement and reason about [Lijnse 2013a]. Furthermore, it is quite difficult to use (see for example the implementation of `-&&-` in Listing 6) so real-world TOP applications almost exclusively use derived combinators that we divide into six subcategories. Most of the derived combinators are expressible in TopHat as well as we will see in the following sections.

3.6.1 and *and* or *combinators.* The simpler derived parallel combinators, disjunction (`-||-`, `-||`, and `||-`) and conjunction (`-&&-`) are used in 45.41% and 30.11% of all parallel combinators. Conjunction and disjunction are both binary combinators that run the tasks at the same time in a `parallel` and combine the task values accordingly. There are also list variants of this combinator available called `allTasks` and `anyTask`. All signatures are shown in Listing 5.

```
:: ParallelTaskType
   = Embedded
   | Detached Bool TaskAttributes

:: ParallelTask a :== (SharedTaskList a) → Task a

parallel :: [(ParallelTaskType, ParallelTask a)]
            [TaskCont [(Int, TaskValue a)]
                      (ParallelTaskType, ParallelTask a)]
            → Task [(Int, TaskValue a)]
```

Listing 4: Super parallel combinator in iTask.

```
(-&&-)   :: (Task a) (Task b) → Task (a, b)
(-||-)   :: (Task a) (Task a) → Task a
(-|| )   :: (Task a) (Task b) → Task a
( ||-)   :: (Task a) (Task b) → Task b
allTasks :: [Task a] → Task [a]
anyTask  :: [Task a] → Task a
```

Listing 5: And and or derived combinators in iTask.

The simple *and*- and *or*-combinators have the same semantics as TopHats pair (⋈) and choose (◀▶) combinators. Due to the complexity of the super combinator, deriving them is not trivial (see Listing 6). Derived combinators `allTasks` and `anyTask` can be expressed as folds over using ⋈ and ◀▶ [Steenvoorden 2022]. Finally, the left and right censoring variants (`-||` and `||-`) are derived using the censor combinator:

$$\begin{aligned} t_1\ \texttt{-||}\ t_2 &= t_1 \blacktriangleleft\blacktriangleright \lfloor t_2 \rfloor \\ t_1\ \texttt{||-}\ t_2 &= \lfloor t_1 \rfloor \blacktriangleleft\blacktriangleright t_2 \end{aligned}$$

```
(-&&-)   :: (Task a) (Task b) → Task (a, b)
(-&&-) taska taskb = parallel
    [ (Embedded, \_ → taska @ Left)
    , (Embedded, \_ → taskb @ Right)] [] @? res
where
    res (Value [(_,Value (Left a) sa)
               ,(_,Value (Right b) sb)] _)
        = Value (a,b) (sa && sb)
    res _ = NoValue
```

Listing 6: Implementation of the and derived combinator in iTask.

3.6.2 detach *combinators.* The third biggest category of combinators is the *detach* category, spanning 10.79% of the parallel combinator category. These combinators are implemented using the `Detached` type of parallel task to graft a task tree onto another task tree. The most popular combinator is the assign combinator (`@:`,

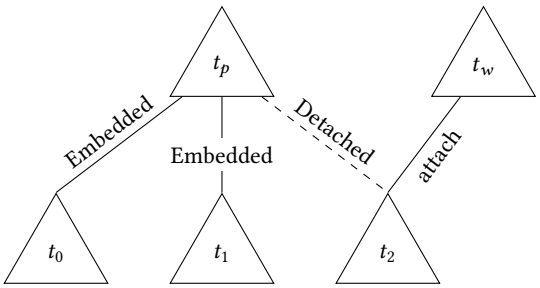


Figure 3: Visualisation of detach behaviour.

Listing 7). This *assign*s a task to a specific user but allows the original task to observe the progress: a loose coupling. So the task is *detached* and the user can see this incoming task and choose to *attach* to the task to work on it.

```
(@:) :: User (Task a) → Task a
attach :: InstanceNo Bool → Task AttachmentStatus
```

Listing 7: Assign core combinator in iTask.

Figure 3 illustrates a parallel with a detach task. Task t_p is a parallel task, arbitrarily deep in a task tree with several embedded tasks and one detached task. Detaching the task means separating it from the task tree but it maintains a loose link that allows progress to be observed. The task is part of t_p, but its user interface is not shown and it cannot be worked on by t_p. In the figure, t_w attaches to the task to work on it.

In the iTask implementation, the parallel combinator uses an SDS to store the list of tasks. SDS's are persistent over execution and data is serialised using the *GraphCopy* library for serialising execution graphs, i.e. Graph Packing [Berthold 2011; Oortgiese et al. 2017]. Serialising tasks, i.e. execution graphs, works well but is not stable across different versions of the code or even compilations. The commercial application (**top2024**) therefore never uses detach behaviour directly but always simulates it using data. Instead of detaching a task for someone to attach to, a label is written into an SDS. The other user then knows which task to start from the label and explicitly communicates its value via a dedicated SDS.

Detaching parts of task trees and grafting it to other task trees is currently not implementable in TopHat. Finding new core combinators to implement this remains future work. Its limited form, as used by the commercial application, can be simulated similarly using labels.

3.6.3 reflect *combinators.* Then, there is a category of parallel combinators called the *reflect* combinators, with an amount of 6.28% of the total parallel combinators. These combinators utilise the `parallel` super combinator to monitor other task's progress *across* control flow. There are three reflection combinators that are often used shown in Listing 8. All expose the task value in SDSs to provide to a task function.

Feed-forward (`>&>`) exposes the left-hand side task's value in an SDS that is passed to the right-hand side task function. The task value of the right-hand side is considered the task value of the compound task. Feed-sideways does the same but considers the task value of the left-hand side as the task value of the compound task. Finally there is the feed-bidirectional combinator. This ties a knot between two tasks by exposing the one observable task value to the other and vice versa. The task value of the combined task is the pair of the individual tasks, similar to the conjunctive combinator (`-&&-` in iTask, ⋈ in TopHat). In Section 4 we provide a new TopHat core combinator to implement these combinators.

```
(>&>) :: (Task a) (Shared (TaskValue a)) → Task b) → Task b
(>&^) :: (Task a) (Shared (TaskValue a)) → Task b) → Task a
(<&>) :: ((Shared (TaskValue b)) → Task a)
         ((Shared (TaskValue a)) → Task b)
             → Task (a, b)
```

Listing 8: Derived reflection combinators in iTask.

3.6.4 pool *combinators.* Pool combinators are combinators that manage a list of tasks, i.e. shrinking and growing the list by actions of the user. They comprise around 6.12% of all parallel combinators. The most used combinator in the pool category is the sideStep (`>^*`) combinator (Listing 9). This combinator has a similar type signature to the regular step (Listing 2) only with a different return type: `Task a` instead of `Task b`. Instead of continuing with one of the continuations, the picked continuation task is added as a child to the parallel.

```
(>^*) :: (Task a) [TaskCont a (Task b)] → Task a
```

Listing 9: Sidestep derived combinator in iTask.

When the `TaskConts` are just actions, i.e. buttons, this is implementable using the pool combinator from TopHat (⋈). Otherwise it is considered to be *rest* behaviour.

3.6.5 rest *combinators.* Then there is the *rest* category, only spanning 1.29% of parallel combinator use. This type of behaviour uses the `parallel` combinator to dynamically shrink and grow the task list without user intervention. Implementing this in TopHat is possible but many nice properties such as symbolic execution [Steenvoorden and Naus 2024] will be lost.

3.7 Exception combinators

Finally, the last 0.54% of the main categories of combinators concerns exception handling. In iTask, exceptions are caught using either sequential or parallel combinators, but it is also possible to catch them using the derived combinators `try` and `catchAll`.

Exceptions are also (ab)used as a way of short circuiting execution. For example, when the left-hand side of a parallel *or* combinator throws an exception, the right-hand side of the combinator is not executed any more, the exception prevented that. This is never used directly and only found in one derived combinator, `whileUnchanged` (Listing 10), so it can be considered an implementation detail. This function has two arguments, the first argument is an SDS, the second a task function that is restarted each time the SDS changes. Internally it is implemented with an *or* combinator that is short circuited.

```
whileUnchanged :: (Shared a)    // SDS to observe
                  (a → Task b) // Task function
                  → Task b
```

Listing 10: The type of iTask's `whileUnchanged` function.

Typical use cases of exceptions are easily implementable in TopHat by encoding them as a sum type such as `MaybeError`. It remains future work to see how the short circuiting behaviour can also be implemented.

3.8 Discussion

The analysis of the large code base showed that TopHat covers the majority of real-world workflows. All real-world sequential and transform combinators can be expressed in equivalent TopHat. This already comprises 80.43% of all combinators. Of the remaining 19.57%, 3.50% is not yet expressible, containing some variants of the parallel combinator and exceptions. For the *reflect* behaviour, we have identified a core combinator that allows us to express this. The implementation and semantics are shown in the next section. We leave the *detach* and *exception* behaviour for future work.

Further work could be to split out the combinators even more. For example analysing task continuation usage as well. Alternatively, combinator distribution can be propagated, there are many derived combinators that use multiple core combinators, this could be propagated to get a more detailed view of core combinator statistics. Finally, the SDSs in iTask are richer and support combinators as well, it would be interesting to see how they compare to TopHat.

4 Reflection on task values

During the quantitative analysis of the combinator usage in real-world TOP applications, we identified a gap. All different *reflect* behaviours, comprising 6.28% of the parallel combinator usage, and possibly some of the *rest* behaviour, can be expressed with a single new core combinator: *reflect*. In this section, we describe the observed *reflect* combinators with examples from the real-world applications, introduce a reflect core combinator (◎) with its semantics in TopHat, and finally show how to derive the real-world combinators from it.

4.1 Reflect derived combinators

There are two combinators that comprise almost all of the reflection behaviour seen in real-world applications. Those are feed-sideways (`>&^`) and feed-forward (`>&>`), which types we have seen in Listing 8. They expose task values into an SDS so that other tasks can observe the progress *across* control flow.

In the analysed examples we see higher-level workflows that depend on this behaviour, for example in the helper function `crud` used in **taxman2018** and **shipadventure2017**. Listing 11 shows this function. It creates a task with which one can execute create/read/update/delete (CRUD) operations on an SDS. The left-hand side of the combinator is a task for selecting the key of the entry you want to edit. The right-hand side of the combinator is an expression that creates a task from the currently selected key. All operation on the map are guarded except for *Create* for it needs no key. When the result of the database operations are not needed for the compound task, `feedSideways` is used. Changing it to `feedForward` changes the task value to the result of the operation once it is finished. The generated resulting application is shown in Figure 4.

```
crud :: (Shared (Map k v)) → Task k
crud sds = (enterChoiceWithShared [ChooseFromList id]
        (mapRead keys sds)
        <<@ Title "Entries in database"
    >&^ \mk→forever $ viewSharedInformation [] mk
        <<@ Title "Current selection"
    >>* [ OnAction (Action "Create") (always  createTask)
        , OnAction (Action "Read")   (hasJust readTask)
        , OnAction (Action "Update") (hasJust updateTask)
        , OnAction (Action "Delete") (hasJust deleteTask)
        ])
where
    hasJust f = ifValue isJust (f o fromJust)
    createTask = enterInformation []
        <<@ Label "Enter the key and value"
        >>? \(k, v)→upd (put k v) sds
    readTask k = ···
    ···
```

Listing 11: A simple CRUD interface to an SDS

Furthermore, `feedForward` is useful as a debugging tool when doing rapid prototyping, an often described use case of iTask [Lijnse 2013b]. Such usage has for example been seen in **tasklets2015** codebase. In this example, `feedForward` is combined with the `withSelection` function. Listing 12 shows the type of this function and an example of its usage. It results in the user interface shown in Figure 5.

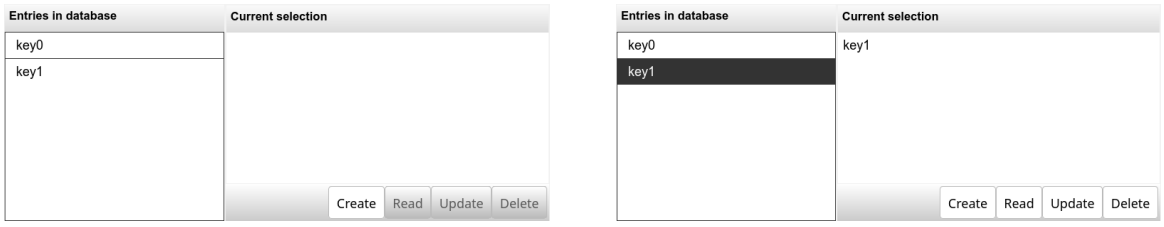
```
withSelection :: (Task c)  // When there is no value yet
    (a → Task b)           // When there is a value
    (Shared (TaskValue a)) // The SDS reflecting the value
    → Task b

withSel :: Task Int
withSel = enterInformation [] <<@ Label "Enter a number"
    >&> withSelection (viewInformation [] "Nothing entered")
        (\v→viewInformation [] v <<@ Label "You entered")
```

Listing 12: The type of `withSelection` and an example of debugging a task value.

Recently, a new derived reflection combinator, feed-bidirectionally (`<&>`), has been added to the iTask system. This novel combinator ties a knot between two tasks, letting them observe each other's result. This is typically used high up the tree and is due to its novelty only observed in **top2024**. The task value in that case often represents some kind of progress or some kind of status.

4.2 Reflection in TopHat

We observed that these three combinators, feed-forward, feed-sideways, and feed-bidirectionally, in iTask all *reflect* the value of a task into an SDS. That is, other tasks can see and react upon



(a) Without a selection, only the Create button is enabled.

(b) When selecting a key, the Read, Write and Update buttons are enabled as well.

Figure 4: The interface for the CRUD task.

(a) When there is no task value yet, it shows only the label *Nothing entered*.

(b) When there is a task value, it shows the label *You entered* and the current task value.

Figure 5: The user interface of Listing 12.

the value of the reflected task by watching this SDS or address. To accommodate this, we extend TopHat with a new core combinator, reflect:

$$t ::= \ldots \mid \circledcirc_a \ t$$

where the value of task t is reflected at address a.

The typing rule for the reflect combinator is as follows:

$$\text{T-Reflect}\quad \frac{\Gamma, \Sigma \vdash a : \textsc{Ref}\ (\textsc{Maybe}\ \tau) \qquad \Gamma, \Sigma \vdash t : \textsc{Task}\ \tau}{\Gamma, \Sigma \vdash \circledcirc_a \ t : \textsc{Task}\ \tau}$$

Here, Ref τ is the type of references to τ, and Task τ is a task containing a value of type τ. As tasks can or cannot have a value, we need to take this into account and use the standard Maybe type to wrap the reflected value in the reference.

Next, we discuss the implementation of reflection in TopHat's semantics. Of the semantic arrows shown in Section 2, only the normalisation ($\downarrow$) and handling ($\longrightarrow$) semantics need to be extended. However, all observations need to be extended to support reflection on tasks.

4.2.1 Normalisation and handling. As stated in Section 2.6, normalisation is the process of preparing tasks for user input. They form a subset of tasks t. Normalising $\circledcirc_a \ t$ in state σ comprises:

(1) normalising t in σ to normalised task n', delivering a new state σ' and some dirty addresses δ';
(2) adding a to the dirty addresses, as we modified it;
(3) setting address a to the value of n', which is precisely the reflection behaviour we like to model;
(4) returning a new reflect combinator on address a, but now on the normalised task n'.

The final derivation rule for normalisation reads:

$$\text{N-Reflect}\quad \frac{t, \sigma \downarrow n', \sigma', \delta'}{\circledcirc_a t, \sigma \downarrow \circledcirc_a \ n', [a \mapsto v]\sigma', \delta' \cup \{a\}}\ v = \begin{cases} \text{Just}\ \mathcal{V}(n', \sigma') & \text{if defined} \\ \text{Nothing} & \text{otherwise} \end{cases}$$

Here we use constructors Just and Nothing to take into account that normalised task n' can or cannot have a value.

To handle inputs ι by reflections we simply pass the input on to the inner, now normalised, task n. The result is now a not-normalised task t'.

$$\text{H-Reflect}\quad \frac{n, \sigma \xrightarrow{\iota} t', \sigma', \delta'}{\circledcirc_a \ n, \sigma \xrightarrow{\iota} \circledcirc_a \ t', \sigma', \delta'}$$

4.2.2 Observations. Observations on reflections are mostly delegating to the same observation on the inner task. Only $\mathcal{W}$ also adds address a to the watch set. Note that all observations, except $\mathcal{F}$, actually operate on a normalised task n.

$$\begin{aligned} \mathcal{F}(\circledcirc_a \ t) &= \mathcal{F}(t) \\ \mathcal{V}(\circledcirc_a \ n, \sigma) &= \mathcal{V}(n, \sigma) \\ \mathcal{I}(\circledcirc_a \ n) &= \mathcal{I}(n) \\ \mathcal{W}(\circledcirc_a \ n) &= \mathcal{W}(n) \cup \{a\} \\ \mathcal{R}(\circledcirc_a \ n, \sigma) &= \mathcal{R}(n, \sigma) \end{aligned}$$

As the value of n is reflected at address a, one could think that the value at address a is *always* the same as the value of the reflected, normalised task n:

$$\mathcal{V}(\circledcirc_a \ n, \sigma) = \mathcal{V}(n, \sigma) = \sigma(a) \quad \text{Not true!}$$

Here $\sigma(a)$ looks up the value stored at address a in state σ. Sadly, this is not true. Take for example the following task:

$$\textbf{share}\ \mathsf{Nothing} \blacktriangleright \lambda a.\ (\circledcirc_a\ \blacksquare 37) \bowtie (a := 42)$$

This should reflect the value 37 in a but *also* sets a to 42. Because the semantics of the pairing combinator ($\bowtie$) is defined to be *left-to-right*, normalising $a := 42$ is done *after* normalising $\circledcirc_a\ \blacksquare 37$. Therefore, the value stored in a is 42 and not 37 and the value of the left task is not reflected in the address. This could be solved by introducing read-only memory locations in the host language.

4.3 Deriving the feed combinators

With the semantics of reflection in place, we can now define the feed-forward, -sideways, and -bidirectionally combinators. For this, we need one helper on tasks, *censor* ($\lfloor t \rfloor$), which we already discussed in Section 3.5 and repeat its definition here for convenience:

$$\lfloor t \rfloor = t \blacktriangleright \lambda\_.\ \lightning$$

Now we can define feed-forward ($\circledcirc$), -sideways ($\circledcirc$), and -bidirectionally ($\circledcirc$) using censor as follows:

$$\begin{aligned} t \circledcirc e &= \textbf{share}\ \mathsf{Nothing} \blacktriangleright \lambda a.\ \lfloor \circledcirc_h t \rfloor \blacklozenge e\ a \\ t \circledcirc e &= \textbf{share}\ \mathsf{Nothing} \blacktriangleright \lambda a.\ \circledcirc_h t \blacklozenge \lfloor e\ a \rfloor \\ e_1 \circledcirc e_2 &= (\textbf{share}\ \mathsf{Nothing} \bowtie \textbf{share}\ \mathsf{Nothing}) \\ &\quad \blacktriangleright \lambda\{a_1, a_2\}.\ \circledcirc_{a_1}(e_1\ a_2) \bowtie \circledcirc_{a_2}(e_2\ a_1) \end{aligned}$$

To feed-forward $t \circledcirc e$ or feed-sideways $t \circledcirc e$, we start by sharing the value Nothing at address a, this address is then used by the reflect combinator to reflect the value of task t. However, in case of feed-forward, we are not interested in the result of this task, but in the result of the task computed by $e\ a$. Therefore, we censor the left-hand side of the choice operator. This results in the right-hand side at the moment it produces a value. In case of feed-sideways, this is exactly the other way around, we censor the right-hand side to result only the left-hand side reflected task.

In case of feed-bidirectionally $e_1 \circledcirc e_2$, we start sharing at two addresses, a_1 and a_2. We reflect the value of e_1 in a_1, and of e_2 in a_2. Note we pass the addresses reflecting *the other* task to each expressions.

5 Related work

In the quantitative and qualitative analysis of this paper, around 50 TOP related papers and case studies where extracted. We are quite confident that, barring at most one or two papers, this is the entire body of research on TOP. However, there is some other related work.

Functional reactive programming (FRP) is a programming style that at first glance looks similar to TOP. The term was first coined by Elliott and Hudak [1997] as a technique to declaratively specify animations. Since then, many implementations can be found in other domains [Bainomugisha et al. 2013], including workflow modelling [Reynders et al. 2014]

Where TOP focusses on collaboration, FRP is focussed on data dependencies and behaviours of them. Similar to TOP, vanilla FRP can give no guarantees on memory usage, but extensions such as arrowised FRP [Nilsson et al. 2002] or modal FRP [Bahr 2022].

Workflow modelling in programming languages is a well-trodden field. With TOP, interactive collaborative systems are programmed using a high abstraction level making it very suitable to model workflows. Van der Aalst et al. [2003] and Russell et al. [2006] provide an overview of languages and frameworks and defines a benchmark.

Constraint-based approaches to model business processes, such as Declare [Montali et al. 2013] and Ampersand [Joosten 2018] follow a different approach to TOP. Their semantics can be characterized with logic-based formalisms like relation algebras. Using these formalisms, one describes rules to keep information in the system consistent. Instead of specifying a workflow, it is automatically derived from this specification. Alternatively, timed Dynamic Condition Response (DCR) graphs [Hildebrandt et al. 2021] are a way of specifying workflows which allow formal verification of both safety and liveness properties.

6 Conclusions

We analysed sixteen real-world TOP applications. Thirteen came from literature, two are internal to iTASK and one is a commercial GIS application. Together, they total 140 kLOC defined in a total of 395 modules. From the combinator usage, several gaps are identified between TOP and iTASK.

We find that over 96.50% of combinator usage of real-world TOP programs can be expressed in TOPHAT. However, as most of these examples make use of recursion, they can only be expressed in an extended version of TOPHAT without termination assurance. In practice, this means the examples can be executed, but TOPHAT's symbolic execution engine is not guaranteed to terminate.

We observed that parallel combinators are used way less than sequential combinators (±16% versus ±59%). Furthermore, we see that parallel combinators typically appear higher up in the task tree, giving some indication that they are used to describe workflow higher up, i.e. are more important. With the new reflect combinator, we capture an extra 6% of all parallel combinator uses in real-world application.

6.1 Future work

Detaching tasks, i.e. separating tasks from their task tree and allowing other task trees to take over the task, is something that is available in iTASK but not in TOPHAT. It would be interesting to see if and what core combinator we would need in order to express this behaviour as well.

SDSS in iTASK are much richer and support combinators as well, it would be interesting to see how they compare to TOPHAT.

Finally, in iTASK, there is an exception mechanism available. For example, a task may throw an exception, this exception bubbles up and is caught by a sequential combinator. All real-world usage of exceptions can easily be simulated by using an `Either` or `Error` type but one. The exception mechanism is used to short-circuit derived combinators such as `whileUnchanged`. Figuring out how to shortcircuit combinators in TOPHAT is future work.

References

Peter Achten. 2013. Why Functional Programming Matters to Me. In *The Beauty of Functional Code*, Peter Achten and Pieter Koopman (Eds.). Vol. 8106. Springer Berlin Heidelberg, Berlin, Heidelberg, 79–96. doi:10.1007/978-3-642-40355-2_7

Peter Achten, Jurriën Stutterheim, László Domoszlai, and Rinus Plasmeijer. 2014. Task oriented programming with purely compositional interactive scalable vector

graphics. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*. ACM, New York, NY, USA, 7.

Kalev Alpernas, Yotam M. Y. Feldman, and Hila Peleg. 2020. The wonderful wizard of LoC: paying attention to the man behind the curtain of lines-of-code metrics. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 146–156. doi:10.1145/3426428.3426921 event-place: Virtual, USA.

Patrick Bahr. 2022. Modal FRP for all: Functional reactive programming without space leaks in Haskell. *Journal of Functional Programming* 32 (2022), e15. doi:10.1017/S0956796822000132

Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A survey on reactive programming. *ACM Comput. Surv.* 45, 4 (Aug. 2013), 1–34. doi:10.1145/2501654.2501666 Place: New York, NY, USA Publisher: Association for Computing Machinery.

Jost Berthold. 2011. Orthogonal Serialisation for Haskell. In *Implementation and Application of Functional Languages*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 38–53.

Tom H. Brus, Marko C.J.D. van Eekelen, Maarten O. van Leer, and Marinus J. Plasmeijer. 1987. Clean — A language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–384.

László Domoszlai and Tamás Kozsik. 2013. Clean Up the Web! In *The Beauty of Functional Code: Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday*, Peter Achten and Pieter Koopman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 133–150. doi:10.1007/978-3-642-40355-2_10

László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014a. Editlets: type-based, client-side editors for iTasks. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/2746325.2746331 event-place: Boston, MA, USA.

László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014b. Parametric Lenses: Change Notification for Bidirectional Lenses. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*. ACM, New York, NY, USA, 1–11. doi:10.1145/2746325.2746333 event-place: Boston, MA, USA.

László Domoszlai and Rinus Plasmeijer. 2015. Tasklets: Client-Side Evaluation for iTask3. In *Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*, Viktória Zsók, Zoltán Horváth, and Lehel Csató (Eds.). Springer International Publishing, Cham, 428–445. doi:10.1007/978-3-319-15940-9_11

Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *ACM SIGPLAN Notices*, Vol. 32. ACM, New York, NY, USA, 263–273.

Jeroen Henrix, Rinus Plasmeijer, and Peter Achten. 2012. GiN: A Graphical Language and Tool for Defining iTask Workflows. In *Trends in Functional Programming*, Ricardo Peña and Rex Page (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 163–178.

Thomas T. Hildebrandt, Håkon Normann, Morten Marquard, Søren Debois, and Tijs Slaats. 2021. Decision Modelling in Timed Dynamic Condition Response Graphs with Data. In *Business Process Management Workshops - BPM 2021 International Workshops, Rome, Italy, September 6-10, 2021, Revised Selected Papers (Lecture Notes in Business Information Processing, Vol. 436)*, Andrea Marrella and Barbara Weber (Eds.). Springer, 362–374. doi:10.1007/978-3-030-94343-1_28

Stef Joosten. 2018. Relation Algebra as programming language using the Ampersand compiler. *J. Log. Algebraic Methods Program.* 100 (2018), 113–129. doi:10.1016/J.JLAMP.2018.04.002

Tosca Klijnsma and Tim Steenvoorden. 2022. Semantic Equivalence of Task-Oriented Programs in TopHat. In *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13401)*, Wouter Swierstra and Nicolas Wu (Eds.). Springer, Cham, 100–125. doi:10.1007/978-3-031-21314-4_6

Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A Task-Based DSL for Microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*. ACM Press, Vienna, Austria, 1–11. doi:10.1145/3183895.3183902

Pieter Koopman, Rinus Plasmeijer, and Peter Achten. 2011. An Executable and Testable Semantics for iTasks. In *Implementation and Application of Functional Languages*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 212–232.

Bas Lijnse. 2013a. Evolution of a Parallel Task Combinator. In *The Beauty of Functional Code: Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday*, Peter Achten and Pieter Koopman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 193–210. doi:10.1007/978-3-642-40355-2_14

Bas Lijnse. 2013b. *TOP to the rescue: task-oriented programming for incident response applications*. UB Nijmegen, Nijmegen. OCLC: 833851220.

Bas Lijnse. 2022. Modeling Real World Crisis Management Plans with C2Sketch.. In *ISCRAM 2022 Conference Proceedings – 19th International Conference on Information Systems for Crisis Response and Management*. ISCRAM, Brussels, Belgium, 404–413. event-place: Tarbes, France.

Bas Lijnse. 2024. Toppyt. https://gitlab.com/baslijnse/toppyt

Bas Lijnse, Jan Martin Jansen, Rinus Plasmeijer, and others. 2012. Incidone: A task-oriented incident coordination tool. In *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM*, Vol. 12. ISCRAM, Brussels, Belgium, 1–5.

Bas Lijnse and Rinus Plasmeijer. 2009. iTasks 2: iTasks for End-users. In *International Symposium on Implementation and Application of Functional Languages*. Springer, Cham, 36–54.

Bas Lijnse and Rinus Plasmeijer. 2021. Typed Directional Composable Editors in iTasks. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20)*. Association for Computing Machinery, New York, NY, USA, 115–126. doi:10.1145/3462172.3462197 event-place: Canterbury, United Kingdom.

Mart Lubbers. 2023. *Orchestrating the Internet of Things with Task-Oriented Programming*. PhD Thesis. Radboud University Press, Nijmegen. doi:10.54195/9789493296114.

Mart Lubbers and Peter Achten. 2024. Clean for Haskell Programmers. https://arxiv.org/abs/2411.00037 _eprint: 2411.00037.

Mart Lubbers, Pieter Koopman, Adrian Ramsingh, Jeremy Singer, and Phil Trinder. 2023. Could Tierless Languages Reduce IoT Development Grief? *ACM Trans. Internet Things* 4, 1 (Feb. 2023), 1–35. doi:10.1145/3572901 Place: New York, NY, USA Publisher: ACM.

Marco Montali, Federico Chesani, Paola Mello, and Fabrizio Maria Maggi. 2013. Towards data-aware constraints in declare. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, Sung Y. Shin and José Carlos Maldonado (Eds.). ACM, 1391–1396. doi:10.1145/2480362.2480624

Nico Naus and Tim Steenvoorden. 2020. Generating Next Step Hints for Task Oriented Programs Using Symbolic Execution. In *Trends in Functional Programming - 21st International Symposium, TFP 2020, Krakow, Poland, February 13-14, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12222)*, Aleksander Byrski and John Hughes (Eds.). Springer, Cham, 47–68. doi:10.1007/978-3-030-57761-2_3

Nico Naus, Tim Steenvoorden, and Markus Klinik. 2019. A symbolic execution semantics for TopHat. In *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*, Jurriën Stutterheim and Wei-Ngan Chin (Eds.). ACM, New York, USA, 1–11. doi:10.1145/3412932.3412933

Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional reactive programming, continued. In *Proceedings of the ACM SIGPLAN workshop on Haskell - Haskell '02*. ACM Press, Pittsburgh, Pennsylvania, 51–64. doi:10.1145/581690.581695

Arjan Oortgiese, John van Groningen, Peter Achten, and Rinus Plasmeijer. 2017. A Distributed Dynamic Architecture for Task Oriented Programming. In *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages*. ACM, New York, NY, USA, 7. event-place: Bristol, UK.

Rinus Plasmeijer, Peter Achten, and Pieter Koopman. 2007. iTasks: executable specifications of interactive work flow systems for the web. *ACM SIGPLAN Notices* 42, 9 (2007), 141–152.

Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, and Thomas van Noort. 2009. An iTask Case Study: A Conference Management System. In *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 306–329. doi:10.1007/978-3-642-04652-0_7

Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, Thomas van Noort, and John van Groningen. 2011. iTasks for a Change: Type-safe Run-time Change in Dynamically Evolving Workflows. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '11)*. ACM, New York, NY, USA, 151–160. doi:10.1145/1929501.1929528 event-place: Austin, Texas, USA.

Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming (PPDP '12)*. ACM, New York, NY, USA, 195–206. doi:10.1145/2370776.2370801 event-place: Leuven, Belgium.

Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. 2021. *Clean Language Report version 3.1*. Technical Report. Institute for Computing and Information Sciences, Nijmegen. 127 pages.

Bob Reynders, Dominique Devriese, and Frank Piessens. 2014. Multi-Tier Functional Reactive Programming for the Web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 55–68. doi:10.1145/2661136.2661140 event-place: Portland, Oregon, USA.

Jarrett Rosenberg. 1997. Some misconceptions about lines of code. In *Proceedings Fourth International Software Metrics Symposium*. IEEE, New York, NY, USA, 137–142. doi:10.1109/METRIC.1997.637174 event-place: Albuquerque, NM, USA.

Nick Russell, Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, and Nataliya Mulyar. 2006. *Workflow Control-Flow Patterns: A Revised View*. Technical Report BPM-06-22. BPM Center. http://www.workflowpatterns.com/documentation/documents/BPM-

06-22.pdf

Tim Steenvoorden. 2022. *TopHat: Task-oriented programming with style.* Ph. D. Dissertation. Radboud University, Nijmegen, the Netherlands.

Tim Steenvoorden and Nico Naus. 2024. Dynamic TopHat: Start and Stop Tasks at Runtime. In *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages (IFL '23)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3652561.3652574 event-place: Braga, Portugal.

Tim Steenvoorden, Nico Naus, and Markus Klinik. 2019a. TopHat: A Formal Foundation for Task-Oriented Programming. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (PPDP '19)*. ACM, New York, NY, USA, 1–13. doi:10.1145/3354166.3354182 event-place: Porto, Portugal.

Tim Steenvoorden, Nico Naus, and Markus Klinik. 2019b. TopHat: A formal foundation for task-oriented programming. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, New York, USA, 17:1–17:13. doi:10.1145/3354166.3354182

Jurriën Stutterheim. 2017. *A Cocktail of Tools: Domain-Specific Languages for Task-Oriented Software Development.* UB Nijmegen, Nijmegen. https://hdl.handle.net/2066/181589

Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. 2018. Maintaining Separation of Concerns Through Task Oriented Software Development. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Vol. 10788. Springer International Publishing, Cham, 19–38. doi:10.1007/978-3-319-89719-6

Jurriën Stutterheim, Rinus Plasmeijer, and Peter Achten. 2014. Tonic: An infrastructure to graphically represent the definition and behaviour of tasks. In *International Symposium on Trends in Functional Programming*. Springer, Cham, 122–141.

Wil M.P. van der Aalst, Arthur H.M. ter Hofstede, Bartosz. Kiepuszewski, and Alistair P. Barros. 2003. Workflow Patterns. *Distributed and Parallel Databases* 14, 1 (July 2003), 5–51. doi:10.1023/A:1022883727209

Fixture Layout Optimization in Wood Industry: A Case Study

Anna Vitali
anna.vitali7@unibo.it
University of Bologna
Bologna, Italy, IT

Roberto Amadini
roberto.amadini@unibo.it
University of Bologna
Bologna, Italy, IT

Maurizio Gabbrielli
maurizio.gabbirelli@unibo.it
University of Bologna
Bologna, Italy, IT

ABSTRACT

In the wood industry, workpieces are commonly processed using machines equipped with movable bars and suction cups that serve as fixtures to stabilize the workpiece and reduce vibrations during machining. Optimizing the layout of these fixtures is essential to ensure both stability and high production quality. In collaboration with a manufacturing company, we propose a declarative approach to fixture layout optimization based on Constraint Programming (CP). Our method automatically generates fixture configurations that guarantee workpiece stability while adhering to geometric, safety, and resource constraints. The model is implemented in MiniZinc to offer flexibility and compatibility with various solving technologies. Empirical evaluations on a real-world case study prove that our approach significantly improves both the quality and usability of the company's existing software tool.

KEYWORDS

Fixture layout optimization, Declarative Programming, Moments of inertia, Wood industry

ACM Reference Format:
Anna Vitali, Roberto Amadini, and Maurizio Gabbrielli. 2025. Fixture Layout Optimization in Wood Industry: A Case Study. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

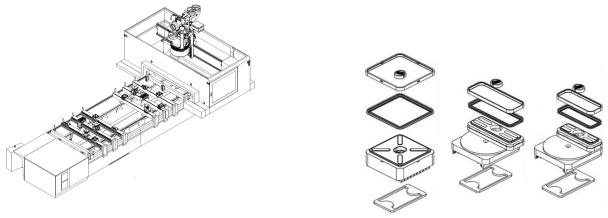
1 INTRODUCTION

In the wood industry, maintaining the stability of workpieces during operations such as cutting, milling, and contouring is crucial for ensuring the quality of the finished product [9]. This stability is typically achieved by strategically placing fixtures to minimize vibrations and oscillations. However, determining the optimal placement of fixtures—while ensuring safety during cutting operations and keeping the workpiece in place—is a complex challenge [22].

Fixture design and layout optimization has been extensively studied over the years [2, 4, 17]. Various approaches have been proposed to solve the layout optimization problem, including iterative algorithms that focus on geometric constraints [6, 23], linear or mixed-integer programming (LP/MIP) formulations [14, 15, 20], and genetic algorithms [11, 13].

PPDP '25, September 10–11, 2025, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX



(a) Movable bars worktable **(b) Suction cups**

Figure 1: Characteristics of a center of work

Although numerous studies on fixture layout optimization exist in the literature, few focus on suction cup fixtures with characteristics similar to those used in the wood industry, as considered in our case. Most existing research has primarily targeted the metalworking sector, particularly emphasizing optimization strategies based on the '3-2-1' locating principle [11, 14, 15]. This approach ensures maximum rigidity using a minimal number of fixture elements—specifically, six points of contact: three on the base plane to restrict movement along the Z-axis, two on a side plane to restrict Y-axis movement, and one on another side plane to restrict movement along the X-axis [11]. Unfortunately, this principle is not applicable to our scenario, where fixtures can only be positioned beneath the workpiece and must comply with specific geometric constraints.

This study, conducted in collaboration with a company based in northern Italy[1], focuses on a specialized type of work center equipped with a worktable featuring movable bars. These bars can shift horizontally in both directions while adhering to predefined safety constraints (Fig. 1a). They serve as mounting bases for suction cup fixtures, which securely hold the workpieces from below. Each fixture has a square base that allows vertical movement along the bar into which it is inserted and can also be rotated. The suction cups themselves may be either square or rectangular in shape (Fig. 1b).

To define an effective fixture layout, it is essential to consider the geometry of the workpiece, its placement on the working surface, and the nature of the machining operations. These may include through or non-through processes and may or may not involve the removal of specific components [6, 15, 19].

For cutting operations, fixtures must not be placed beneath areas where through cuts will occur, in order to prevent damage. Furthermore, resource constraints must be taken into account: fixtures cannot overlap, are available in limited numbers, and must be mounted directly on the bars. Within these constraints, the central challenge is to allocate the available resources efficiently while ensuring that the workpiece is adequately supported.

[1] Company name omitted for privacy reasons.

This paper proposes a declarative model to address the fixture layout problem, based on the *Constraint Programming* (CP) paradigm [18]. The model is implemented in the MiniZinc language [16], offering both flexibility and generality by being parametric with respect to input data (e.g., fixture types or worktable geometry) and solver-independent. As a result, it can be solved using any solver that supports MiniZinc—not only CP-based solvers, but also alternatives such as Mixed-Integer Programming (MIP) and Constraint Logic Programming (CLP) solvers.

We compare the results obtained using two different objective functions: one based on the exact computation of the principal moments of inertia, and the other using the Manhattan distance between pairs of fixtures. Results show that the proposed approach outperforms the company's current tool. While the solution based on principal moments of inertia offers slightly higher precision, it requires significantly more computation time than the Manhattan distance-based approach. Therefore, the latter is more suitable for practical, real-time applications.

Paper structure. Section 2 presents the formulation of the proposed model. Section 3 discusses the results of a case study comparing the two objective functions. Section 4 concludes the paper.

2 MODEL FORMULATION

In this section we describe the CP model in terms of input parameters, variables, constraints, and objective function(s). The model is *parametric* because it is defined in terms of its parameters: changing the input data does not affect the model itself.

2.1 Parameters

The input parameters shaping a particular problem instance are:

- The width $\texttt{WTab}$ and height $\texttt{HTab}$ of the worktable
- The minimal horizontal distance $\texttt{WMin}$ between two fixtures placed on different bars.
- The number $\texttt{NTypes}$ of different fixture types, and the number $\texttt{NFix}_i$ of available fixtures for each type $i \in [1..\texttt{NTypes}]$.
- The total number of fixtures $\texttt{NFix} = \sum_{i=1}^{\texttt{NTypes}} \texttt{NFix}_i$.
- The width $\texttt{WFix}_i$ and height $\texttt{HFix}_i$ of each fixture of type i
- The workpiece shape, represented by linear inequalities of the form $ax + by + c \leq 0$ defining the workpiece region.

2.2 Variables

To determine which fixtures to use and their positions on the worktable, we define three arrays of variables x, y, t of size $\texttt{NFix}$ such that, for $i = 1, \ldots, \texttt{NFix}$:

- (x_i, y_i) is the coordinate of the lower-left corner of the i-th fixture, with $x_i \in [0..\texttt{WTab}]$ and $y_i \in [0..\texttt{HTab}]$
- t_i is the type of the i-th fixture, with $t_i \in [0..\texttt{NTypes}]$

We also define a variable $n_i \in [0..\texttt{NFix}_{t_i}]$ for each $i \in [0..\texttt{NTypes}]$ counting the number of fixtures of type i, and a variable n corresponding to the total number of fixtures employed (see Sect. 2.3.1).

2.3 Constraints

2.3.1 Fixture constraints. For each fixture of type $i \in [1..\texttt{NTypes}]$, we impose the CP *global* constraint [1]:

$$n_i = count(t, i)$$

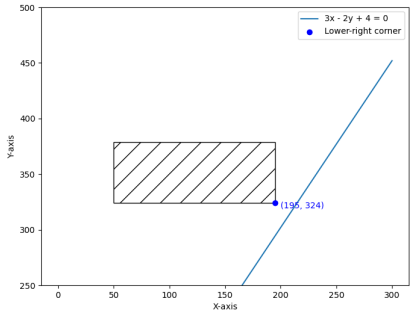


Figure 2: Geometric constraint for fixtures placement

counting the number of fixtures in t with type i, and set $n = \sum_i n_i$.

To break symmetries, we impose that $(x_{i-1}, y_{i-1}) \preceq (x_i, y_i)$ for $i \in [2..\texttt{NFix}]$ through the *lex_chain* global constraint. Because we may not use all the available fixtures, we enforce the invariant:

$$(i > n) \iff (x_i = \texttt{WTab} \ \wedge \ y_i = \texttt{HTab} \ \wedge \ t_i = 0)$$

2.3.2 Geometric constraints. Fixtures must be placed within the boundaries of the workpiece region, must not interfere with areas affected by through-machining operations, must not overlap with each other, and must be positioned on top of the bars that form the workplane. The workpiece region is defined by linear inequalities of the form $ax + by + c \leq 0$. Therefore, each coordinate (x_i, y_i) with $i \leq n$ must satisfy the condition $ax'_i + by'_i + c \leq 0$ where:

$$x'_i = \begin{cases} x_i & \text{if } a \leq 0 \\ x_i + \texttt{WFix}_{t_i} & \text{if } a > 0 \end{cases} \qquad y'_i = \begin{cases} y_i & \text{if } b \leq 0 \\ y_i + \texttt{HFix}_{t_i} & \text{if } b > 0 \end{cases}$$

Indeed, according to the sign of coefficients a and b, we only have to check that one vertex of the rectangular fixture is within the workpiece region. For example, if the inequality is $3x - 2y + 4 \leq 0$, we impose $3(x_i + \texttt{WFix}_{t_i}) - 2y_i + 4 \leq 0$ to enforce the lower-right corner within the hyperplane defined by that inequality (Fig. 2). If so, also all the other fixture points must be within that region.

To encode the no-overlap constraint between fixtures we employ the *diffn* global constraint [1]:[2]

$$diffn(x, y, [\texttt{WFix}_{t_i} \mid i \in [1..n]], [\texttt{HFix}_{t_i} \mid i \in [1..n]]).$$

This constraint ensures that each rectangle with origin (x_i, y_i), width $\texttt{WFix}_{t_i}$ and height $\texttt{HFix}_{t_i}$ is not overlapping. Notably, this constraint can be extended to ensure that no fixture is placed within regions designated for through-machining operations.

Finally, we guarantee that the horizontal distance between two fixtures placed on different bars is at least $\texttt{WMin}$. Formally, for each $i \in [1..n-1]$ we impose that:

$$x_i \neq x_{i+1} \implies x_i + \texttt{WFix}_{t_i} + \texttt{WMin} \leq x_{i+1}.$$

2.4 Objective function

To maximize the stability of the workpiece through fixture placement, we analyze the moments of inertia of the fixture layout.

[2] We use the *non-strict* version of diffn, to ignore the $\texttt{NFix} - n$ unemployed fixtures.

The moment of inertia is a geometric property that quantifies a structure's resistance to bending and torsional forces. A higher moment of inertia corresponds to increased rigidity and reduced deflection under load [3, 10]. In our case, the overall moments of inertia of the fixture system are computed using the coordinates of the four vertices $v_{i,1}, \ldots, v_{i,4}$ of each fixture i, defined as:

$$v_{i,1} = (x_i, y_i), \quad v_{i,2} = (x_i, y_i + \texttt{HFix}_{t_i})$$
$$v_{i,3} = (x_i + \texttt{WFix}_{t_i}, y_i + \texttt{HFix}_{t_i}), \quad v_{i,4} = (x_i + \texttt{WFix}_{t_i}, y_i)$$

We denote with $v_{i,j,x}$ and $v_{i,j,y}$ the coordinates of $v_{i,j}$ on the x- and y- axis, respectively. We define the centroid of fixture i as $(cx_i, cy_i) = \left(x_i + \frac{\texttt{WFix}_{t_i}}{2}, y_i + \frac{\texttt{HFix}_{t_i}}{2}\right)$ and its area as $A_i = \texttt{WFix}_{t_i} \cdot \texttt{HFix}_{t_i}$.

We then compute the following:

- Absolute moments of inertia J'_x, J'_y, J'_{xy}:

$$J'_x = \sum_{i=1}^{n} \left(\frac{1}{12} \sum_{j=1}^{4} \left(v_{i,j,y}^2 + v_{i,j,y} \cdot v_{i,j+1,y} + v_{i,j+1,y}^2 \right) \cdot \left(v_{i,j,x} \cdot v_{i,j+1,y} - v_{i,j+1,x} \cdot v_{i,j,y} \right) \right),$$

$$J'_y = \sum_{i=1}^{n} \left(\frac{1}{12} \sum_{j=1}^{4} \left(v_{i,j,x}^2 + v_{i,j,x} \cdot v_{i,j+1,x} + v_{i,j+1,x}^2 \right) \cdot \left(v_{i,j,x} \cdot v_{i,j+1,y} - v_{i,j+1,x} \cdot v_{i,j,y} \right) \right),$$

$$J'_{xy} = \sum_{i=1}^{n} \left(\frac{1}{24} \sum_{j=1}^{4} \left(v_{i,j,x} \cdot v_{i,j+1,y} + 2v_{i,j,x} \cdot v_{i,j,y} + 2v_{i,j+1,x} \cdot v_{i,j+1,y} + v_{i,j+1,x} \cdot v_{i,j,y} \right) \cdot \left(v_{i,j,x} \cdot v_{i,j+1,y} - v_{i,j+1,x} \cdot v_{i,j,y} \right) \right)$$

- Overall center of gravity (gx, gy):

$$gx = \frac{\sum_{i=1}^{n} A_i \cdot cx_i}{A}, \qquad gy = \frac{\sum_{i=1}^{n} A_i \cdot cy_i}{A}$$

  where $A = \sum_{i=1}^{n} A_i$ is the total area.
- Moments of inertia J_x, J_y, J_{xy} w.r.t. the centroidal reference system (x, y) passing through (gx, gy):

$$J_x = J'_x - A \cdot gy^2, \quad J_y = J'_y - A \cdot gx^2, \quad J_{xy} = J'_{xy} - A \cdot gx \cdot gy$$

- Principal moments of inertia I and J:

$$I = \frac{J_x + J_y}{2} - \frac{1}{2}\sqrt{(J_x - J_y)^2 + 4J_{xy}^2}$$
$$J = \frac{J_x + J_y}{2} + \frac{1}{2}\sqrt{(J_x - J_y)^2 + 4J_{xy}^2}$$

The goal is to maximize the sum of the principal moments of inertia, i.e., to maximize the *objective function*

$$f_{iner} = I + J.$$

Computing f_{iner} involves complex non-linear operations, making the problem challenging to model and solve efficiently [12]. Moreover, operations like division and square root make the CP modeling much harder as very few CP solvers support float variables.

Notably, the literature reports that the optimal fixture placement tends to occur along the perimeter of the workpiece [11, 14, 15, 19, 23]. An alternative objective function consists therefore in positioning fixtures as far apart as possible, thereby improving stability, by also promoting the use of larger fixtures when available. We

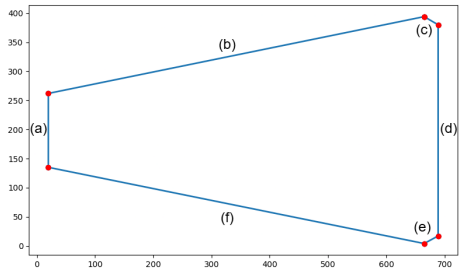


Figure 3: Stair step of a spiral staircase

therefore devised an alternative objective function f_{dist} defined by the sum of the pairwise Manhattan distances between the centroids of the placed fixtures, plus the sum of their individual dimensions:

$$f_{dist} = \sum_{1 \le i < j \le n} \left(|cx_i - cx_j| + |cy_i - cy_j| \right) + \sum_{k=1}^{n} \left(\texttt{WFix}_{t_k} + \texttt{HFix}_{t_k} \right).$$

We use the Manhattan distance instead of the Euclidean distance to formulate the problem using integer variables only, thus avoiding float variables. This is particularly important for CP solvers, which are optimised for integer reasoning and rely on constraint propagation within finite domains. The term $\sum_{k=1}^{n} \left(\texttt{WFix}_{t_k} + \texttt{HFix}_{t_k} \right)$ is used in place of the total fixture area $\sum_{k=1}^{n} A_i$ to maintain consistency in dimensional scale.

3 CASE STUDY

We evaluated our approach on a real case where the workpiece is a stair step of a spiral staircase. The workpiece area, shown in Fig. 3, is defined by the following inequalities:

$$(a): \ x \ge 20 \qquad (d): \ x \le 689$$
$$(b): \ y \le 0.20x + 257.91 \qquad (e): \ y \ge 0.54x + -356.21$$
$$(c): \ y \le -0.58x + 781.92 \qquad (f): \ y \ge -0.20x + 139.06$$

The width and height of the worktable are respectively $\texttt{WTab} = 700$ and $\texttt{HTab} = 400$. The minimal distance between fixtures is $\texttt{WMin} = 140$. We have two different types of fixtures available:

- Five square fixtures of dimension 145×145 mm.
- Five rectangular fixtures of dimension 145×55 mm.

So, $\texttt{NFix}_1 = \texttt{NFix}_2 = 5$ with $\texttt{WFix}_1 = \texttt{HFix}_1 = \texttt{WFix}_2 = 145$ and $\texttt{HFix}_2 = 55$.

We set an upper bound NMax on the number of fixtures (i.e., we impose $n \le \texttt{NMax}$) and we evaluated different values of NMax in the range [2..10]. We assume a minimum number of 2 fixtures (i.e., $n \ge 2$) as placing only one fixture is not a realistic case.

We defined two models, one using the objective function f_{iner} and the other using f_{dist}. Both were implemented in Minizinc [16]. For the model using f_{dist}, four solvers were evaluated: Chuffed [5], Gecode [21], OR-Tools [7] and Gurobi [8]. For the f_{iner} model, we restricted our evaluation to solvers that support floating-point variables, namely Gecode and Gurobi.

Table 1: Results with f_{iner}. Bold fold denotes an optimal f_{iner}-value found before the timeout.

		NMax								
Solver	**Metric**	2	3	4	5	6	7	8	9	10
Gurobi	Runtime [ms]	14250	189000	90000	258000	timeout	timeout	timeout	timeout	timeout
	f_{iner}-value	**265072**	**379990**	**379990**	**387506**	389034	N/A	N/A	N/A	N/A

Table 2: Results using f_{dist}. Bold fold denotes the f_{iner}-value corresponding to a f_{dist}-optimal solution.

		NMax								
Solver	**Metric**	2	3	4	5	6	7	8	9	10
Chuffed	Runtime [ms]	1566	904	2039	3333	3700	2533	4873	5222	5593
	f_{iner}-value	**264416**	**238480**	**258584**	**354968**	**380905**	**380905**	**380905**	**380905**	**380905**
Gurobi	Runtime [ms]	503	745	961	1098	1540	1856	4674	5534	10840
	f_{iner}-value	**264416**	**238350**	**258584**	**308459**	**380905**	**380905**	**380905**	**380905**	**380905**
Gecode	Runtime [ms]	549	18292	timeout	timeout	timeout	timeout	timeout	timeout	timeout
	f_{iner}-value	**264416**	**239170**	258584	258584	258584	373308	258584	258584	258584
OR-Tools	Runtime [ms]	882	7804	260000	timeout	timeout	timeout	timeout	timeout	timeout
	f_{iner}-value	**264416**	**216912**	**224606**	**334565**	334565	338321	373467	373467	367863

All experiments were conducted with a time limit of 300 seconds on a 64-bit Windows machine equipped with an Intel Core i5 CPU and 16 GB of RAM.

Table 1 and 2 show the results using f_{iner} and f_{dist}, respectively, across different values of NMax. Table 1 does not include results for Gecode, as it fails to find any feasible solution within the time limit. Gurobi, being a MIP solver that handle continuous variables more naturally, can find optimal solutions for $\texttt{NMax} \leq 5$ but cannot find any solution for $\texttt{NMax} > 6$.

Table 2 shows that when using f_{dist} all solvers are able to find at least a feasible solution. In particular, Chuffed and Gurobi solve all instances to optimality. The best performance is achieved by Chuffed, able to find the optimal f_{dist}-value under six seconds in the worst case. Importantly, Table 2 reports the corresponding f_{iner}-values for the best solutions found—rather than the f_{dist}-values—to allow for direct comparison in terms of principal moments of inertia.

Since f_{dist} objective can admit multiple equivalent optimal solutions, small variations (such as swapping two fixtures) may result in distinct configurations with the same objective value. However, these solutions can yield different f_{iner} values, as shown for $\texttt{NMax} \in \{3, 4, 5\}$ in Table 2. Appendix A reports a visual representation of the solutions obtained for each solver and objective function.

As expected, using f_{iner} encourages the placement of larger fixtures, since larger areas result in higher moments of inertia. Consequently, not all available fixtures are always used—for instance, with $\texttt{NMax} = 4$, only 3 fixtures are placed. Solutions generated using f_{dist} typically yield lower f_{iner} values, but are computed much faster, making them particularly suitable for large workpieces or higher fixture counts.

4 CONCLUSIONS

This paper presents a declarative model for fixture layout optimization in the wood industry, developed using Constraint Programming (CP) and designed for compatibility with multiple solvers.

The model addresses the practical requirement of ensuring workpiece stability during machining, originally formulated using an objective function based on the principal moments of inertia. However, this formulation introduces significant non-linearity, increasing the computational complexity of the problem.

To improve tractability, an alternative objective function was proposed based on the Manhattan distance between pairs of fixtures. This approach avoids complex computations while still capturing essential spatial relationships, resulting in a substantial reduction in solving time. Both models outperform the company's current tool, which is based solely on geometric constraints and requires the user to manually define both the number and type of fixtures for each workpiece, facilitating the introduction of errors.

With our solution the operator is relieved of the responsibility of making these choices. The number and type of fixtures to be used is determined automatically along with their optimal position. This results in a reduction of the time required to prepare the work table and minimises errors that can occur.

Future work includes investigating symmetry-breaking constraints and developing advanced search heuristics to further improve CP solver efficiency. Additionally, local search techniques could be applied to refine solutions—such as allowing fixture rotation—starting from the CP-generated layout. Another promising direction involves integrating fixture layout optimization with cut-path planning. While this would increase the model's complexity, it is expected to yield more realistic, accurate solutions and enhance overall productivity.

ACKNOWLEDGMENTS

This work is partly supported by the FREEDA project, funded by the frameworks PRIN (MUR, Italy) and Next Generation EU (CUP: I53D23003550006) and by PNRR - M4C2 - Investimento 1.3 e 3.3, Partenariato Esteso PE00000013 - FAIR - Future Artificial Intelligence Research - Spoke 8 Pervasive AI.

REFERENCES

[1] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. 2007. Global Constraint Catalogue: Past, Present and Future. *Constraints An Int. J.* 12, 1 (2007), 21–62.

[2] Zhuming M Bi and WJ Zhang. 2001. Flexible fixture design and automation: Review, issues and future directions. *International Journal of Production Research* 39, 13 (2001), 2867–2894.

[3] Peter H Bischoff and Shawn P Gross. 2011. Equivalent moment of inertia based on integration of curvature. *Journal of Composites for Construction* 15, 3 (2011), 263–273.

[4] Iain Boyle, Yiming Rong, and David C Brown. 2011. A review and analysis of current computer-aided fixture design approaches. *Robotics and Computer-Integrated Manufacturing* 27, 1 (2011), 1–12.

[5] Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Gregory Gange, and Kevin Francis. 2018. Chuffed: A Lazy Clause Generation Solver. https://github.com/chuffed/chuffed. Accessed: 2025-05-27.

[6] Minh Duc Do, Younghoon Son, and Hae-Jin Choi. 2018. Optimal workpiece positioning in flexible fixtures for thin-walled components. *Computer-Aided Design* 95 (2018), 14–23.

[7] Google LLC. 2024. OR-Tools: Google's Operations Research Tools. https://github.com/google/or-tools. Accessed: 2025-05-27.

[8] Gurobi Optimization, LLC. 2025. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[9] Michael R Jackson, Robert M Parkin, and Neil Brown. 2002. Waves on wood. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture* 216, 4 (2002), 475–497.

[10] İlker Kalkan. 2013. Deflection prediction for reinforced concrete beams through different effective moment of inertia expressions. *International Journal of Engineering Research and Development* 5, 1 (2013), 1–1.

[11] Necmettin Kaya. 2006. Machining fixture locating and clamping position optimization using genetic algorithms. *Computers in industry* 57, 2 (2006), 112–120.

[12] Sven Leyffer, Jeff Linderoth, James Luedtke, Andrew Miller, and Todd Munson. 2009. Applications and algorithms for mixed integer nonlinear programming. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012014.

[13] Guilong Li, Shichang Du, Delin Huang, Chen Zhao, and Yafei Deng. 2019. Elastic mechanics-based fixturing scheme optimization of variable stiffness structure workpieces for surface quality improvement. *Precision Engineering* 56 (2019), 343–363.

[14] RJ Menassa and WR DeVries. 1991. Optimization methods applied to selecting support positions in fixture design. (1991).

[15] Richard Thomas Meyer and Frank W Liou. 1997. Fixture analysis under dynamic machining. *International Journal of Production Research* 35, 5 (1997), 1471–1489.

[16] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. 2007. MiniZinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 529–543.

[17] Sertac Pehlivan and Joshua D Summers. 2008. A review of computer-aided fixture design with respect to information support requirements. *International Journal of Production Research* 46, 4 (2008), 929–947.

[18] Francesca Rossi, Peter Van Beek, and Toby Walsh. 2006. *Handbook of constraint programming*. Elsevier.

[19] Utpal Roy and Jianmin Liao. 1999. Geometric reasoning for re-allocation of supporting and clamping positions in the automated fixture design system. *IIE transactions* 31, 4 (1999), 313–322.

[20] QA Sayeed and EC De Meter. 1999. Mixed-integer programming model for fixture layout optimization. (1999).

[21] Christian Schulte, Mikael Lagerkvist, and Guido Tack. 2006. Gecode. *Software download and online material at the website: http://www. gecode. org* (2006), 11–13.

[22] Sriramkumar Sivasubramanian. 2019. Optimization and simulation of formed suction cup to position aerospace CFRP parts during milling.

[23] Aaron S Wallack and John F Canny. 1997. Planning for modular and hybrid fixtures. *Algorithmica* 19 (1997), 40–60.

A VISUAL COMPARISON OF FIXTURES LAYOUTS

Tables 3 and 4 present the visual results obtained from the different solvers, evaluated across varying values of `NMax`.

We can see that the solutions when using f_{dist} function look very similar. However, when the number of fixtures is limited to three, Chuffed, Gecode, and OR-Tools tend to place a smaller fixture on the left side of the workpiece, below its center. This placement notably affects the moments of inertia (as shown in Table 2). In contrast, Gurobi aligns the fixture with the center of the workpiece, adopting on the right side two large fixtures, resulting in a higher value of f_{iner}.

Table 3: Visual results for solvers' solutions

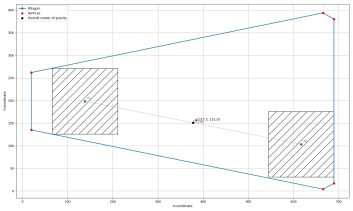
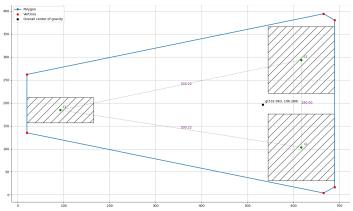
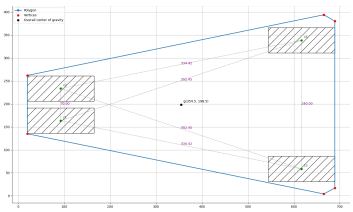
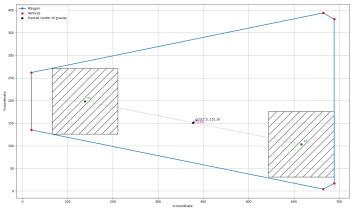
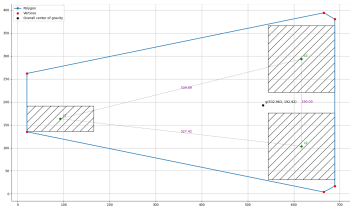
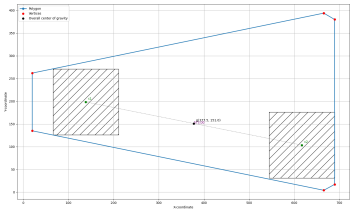
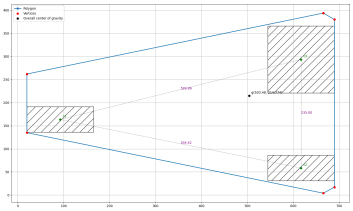
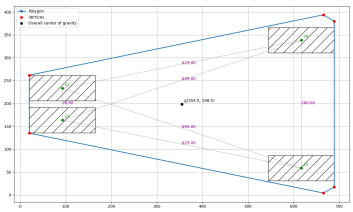
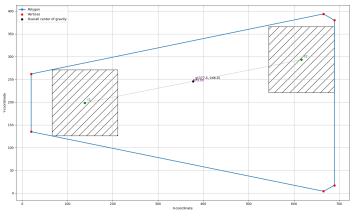
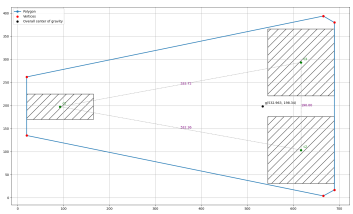
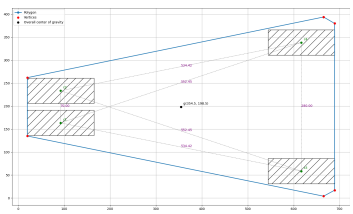
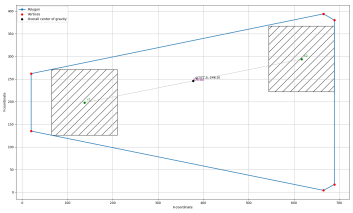
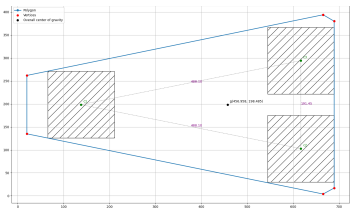
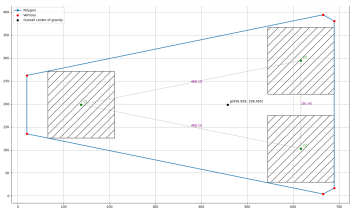
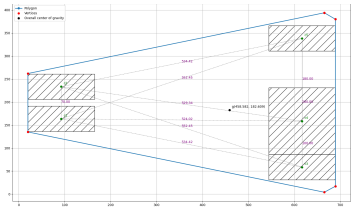
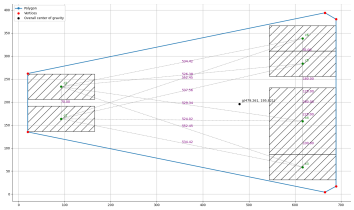
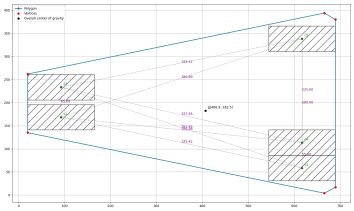
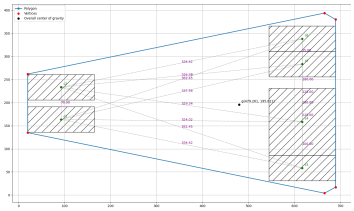
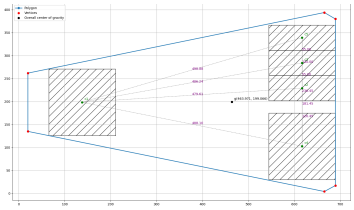
	NMax		
Solver	2	3	4
Chuffed			
Gecode (f_{dist})			–
OR-Tools			
Gurobi (f_{dist})			
Gurobi (f_{iner})			

Table 4: Visual results for solvers' solutions

	NMax	
Solver	5	6-10
Chuffed		
OR-Tools	–	–
Gurobi (f_{dist})		
Gurobi (f_{iner})		–

Determinism Types for Functional Logic Programming

Michael Hanus
Kiel University
Kiel, Germany
mh@informatik.uni-kiel.de

Kai-Oliver Prott
Kiel University
Kiel, Germany
kpr@informatik.uni-kiel.de

Abstract

Functional logic programming languages, such as Curry, integrate features of functional and logic paradigms, in particular, demand-driven deterministic evaluation from functional programming with non-deterministic search from logic programming. Though useful for programming, this combination can lead to unintended results and subtle bugs. To support programming with this powerful computation model, this paper proposes a method to detect unintended non-determinism at compile time. For this purpose, we propose determinism types to approximate the determinism behavior of functions and expressions. In contrast to standard types in strongly typed languages, determinism types do not restrict the set of admissible programs but support the programmer and programming tools in reasoning about functional logic programs, e.g., to enforce determinism in top-level I/O operations. We present the motivation behind this approach, discuss core concepts of functional logic programming and Curry, and outline methods to check for determinism through type-based analysis.

CCS Concepts

• **Software and its engineering** → **Compilers**; **Functional languages**; **Semantics**; *Multiparadigm languages*; • **Theory of computation** → *Logic and verification.*

Keywords

functional programming, logic programming, functional logic programming, determinism, type systems, Curry, static analysis, non-determinism

ACM Reference Format:
Michael Hanus and Kai-Oliver Prott. 2025. Determinism Types for Functional Logic Programming. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Functional logic programming [10, 21] combines the demand-driven and, for particular classes of programs, optimal [29] evaluation model from functional programming with the expressiveness and flexibility of logic programming. This synergy, exemplified by the language Curry [27], introduces powerful programming abstractions, including higher-order functions, unification with evaluable functions, and non-deterministic computations. However, these strengths also bring challenges in reasoning about program behavior, especially concerning determinism.

Determinism refers to the property that a program or operation always produces the same output for the same input, without involving any implicit choices or ambiguity. In functional logic languages, non-determinism is a first-class feature, but many programs are intended to be deterministic. Accidental non-determinism can lead to subtle bugs, unpredictable results, or even run-time errors, especially when interacting with I/O.

For instance, consider the following definition from the Haskell library `Data.Maybe`:

```
isNothing :: Maybe a → Bool
isNothing Nothing = True
isNothing _       = False
```

Although the last equation defining `isNothing` does not seem meaningful on its own, it is reasonable in Haskell due to its sequential pattern matching from top to bottom. This is different in Curry where *all* rules defining an operation are taken into account in order to search for values or answers, as in logic programming. Consequently, `isNothing Nothing` returns both `True` and `False` in Curry, where the order is unspecified. Hence, the top-level evaluation of

```
print (isNothing Nothing)
```

leads to a run-time error since one cannot duplicate the world to enable non-deterministic printing.

To avoid such errors and to express determinism behavior in programs, we propose to add *determinism type signatures* as an optional language extension to Curry. These signatures allow developers to annotate functions with their expected determinism behavior which is statically checked by the compiler. For example, in the following Curry code snippet, the operation `append` is annotated to indicate that it is deterministic:

```
append :? Det → Det → Det
append :: [a] → [a] → [a]
append [] ys = ys
append (x:xs) ys = x : append xs ys
```

Since the result of a call to `append` contains the values of both arguments, the result of `append` is deterministic only if the input lists are deterministic as well. That is why the determinism signature (annotated by "`:?`") contains a `Det` for both arguments, which are separated by the arrow →. The similarity to the standard type signature is intentional and helps programmers reason about both the shape and the determinism of their functions in a uniform way. When `append` is called with free variables, Curry's demand-driven evaluation strategy instantiates these variables only as much as

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

required for the pattern matching on the arguments. Although this leads to more than one result, we consider free variables as (potentially) non-deterministic. Thus, the signature of `append` still holds.

For some functions, the determinism status of the result does not depend on the arguments. For example, the operation `const` is deterministic even if its second argument is non-deterministic:

```
const :? Det → Any → Det
const :: a   → b   → a
const x _ = x
```

Here, the second argument can be non-deterministic (`Any`), but since it is not used in the result, the operation as a whole remains deterministic. In order to avoid the definition of multiple determinism types for a single operation, we assume that `Det` is a subtype of `Any` so that we can also apply `const` to a second argument of type `Det`.

The disuse of arguments is not the only possibility to deal with non-deterministic arguments in deterministic operations. An important feature of logic-oriented languages is the encapsulation of non-deterministic computations. Similarly to Prolog's `findall` predicate, Curry supports an operation `allValues` [5] which returns the list of all values of the argument. With our proposal, the signature of `allValues` is

```
allValues :? Any → Det
allValues :: a   → [a]
```

Similarly, one can also specify the determinism behavior of set functions [9] which provide an alternative method to encapsulate search.

Another use of determinism types is to enforce determinism in instances of type classes [46]. For example, it is reasonable to require that the function `show` that converts a value to a string is deterministic, since data is usually shown in I/O operations. This might be expressed by the type class definition

```
class Show a where
  show :? Det → Det
  show :: a → String
```

Now, providing an instance of `Show` with a non-deterministic implementation like the following would be rejected by the compiler:

```
instance Show Bool where
  show True = "True"
  show _    = "False"
```

This instance is non-deterministic since it returns `"True"` and `"False"` for the input `True` due to the overlapping patterns, similarly to the operation `isNothing` discussed above. This is a common mistake in Curry by Haskell programmers, where one intends to define a deterministic function but accidentally created a non-deterministic one by using the underscore pattern as a catch-all case.

Non-determinism is essential for problems involving search, inference, and symbolic computation so that we do not wish to restrict its use. However, in practice, many operations are intended to be deterministic—even in languages that support non-determinism. Detecting when operations inadvertently exhibit non-deterministic behavior is valuable both for correctness and efficiency. This is our main motivation to add determinism types to functional logic programs.

Adding determinism information to the type system offers multiple benefits:

- **Intentional clarity**: Determinism type annotations serve as documentation of intent, signaling to other developers and tools that an operation should behave deterministically.
- **Static guarantees**: The determinism checker can detect violations early, reducing the need for dynamic debugging.
- **Optimization opportunities**: Knowing an operation is deterministic allows compilers to generate more efficient code and eliminate some run-time checks.
- **Safe I/O**: Ensuring determinism in I/O contexts prevents run-time crashes due to undefined behavior. We can achieve this by emitting a warning or error when the program entry point (e.g., the `main` function) does not have type `Det`.

Ultimately, determinism types and their analysis aim to provide a compositional and reliable way to reason about program behavior without sacrificing the flexibility of non-deterministic programming. Because our proposal is lightweight, compositional, and compatible with existing Curry implementations, it is suitable for practical use in both interactive and compiled environments.

In the next section, we review the core concepts of functional logic programming and Curry. Section 3 introduces a typed kernel language which is used in this paper. The syntax of determinism types and their meaning is shown in Section 4. Section 5 presents the rules for determinism typing. Some formal results about determinism typing and properties of programs with determinism types are given in Section 6. We also provide a full formalization of these results in the *Rocq* proof assistant [13]. Section 7 surveys potential applications of determinism types. Section 8 presents some extensions of the current framework. An implementation of determinism types is sketched in Section 9 before we conclude with a survey of related work in Section 10.

2 Functional Logic Programming and Curry

Functional logic programming languages unify the most important principles from functional programming (demand-driven evaluation, strong typing with parametric polymorphism, higher-order functions) and logic programming (non-determinism, computing with partial information, constraints). The functional logic language Curry[1] has a Haskell-like syntax, while its operational semantics [1] incorporates ideas from logic programming such as non-determinism and handling of free variables. The combination of reduction and instantiation of free variables is called narrowing [36, 40]. Curry is based on the needed narrowing strategy [6] which is optimal w.r.t. number of evaluation steps and computed solutions.

A distinguishing feature of Curry, different from Haskell, is the selection of rules. Whereas Haskell applies the (from top to bottom) first matching rule to reduce a subexpression, Curry uses all applicable rules. Thus, the choice operator "?" is predefined by

```
(?) :: a → a → a
x ? y = x
x ? y = y
```

[1] www.curry-lang.org

so that the values of "`True ? False`" are `False` and `True`.

In order to encapsulate non-deterministic computations and to return all values of an expression involving non-determinism in a single data structure, there is an operation `allValues` already introduced in Section 1. Since the result of this operation depend on the concrete evaluation strategy [15], Curry provides *set functions* [9] to encapsulate search in an evaluation-independent manner. For each defined operation f, f_S denotes its corresponding set function. f_S encapsulates the non-determinism caused by evaluating f except for the non-determinism caused by evaluating the arguments to which f is applied. For instance, consider the operation

```
hamlet b = b ? not b
```

Then the expression `hamlet True` non-deterministically yields both values `True` and `False`, whereas the expression $\texttt{hamlet}_S$ `True` evaluates to (an abstract representation of) the set `{True,False}`. Since the non-determinism of arguments is not encapsulated by a set function, the expression $\texttt{hamlet}_S$ `(hamlet True)` evaluates to the equivalent sets `{True,False}` and `{False,True}`.

To support the implementation of larger applications, Curry has many additional features not described here, like modules, which are similar to Haskell, and monadic I/O [45] for declarative input/output. The latter is based on the idea that operations performing I/O are considered as functions manipulating the state of the external world (or environment). Since a non-deterministic choice duplicates the state of a computation but the "world" (terminal, file system, etc.) cannot be duplicated, non-determinism and I/O are incompatible. Thus, Curry implementations emit a run-time error when non-deterministic I/O operations are applied. The exact mechanism for detecting this error is implementation-specific and often involves checking for multiple results during evaluation. Such errors can be avoided by encapsulating non-deterministic expressions.

Example 1 (Non-determinism and I/O). Consider the following definitions:

```
coin :: Int
coin = 0 ? 1

f :: Int → Int
f x = x + coin

main = print (f coin)
```

Since `f coin` has more than one value (0, 1, or 2), it is unclear which of the values should be printed. Therefore, as discussed above, the execution of `main` leads to a run-time error. To avoid this, one has to encapsulate the argument of `print`, where it must be decided by the programmer which of the values (e.g., all, one, minimum) should be printed. For instance,

```
main = print (allValues (f coin))
```

prints the list of all values. Note that the use of the set function of `f`, as in

```
main = print (fS coin)
```

is not sufficient, since the non-determinism of the argument `coin` is not encapsulated.

This example shows that it would be helpful to have a tool which helps to encapsulate the right expressions. This is the motivation for adding determinism types to Curry.

3 A Typed Kernel Functional Logic Language

Curry is a powerful language with a lot of syntactic sugar — influenced by Haskell but with additional features, like functional patterns [7] and set functions as mentioned above. Moreover, Curry's type system is inspired by Haskell so that it offers parametric polymorphism and type (constructor) classes. In order to reduce the features covered by determinism typing, we consider a restricted kernel language similarly to operational descriptions [1], implementations [24], and analyses [25] of functional logic programs. This kernel language, called FlatCurry, consists of top-level operations without patterns, i.e., patterns are translated into case expressions and choice operators. Although we defined "?" by two rules with overlapping left-hand sides in the previous section, one can also consider "?" as a primitive choice operator and translate overlapping rules into non-overlapping rules with explicit choice operators "?" in the right-hand sides. For instance, overlapping rules like

```
f True = True
f _    = False
```

can be translated to the single non-overlapping rule

```
f x = case x of True  → True
                False → True ? False
```

Since any functional logic program can be transformed into this form [3], FlatCurry can be and is often used as an intermediate language to compile Curry programs. The syntax of FlatCurry as used in this paper is summarized in Figure 1. It is a standard higher-order language with case expressions for pattern matching and choices and introduction of free variables for logic programming.

For instance, the operation `append` shown in Section 1 can be represented in FlatCurry as

```
append = λxs →
           λys → case xs of
                   { []   → ys
                   ; z:zs → z : append zs ys }
```

An intricate point with determinism types is the handling of functional values. For instance, consider the following operations which use the identity operation `id` and the Boolean negation `not` in two different ways.

```
idNot1 x = id x ? not x

idNot2 = id ? not
```

When these operations are applied to a Boolean value, both values `True` and `False` are non-deterministically returned. What should be the determinism type of these operations? The type of `idNot1` is clearly

```
idNot1 :? Det → Any
```

The determinism type of `idNot2` is not obvious. It could be

```
idNot2 :? Det → Any
```

(since it is a functional value returning different values when it is applied) or also

$$
\begin{array}{lcll}
P & ::= & F_1 \ldots F_m & \text{(program)} \\
F & ::= & f = e & \text{(function definition)} \\
e & ::= & x & \text{(variable)} \\
 & | & C & \text{(data constructor)} \\
 & | & e_1\ e_2 & \text{(application)} \\
 & | & \lambda x \to e & \text{(abstraction)} \\
 & | & e_1\ or\ e_2 & \text{(choice)} \\
 & | & let\ x\ free\ in\ e & \text{(free variable)} \\
 & | & case\ e\ of\ \{p_1 \to e_1; \ldots; p_k \to e_k\} & \text{(case expression, } k > 0\text{)} \\
p & ::= & C\ x_1\ \ldots\ x_n & \text{(pattern, } n \geq 0\text{)}
\end{array}
$$

Figure 1: Syntax of the kernel language FlatCurry

```
idNot2 :? Any
```

(since it returns non-deterministically two functional values). Since the operational behavior of both definition is the same, one could consider the determinism types `Det → Any` and `Any` as equivalent.

However, one can distinguish the two different non-deterministic definitions by encapsulation. The expression `allValues idNot1` returns a list with the single element `idNot1` (since the argument cannot be further evaluated), whereas `allValues idNot2` encapsulates the non-deterministic choice of `idNot2` so that a list with two elements (`id` and `not`) is returned. Hence, we get different results when we apply these two expressions to the list `[True]` with the operator `<*>` of the class `Applicative` [33]:

```
allValues idNot1 <*> [True]  ⇝ [True]?[False]
allValues idNot2 <*> [True]  ⇝ [True,False]
```

This is also one of the reasons why `allValues` is considered "unsafe" from a declarative point of view, whereas set functions, which do not encapsulate higher-order values, are a declarative method to encapsulate non-deterministic computations [9].

These complications can be avoided by using determinism types only for values of first-order data types. Interestingly, this restriction fits to the `Data` proposal for Curry [26]: the type of free variables must satisfy the `Data` class constraint which expresses that such variables do not contain functional values. This ensures a constructive method to non-deterministically enumerate values for free variables.

In the following, we consider a typed intermediate language which distinguishes between first-order and higher-order values. For the sake of simplicity, we do not consider polymorphic types but refer to Section 8.2 for a discussion about polymorphism. *First-order types* are Booleans and lists:

$$D \quad ::= \quad \texttt{Bool} \mid \texttt{List}\ D$$

These can be easily extended to other types, like integers, pairs, etc. Note that types in D are types without functional components, i.e., in Curry these are the types which are instances of class `Data` [26] (see also Section 8.1). For some of our examples, we assume an extension of the type system with some additional first-order types.

General *types* are first-order types combined with functional types:

$$T \quad ::= \quad D \mid T \to T$$

Since we are not interested in type inference, we assume that variables introduced in FlatCurry programs (as parameters in abstractions or free variables) are annotated with a type. The typing rules for this language are shown in Figure 2. There, we denote by Γ a *type environment* which is a (partial) mapping from identifiers (taken from a set of names X) to types:

$$\Gamma : X \to T$$

As usual, we denote by $\Gamma[x \mapsto \tau]$ the type environment Γ' with $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ for all $x \neq y$.

Note that Figure 2 contains separate case typing rules for each first-order type. Although Figure 2 defines the type of a case expression with a complete list of branches, one can also omit some branches if they are undefined (as in the case of partially defined operations). Note that our system can be easily extended to support more data types.

A program $P = f_1 = e_1; \ldots; f_k = e_k$ is *type correct* if there is a type environment

$$\Gamma = \{f_1 \mapsto \tau_1, \ldots, f_k \mapsto \tau_k\}$$

such that $\Gamma \vdash e_i :: \tau_i$ $(i = 1, \ldots, k)$ is derivable by the typing rules in Figure 2. In the following, we consider only type-correct programs.

4 Determinism Types

Similarly to standard types in strongly typed programming languages, expressions and operations can be annotated with determinism types. Whereas standard types approximate the set of possible values to which an expression evaluates, a determinism type approximates the determinism behavior. For instance, if an expression is correctly annotated as deterministic (*Det*), its evaluation will never yield a choice between two expressions. Otherwise, the expression is annotated with *Any* meaning that it is potentially non-deterministic. The determinism type system propagates these annotations through applications and abstractions in order to annotate every expression and operation with a determinism type.

In order to associate meaningful determinism types to operations, we allow also function arrows in determinism types. Thus, the *syntax of determinism types* is:

$$\Delta \quad ::= \quad Det \mid Any \mid \Delta \to \Delta$$

Note that the type $Any \to Any$ is formally allowed but has the same information as $Det \to Any$ (see below). Hence it will never

$$\textit{Var}\quad \Gamma \vdash x :: \Gamma(x)$$

$$\textit{True}\quad \Gamma \vdash \texttt{True} :: \texttt{Bool}$$

$$\textit{False}\quad \Gamma \vdash \texttt{False} :: \texttt{Bool}$$

$$\textit{Nil}\quad \Gamma \vdash \texttt{[]} :: \texttt{List}\ \tau \qquad \tau \in D$$

$$\textit{Cons}\quad \Gamma \vdash \texttt{(:)} :: \tau \to \texttt{List}\ \tau \to \texttt{List}\ \tau \qquad \tau \in D$$

$$\textit{App}\quad \frac{\Gamma \vdash e_1 ::\ \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 ::\ \tau_1}{\Gamma \vdash e_1\ e_2 ::\ \tau_2}$$

$$\textit{Abs}\quad \frac{\Gamma[x \mapsto \tau_1] \vdash e ::\ \tau_2}{\Gamma \vdash \lambda x : \tau_1 \to e ::\ \tau_1 \to \tau_2}$$

$$\textit{Choice}\quad \frac{\Gamma \vdash e_1 ::\ \tau \quad \Gamma \vdash e_2 ::\ \tau}{\Gamma \vdash e_1\ ?\ e_2 ::\ \tau}$$

$$\textit{Free}\quad \frac{\Gamma[x \mapsto \tau] \vdash e :: \tau'}{\Gamma \vdash \textit{let } x : \tau \textit{ free in } e :: \tau'} \qquad \tau \in D$$

$$\textit{CaseBool}\quad \frac{\Gamma \vdash e ::\ \texttt{Bool} \quad \Gamma \vdash e_1 ::\ \tau \quad \Gamma \vdash e_2 ::\ \tau}{\Gamma : \textit{case } e \textit{ of } \{\texttt{True} \to e_1; \texttt{False} \to e_2\} ::\ \tau}$$

$$\textit{CaseList}\quad \frac{\Gamma \vdash e ::\ \texttt{List}\ \tau \quad \Gamma \vdash e_1 ::\ \tau' \quad \Gamma[x_1 \mapsto \tau, x_2 \mapsto \texttt{List}\ \tau] \vdash e_2 ::\ \tau'}{\Gamma : \textit{case } e \textit{ of } \{\texttt{[]} \to e_1; \texttt{x1:x2} \to e_2\} ::\ \tau'} \qquad \tau \in D$$

Figure 2: Type system of FlatCurry

be inferred by our inference system. The intended meaning of these types is:

- *Det*: The expression evaluation (w.r.t. standard Curry semantics, see Section 6) is deterministic, i.e., it never yields a choice between two values.
- *Any* → *Det*: If a function of this type is applied to some argument, its evaluation is deterministic, either because the argument is not used or all non-determinism is encapsulated.
- *Det* → *Det*: If a function of this type is applied to a *Det* argument, its evaluation is deterministic, otherwise the evaluation is arbitrary
- *Det* → *Any*: If a function of this type is applied to some argument, its evaluation is arbitrary.
- *Any* → *Any*: As stated before, this type behaves like the previous *Det* → *Any*.
- *Any*: The expression evaluates (w.r.t. standard Curry semantics) in a possibly non-deterministic manner, i.e., it possibly yields choices between values. It might also be a function that behaves in such a way.

As discussed above, we do not associate the determinism type *Det* to a functional value, i.e., an expression of type $\tau_1 \to \tau_2$ has determinism type *Any* or is of the form $\delta_1 \to \delta_2$. We say a determinism type δ is *compatible* to a type $\tau \in T$ if δ has the same shape as τ (except for *Any*) but first-order types are replaced by determinism types:

- For a type $\tau \in D$, *Det* or *Any* is compatible to τ.
- *Any* is compatible to type $\tau_1 \to \tau_2$.
- The determinism type $\delta_1 \to \delta_2$ is compatible to $\tau_1 \to \tau_2 \in T$ if δ_i is compatible to τ_i $(i = 1, 2)$.

Note that if one were to extend the type system with more first-order types, the compatibility relation needs to be extended accordingly.

The motivation for allowing compatibility between *Any* and types of the form $\tau_1 \to \tau_2$ is that we want the determinism type of `idNot2` to be *Any*. Consequently, *Any* must be considered as compatible to its type `Bool → Bool`. This design choice has the advantage of keeping the determinism typing of a choice simple: we always assign the determinism type *Any* to a choice, regardless of whether the result is a function or a first-order value. If, instead, we require `idNot2` to have the determinism type *Det* → *Any*, the determinism typing of choices become unnecessarily complex due to the consideration of functional types. As we will see, we consider in the determinism typing of applications that the function to be applied might have type *Any*.

Here are some examples for determinism types. Consider the identity operation `id` defined by

```
id x = x
```

with type $\tau \to \tau$ for some type $\tau \in D$.

`id :?` *Det* → *Det*

is a possible determinism type for `id`. The determinism type

`id :?` *Det* → *Any*

is also possible but less precise.

The choice operation returns one of its arguments so that its evaluation is always non-deterministic:

`(?) :?` *Det* → *Det* → *Any*

$$\delta_1 \sqsubseteq \delta_2 \quad := \quad \begin{cases} \mathit{true} & \text{if } \delta_2 = \mathit{Any} \vee \delta_1 = \delta_2 = \mathit{Det} \\ \delta_{21} \sqsubseteq \delta_{11} \wedge \delta_{12} \sqsubseteq \delta_{22} & \text{if } \delta_1 = \delta_{11} \to \delta_{12} \wedge \delta_2 = \delta_{21} \to \delta_{22} \\ \mathit{false} & \text{otherwise} \end{cases} \qquad \delta_1 \sqcup \delta_2 \quad := \quad \begin{cases} \delta_2 & \text{if } \delta_1 \sqsubseteq \delta_2 \\ \delta_1 & \text{if } \delta_2 \sqsubseteq \delta_1 \\ \mathit{Any} & \text{otherwise} \end{cases}$$

Figure 3: Subtyping relation and supremum of determinism types

Note that this is the type if the choice operation has first-order types as arguments, otherwise the *Det* must be replaced by a compatible determinism type.

As discussed above, the operation `allValues` encapsulates all non-determinism of its argument, but we have to be careful with functional values. Hence, we allow only the encapsulation of first-order values, i.e., `allValues` has the type

`allValues` :: $\tau \to \tau$

for some $\tau \in D$ (we could also consider a family of operations $\texttt{allValues}_\tau$ for each $\tau \in D$). Then its determinism type is

`allValues` :? $\mathit{Any} \to \mathit{Det}$

Consider a function f with type $\tau_1 \to \cdots \to \tau_n \to \tau$ and $\tau_i \in D$ $(i = 1, \ldots, n)$. Then its set function $\texttt{f}_S$ encapsulates the non-determinism caused by `f` but not the non-determinism occurring in the arguments. Hence, its determinism type is

$\texttt{f}_S$:? $\mathit{Det} \to \cdots \to \mathit{Det} \to \mathit{Det}$

independent of the determinism type of `f` (which has usually the target type *Any*).

These examples illustrate how determinism types can precisely capture the intended behavior of functions, and how they interact with higher-order functions and encapsulation.

5 Determinism Typing

Determinism typing is defined w.r.t. an environment which associates determinism types to identifiers, e.g., bound variables or operations defined in the program. For this purpose, we denote by a *determinism type environment* Δ a (partial) mapping from identifiers (taken from a set of names X) to determinism types:

$$\Delta : X \to \Delta$$

We assume that the determinism types associated to operations defined in a program are compatible to the types defined in Section 3, i.e., if $\Gamma(f) = \tau$, for some $\tau \in T$, then $\Delta(f)$ is compatible to τ. With respect to such a determinism type environment, the determinism typing rules (Figure 4) derive judgements of the form

$$\Delta \vdash e : \delta$$

stating that the expression e has determinism type δ with respect to the determinism type environment Δ.

For the application of an operation to an argument, it must be checked whether the argument's determinism type satisfies the requirements of the operation on its argument. For this purpose, we define a *subtyping relation* on determinism types in Figure 3. For example, if a non-deterministic operation of type $\mathit{Det} \to \mathit{Any}$ is required, we can also use an operation of type $\mathit{Det} \to \mathit{Det}$ or $\mathit{Any} \to \mathit{Det}$. However, we cannot use a non-deterministic operation of type $\mathit{Det} \to \mathit{Any}$ where a deterministic operation of type $\mathit{Det} \to \mathit{Det}$ is expected. Note that any operation can be used where *Any* is required. This simplifies the typing of the choice operator as shown below.

We explain the intended meaning of the various rules for determinism typing shown in Figure 4. To keep the rules compact, we sometimes denote by $\overline{o_n}$ a sequence of objects $o_1, \ldots, o_n$.

Rule Var is obvious. The requirement that x is in the domain of Δ is to ensure that every variable needs to have a corresponding binding site. This is always the case for correct (FlatCurry) programs.

Constructors (rule Cons) are deterministic but might become non-deterministic when applied to some non-deterministic argument, see the subsequent rules for application.

Rule AppAny covers the case when there is no precise information about the operation, in particular, if the expression e_1 involves a choice between two operations. App is the usual application rule where determinism subtyping on arguments is required for the reasons discussed above. For instance, the well-known operation `map` takes a function as a first argument and applies it to all elements of the list provided as the second argument. The FlatCurry definition typed for Boolean lists is

```
map :: (Bool → Bool) → List Bool → List Bool
map =
  λf : Bool → Bool →
    λxs : List Bool →
      case xs of { []   → []
                 ; y:ys → f y : map f ys }
```

If $\Delta(\texttt{map}) = (\mathit{Det} \to \mathit{Det}) \to \mathit{Det} \to \mathit{Det}$ and e is the right-hand side of the definition of `map`, then we derive with the rules in Figure 4 that $\Delta \vdash e :? \Delta(\texttt{map})$. We denote this judgement by the determinism type

```
map :? (Det → Det) → Det → Det
```

Similarly, we derive that `map not [True]` has determinism type *Det*. The operation `hamlet`, as defined in Section 2, has determinism type

```
hamlet :? Det → Any
```

Since $\mathit{Det} \to \mathit{Any}$ is not a subtype of $\mathit{Det} \to \mathit{Det}$ required by `map`, rule App infers the determinism type *Any* for the expression `map hamlet [True]`.

Rule Choice simply infers the determinism type *Any*. It could be made more precise by considering different kinds of function types, but such a more complex technical treatment does not pay off in realistic examples. Rule Free considers the case of a free (logic) variable. Since such a variable evaluates or can be bound to different values, it is typed as non-deterministic. One could make it more precise for the special case that the type τ contains only a

$$\textsc{Var}\quad \Delta \vdash x :? \delta \qquad \text{if } x \in Dom(\Delta) \text{ and } \delta = \Delta(x)$$

$$\textsc{Cons}\quad \Delta \vdash C :? Det \to \cdots \to Det \to Det \qquad \text{if } C \text{ is an } n\text{-ary constructor}$$

$$\textsc{AppAny}\quad \frac{\Delta \vdash e_1 :? Any \quad \Delta \vdash e_2 :? \delta}{\Delta \vdash e_1\ e_2 :? Any}$$

$$\textsc{App}\quad \frac{\Delta \vdash e_1 :? \delta_{11} \to \delta_{12} \quad \Delta \vdash e_2 :? \delta_2}{\Delta \vdash e_1\ e_2 :? \delta_3} \qquad \delta_3 = \begin{cases} \delta_{12} & \text{if } \delta_2 \sqsubseteq \delta_{11} \\ Any & \text{otherwise} \end{cases}$$

$$\textsc{Abs}\quad \frac{\Delta[x \mapsto \delta_1] \vdash e :? \delta_2}{\Delta \vdash \lambda x : \tau \to e :? \delta_1 \to \delta_2} \qquad \delta_1 \text{ is compatible to } \tau$$

$$\textsc{Choice}\quad \frac{\Delta \vdash e_1 :? \delta_1 \quad \Delta \vdash e_2 :? \delta_2}{\Delta \vdash e_1\ ?\ e_2 :? Any}$$

$$\textsc{Free}\quad \frac{\Delta[x \mapsto Any] \vdash e :? \delta}{\Delta \vdash let\ x : \tau\ free\ in\ e :? \delta}$$

$$\textsc{Case}\quad \frac{\Delta \vdash e :? \delta \quad \Delta[\overline{x_{n_i} \mapsto \delta}] \vdash e_i :? \delta_i\ (i = 1, \ldots, k)}{\Delta \vdash case\ e\ of\ \{\overline{p_k \to e_k}\} :? \delta \sqcup \delta_1 \sqcup \ldots \sqcup \delta_k} \qquad \text{where } p_i = C\ \overline{x_{n_i}} \text{ and } \delta \in \{Det, Any\}$$

Figure 4: Determinism typing rules

single value. Since such a use of free variables does not occur in real programs, we omit this slight improvement.

To understand rule Case, consider the determinism type δ of the discriminating argument. Since we assume that the source program is well-typed, case distinctions over functional values are not allowed. Under the assumption that determinism types are compatible to standard types (see Section 6), a functional determinism type cannot occur for the discriminating argument e. Thus, the side condition $\delta \in \{Det, Any\}$ is not a real restriction but only written for clarity.

Concerning the determinism types for pattern variables, note that lists (type `List`) contain only first-order values as elements. Even if we extend the simple type system to allow general data structures with functional values as components (as in Curry source programs), any constructor applied to a function will still be assigned *Any* (see rules Cons and the subtyping in App). Therefore, the pattern variables in a case expressions will always have determinism types *Det* or *Any*, i.e., there is no need to consider other determinism types for them in rule Case. Therefore, rule Case checks the determinism type of each branch under the assumption that the pattern variables have the same determinism type (*Det* or *Any*) as the discriminating argument e. Under these assumptions, the determinism types of e and the right-hand sides e_i of all branches are combined by the least upper bound operation w.r.t. the subtyping relation. In particular, if the determinism type of e is *Any*, the entire case expression must have type *Any* as well. Thus, one could add a simpler inference rule for this specific case, but we put the general rule Case since this is advantageous to establish the formal results about determinism typing presented in Section 6.

Given a program $P = f_1 = e_1; \ldots; f_k = e_k$, we say $f_i :? \delta_i$ $(i = 1, \ldots, k)$ are *correct determinism types* if $\Delta \vdash e_i :? \delta_i$ $(i = 1, \ldots, k)$ can be inferred by the rules of Figure 4 w.r.t. the determinism type environment

$$\Delta = \{f_1 \mapsto \delta_1, \ldots, f_k \mapsto \delta_k\}$$

As we will see later, there always exists a correct determinism typing for a well-typed program. Thus, determinism typing is optional and not a restriction on admissible programs—in contrast to standard typing.

We discuss the determinism typing of a few more examples. We have already shown the determinism typing of `map` above. Consider the expression

```
map (id ? not) [True]
```

By rule Choice, `(id ? not)` has type *Any*. Since *Any* is not a subtype of the type $Det \to Det$ required for the first argument of `map`, rule App infers the type *Any* for `map (id ? not)`. Now rule AppAny infers the type *Any* for the entire expression. This shows how we can apply operations with a deterministic type also on non-deterministic operations and obtain the type *Any*.

As a final example, consider the expression

```
map not (allValues (hamlet True))
```

Due to the determinism type of `hamlet` shown above, the subterm `(hamlet True)` has type *Any*. Since this is a subtype of the argument type required by `allValues`, the subterm `allValues (hamlet True)` has type *Det* by rule App. Due to the deterministic types of `map` and `not`, the entire expression has type *Det*. Actually, this expression evaluates to the single result `[False,True]`.

6 Correctness of Determinism Typing

The inference system for determinism typing possesses several desirable properties, ensuring that it behaves predictably and is suitable for compile-time program analysis. As mentioned above, we consider in the following programs and expressions which are correctly typed w.r.t. the rules in Figure 2 under a type environment Γ. In the formal statements, we need to extend the notion of type compatibility to type contexts. Therefore, we define a determinism type context Δ to be compatible to a type context Γ if, for all variable

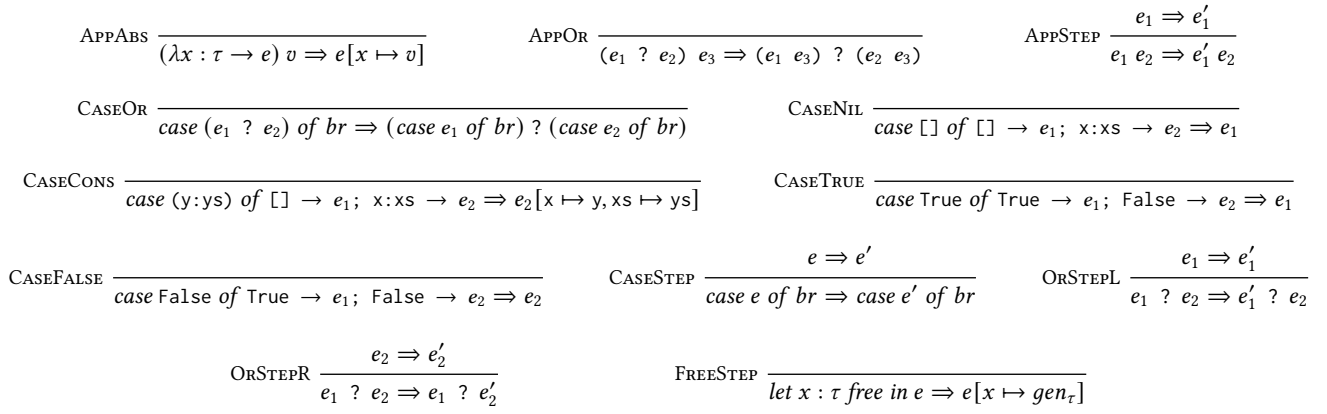


Figure 5: Small-step evaluation rules for expressions

names v where $\Gamma(v)$ is defined, $\Delta(v)$ is compatible to $\Gamma(v)$. That is, the two functions Δ and Γ need to be pointwise compatible.

The first important property is the completeness of determinism typing. This property ensures that the inference system can assign a determinism type to every valid expression in the language. This is crucial since our extension is not meant to restrict the set of admissible programs but rather provides a way to annotate and analyze expressions. Note that this property is not entirely trivial since not every expression can be assigned type *Any*.

Theorem 2 (Typing Completeness). *For all contexts Δ that are compatible to Γ, and all expressions e that are well-typed under Γ, there exists a determinism type δ such that $\Delta \vdash e\ :?\ \delta$ is derivable w.r.t. the determinism typing rules.*

The next interesting property is the preservation of type annotations. The preservation property guarantees that type annotations are stable under evaluation, i.e., evaluation does not introduce any non-determinism not captured by determinism types. The formal statement of this property requires the notion of an evaluation step from an expression e to e', denoted $e \Rightarrow e'$. The formal definition of this relation is shown in Figure 5.

The functional core of evaluation steps is straightforward. The rules AppAbs, CaseCons, and FreeStep use substitutions on expressions. Substitution proceeds structurally through the expression but does not replace occurrences of variables that are bound by inner declarations. For example, in $(\lambda x \to x)[x \mapsto e]$, the x inside the abstraction is not replaced by e, since substitution stops at the binding site of x. Note that there are no unbound variables since free variables are explicitly introduced.

For the evaluation of logical aspects, it should be noted that evaluation is deterministic, i.e., non-deterministic choices are represented by choice structures headed by the choice operator "?" instead of evaluating it to one of its arguments, as in [1]. Thus, non-deterministic choices are kept in a data structure, similarly to pull-tabbing [2, 4] or the Verse calculus [12]. If an expression headed by a choice operator has to be evaluated, as in the rules AppOr and CaseOr, the choice is moved from the argument position to the top. Basically, this is the idea of pull-tabbing [2, 4] used in implementations of Curry which support flexible search strategies [14, 16]. In functional logic languages based on the *call-time choice* semantics [30], such as Curry, it is necessary to annotate choice nodes with tags in order to restrict the set of sensible values, as discussed in [4]. For the sake of simplicity, we omit this additional machinery so that our semantics implements the run-time choice semantics [30]. This causes no problem in our case, since run-time choice always computes the same or more values than call-time choice. Hence, if an evaluation is deterministic w.r.t. run-time choice, it is also deterministic w.r.t. call-time choice.

Another point to mention is the handling of free variables. As shown in [8, 19], free variables are conceptually equivalent to non-deterministic operations that evaluate to all values of the type of the free variable. For instance, a free Boolean variable can be replaced by the operation

```
gen_Bool = True ? False
```

This property is exploited in rule FreeStep, where a free variable of type τ is replaced by the operation gen_τ which generates all values of type τ. Here it is essential that τ is a first-order type so that gen_τ can be defined constructively.

Based on this operational semantics, we can formally state the preservation property of determinism typing.

Theorem 3 (Preservation). *Let Δ be a determinism type environment compatible to Γ, e, e' expressions with $e \Rightarrow e'$ and e well-typed under Γ. Furthermore, let δ be a determinism type such that $\Delta \vdash e\ :?\ \delta$ is derivable by the determinism typing rules. Then there exists a determinism type δ' such that $\delta' \sqsubseteq \delta$ and $\Delta \vdash e'\ :?\ \delta'$.*

Finally, we state the main property of our determinism typing. The soundness property ensures that expressions typed as deterministic never evaluate to expressions headed by choices, thereby guaranteeing run-time determinism when promised.

Theorem 4 (Soundness). *Let Δ be a determinism type environment, e, e' expressions with $e \Rightarrow^* e'$, and $\Delta \vdash e\ :?\ Det$ be derivable by the determinism typing rules. Then e' does not contain a choice at the root.*

If a choice node is evaluated, it is either at the root and its arguments are evaluated, or it will be moved to the top by iterated

applications of rules AppOr and CaseOr. Thus, the soundness theorem ensures that a *Det*-typed expression will never evaluate a choice. Of course, choice nodes might occur in unevaluated subexpressions, for example in unapplied lambda abstractions. However, these will not be pulled to the top by the evaluation strategy and are thus not relevant for the soundness property.

Altogether, these properties demonstrate that determinism typing is a sound and practical foundation for program verification and optimization. A complete formalization of the determinism typing rules, along with machine-checked proofs of the properties discussed above, is available at https://github.com/cau-placc/rocq-ndtypes. This formalization is carried out in the *Rocq* proof assistant [13] (formerly known as Coq). In the formal development, we explicitly track well-typedness, carefully manage variable binding and name clashes during substitution, and establish a range of auxiliary lemmas, such as the preservation of determinism types under substitution.

7 Applications

Determinism types have various applications ranging from compile-time debugging to performance enhancement. In the following we discuss some of the potential applications.

Safe I/O execution. As discussed in the introduction and Theorem 1, the combination of non-determinism and I/O is unsafe since it is not clear which of the non-deterministic I/O actions should be applied to some state of the world. Since non-determinism of expressions is a dynamic property, this problem is not detected at compile time so that it leads to a run-time error in most Curry systems. If one requires that expressions of type `IO` have the determinism type *Det*, we avoid run-time crashes and unexpected behavior. For instance, one can execute an interactive I/O operation `main` only if it has determinism type `main :? Det`. As an example, consider the definition

```
main = print coin
```

Since `coin`, as defined in Theorem 1, is non-deterministic, the determinism type system infers the judgement `main :? Any`. Thus, the compiler can reject this program or issue a warning, preventing a run-time error. To resolve this issue, the programmer can explicitly encapsulate the non-determinism and specify how to handle multiple results. For example, they could print only the first value.

```
main = print (head (allValues coin))
```

If an expression is incorrectly inferred by the compiler as potentially non-deterministic, encapsulation can also be used to suppress the warning or error.

Interactive programming environments. Curry systems have an interactive environment to execute expressions, known as a REPL (Read-Eval-Print-Loop). The REPL can restrict the evaluation of I/O expressions based on their determinism type so that run-time errors cannot occur and the user does not experience unexpected side effects as in Prolog systems.

More precise warnings. In order to avoid unintended uses of non-determinism, the front end of Curry systems issue a warning if an operation is defined by overlapping rules. For instance, when processing the definition of `isNothing` as defined in Section 1, the Curry front end shows

```
Warning: Function `isNothing' is potentially
non-deterministic due to overlapping rules
```

To avoid such warnings in intended uses of overlapping rules (as in the definition of "?"), one can turn off the warnings (globally or per module) but then one might miss warnings about unintended uses of overlapping rules. An elegant resort of these conflicting goals are determinism types. If the programmer explicitly annotates an operation with a determinism type, it is checked whether its definition satisfies this determinism type according to the rules in Figure 4. Moreover, no overlapping warning is issued when the target determinism type is *Any*. Thus, the definition

```
(?) :? Det → Det → Any
(?) :: a → a → a
x ? y = x
x ? y = y
```

is accepted without any warning.

Compiler optimizations. Curry implementations that compile to Haskell like KiCS2 [16] and KMCC [24] use determinism information to optimize its code generation. Purely functional (hence, deterministic) parts of a program can be compiled into more efficient code. For instance, the correct handling of non-determinism when combined with lazy evaluation requires the threading of choice identifiers through function calls [4]. This can be avoided if expressions are known to be deterministic.

Program analysis tools. Determinism types can be used in static analysis tools to check program invariants or identify unexpected sources of non-determinism. Determinism analysis is often performed with simple domains for first-order programs, as in [11, 25]. Our determinism types provide a more refined abstraction of determinism information which can be extended in various ways, as discussed in the following section.

8 Extensions

This section describes two different extensions of our basic proposal to add determinism types to source programs.

8.1 Determinism Types in Type Classes

As discussed in Section 1, an extension of the determinism type system is to allow the user to specify that a method of a given type class is only allowed to be deterministic, enforcing instances of that class to adhere to this restriction. This is reasonable for the method `show` of the type class `Show` due to its intended use in I/O operations. Another interesting example is discussed in the following.

In [26] the type class `Data` is proposed with the following definition:

```
class Data a where
  (===) :: a → a → Bool
  aValue :: a
```

Intuitively, the method "`===`" denotes syntactic equality between values of type `a`, and `aValue` non-deterministically enumerates all values of type `a`. Instances of `Data` are automatically derived for all

first-order types, i.e., types without functional components, which correspond to D as defined in Section 3. The motivation of this proposal is to provide syntactic unification and value bindings for free variables only for reasonable types. Since enumerating functional values is useless (and difficult in a strongly typed language), free variables, unification, and search operators have a `Data` context on their types, i.e., they are not parametric polymorphic but overloaded entities.

Although this extension, available in recent Curry systems, is useful to provide better compile-time checks for reasonable programs, it has also a disadvantage for efficient implementations of Curry. To see this, consider the following example where the deterministic operation `isZero` uses the equality operator "`===`":

```
data Peano = Zero | Succ Peano

isZero :: Peano → Bool
isZero p = p === Zero
```

Both the operation `isZero` and the operator "`===`" are deterministic. However, the type class `Data` also contains the non-deterministic operation `aValue`. Since compilers for Curry use dictionary passing to implement type classes [34], the compiled function for `isZero` has an additional argument: a `Data` dictionary with a non-deterministic implementation of `aValue`.

Curry compilers intended to produce efficient code for purely functional computations, like KiCS2 [16] and KMCC [24], use a determinism analysis on the intermediate language in order to compile deterministic operations quite similar to purely functional languages. In the example of `isZero`, the code is not considered as deterministic since it contains as a parameter the `Data` dictionary with the non-deterministic operation `aValue` even though it is not used. Although this seems to be a tiny problem in this example, the use of `Data` contexts in encapsulated subcomputations results in serious memory problems in larger Curry applications where the entire application is compiled to potentially non-deterministic code, although non-determinism is not used or only used in small subcomputations (see also [24] for benchmarks comparing compiled code with and without determinism optimizations).

By adding determinism type annotations to the type class `Data`, we can ensure that the compiler can infer the correct determinism annotation for `isZero`. The new definition of `Data` with determinism annotations would look like this:

```
class Data a where
  (===) :? Det → Det → Det
  (===) :: a → a → Bool

  aValue :? Any
  aValue :: a
```

This extension allows the determinism checker to distinguish between deterministic and non-deterministic operations within the same type class, even without knowing the code of the instances. Since this is correct only if the compiler checks all `Data` instances, this requires an extensions of the compiler based on our determinism types.

8.2 Polymorphic Determinism Types

A downside of a monomorphic determinism type system is that it does not always lead to the desired result for some higher-order functions. For example, consider the following function to flip the order of arguments of a binary operation:

```
flip :? (Det → Det → Det) → Det → Det → Det
flip :: (a   → b   → c)   → b   → a   → c
flip f x y = f y x
```

The annotation of `flip` means that the function is deterministic if the function and other arguments passed to it are deterministic. Now consider the application `flip const`. The determinism type of `const` is $Det \to Any \to Det$ which is actually $Det \to (Any \to Det)$ since "$\to$" associates to the right. Since this is a subtype of the type $Det \to Det \to Det$ required for the first argument of `flip`, rule App infers that `flip const` has determinism type $Det \to Det \to Det$. Hence, we have a loss of precision since the information about the possibility to pass a first non-deterministic argument to `flip const` and still obtain a deterministic result if the last argument is deterministic is lost.

To solve this problem, we can extend the determinism type system to allow for polymorphic types. That way, the determinism annotation of `flip` would be the same as its type signature and the partially applied `flip const` would be inferred as `Any → Det → Det`. By introducing determinism type variables (analogous to type variables), we can express more general determinism signatures:

```
flip :? (a → b → c) → b → a → c
```

This allows the determinism of `flip f x y` to depend precisely on the determinism of `f`, `x`, and `y`, enabling more accurate inference and fewer false positives.

The introduction of determinism type variables comes with the price of a more complex type system and type inference algorithms. Apart from this specific example, it is not clear whether there are more realistic situations where this extension yields more precision. Therefore, we leave this extension for future work.

9 Implementation

We have implemented a prototype of the determinism type system as an extension of the Curry front end used by several Curry systems.[2] The implementation already includes the type class extension discussed in Section 8.1. Since type inference with subtyping is quite hard to implement, even for decidable type systems, our implementation uses a best-effort approach (note that one can always infer a determinism type, though it might lack precision). Our type system does not guarantee principality of types, i.e., it is possible that a program has multiple determinism types where one is not more specific than the other.

Type inference proceeds similar to type inference in a Hindley-Milner type system with type variables to be unified during the inference process. Since subtyping is relevant only in rule App, it is sufficient to check the subtyping relation for function applications. If during type inference, the type of the applied function is a metavariable (i.e., a type that is yet to be determined), we delay

[2]This implementation is available at https://github.com/cau-placc/curry-frontend/tree/det_types

the subtyping check. In the end, each type gets fully instantiated to obtain a monomorphic type.

Our type inference is trivially complete, because we can assign the type *Any* (or a function type with *Any*) whenever inference fails. However, since we have not yet proven that our inference algorithm always infers valid types, we currently plan to check the result using the determinism typing rules in Figure 4 for verification.

The implementation in the existing Curry front end was minimally invasive, as we only needed to add some syntax, a separate new type checking phase, and some small changes to the interface files required for modular compilation.

We leave a full exploration of type inference for our determinism types as future work.

10 Related Work

Our approach relates to various areas: program analysis, since it is intended to analyze the run-time behavior of programs at compile time; type systems, since we add non-standard types to a strongly typed language; gradual typing, since we use a type-based approach which should not restrict the number of admissible programs but ensure run-time conditions for particular types; effect systems, since the non-deterministic behavior of computations can be considered as computations with effects. In the following, we briefly relate our approach to these areas.

10.1 Program Analysis

Abstract interpretation is a program-analysis technique based on computing with abstract domains which classify sets of concrete values [18]. In this sense, our type *Det* corresponds to a set of values without a choice at the root, see the evaluation rules in Figure 5, whereas *Any* just denote all values. Abstract interpretation has a long tradition in logic programming to optimize their execution by approximating information about modes, types, and sharing at compile time [17]. Information about modes can be used to derive information about determinism behavior, since a predicate like append for list concatenation is deterministic if the first argument is ground. In this case, more efficient code can be generated.

Modes are exploited in [20] to check and infer functional computations in Prolog which are a generalization of deterministic computations. There, modes are used to ensure the mutual exclusion of rules of a predicate if a call to this predicate satisfies the given mode. This information is propagated through a call graph to infer functional properties of other predicates. The language Mercury [41] combine modes with a strong type system and determinism annotations to produce highly efficient code. Mercury classifies predicates as deterministic, semi-deterministic, or non-deterministic, which parallels our *Det* and *Any* classifications but with finer granularity. A key difference to our approach is that mode annotations, which are essential to analyze determinism properties of Mercury programs [28], put strong restrictions on the set of admissible programs and require additional efforts when programming with Mercury. Our approach does not require modes and is intended to infer determinism types in general so that the programmer provides determinism type annotations only in certain cases, e.g., to suppress non-determinism warnings (as discussed in Section 7) or put restrictions on instances of type classes, see Section 8.1. Moreover, Mercury is a strict language, while Curry uses lazy evaluation and, thus, needs to account for potential non-determinism in function arguments in a more nuanced way.

Similarly to logic programming, abstract interpretation techniques have been also applied to functional logic programming and Curry. We already mentioned the relevance of information about deterministic subcomputations to produce efficient code so that compilers from Curry into deterministic target languages, like KiCS2 [16] and KMCC [24], integrate a determinism analysis. CASS [25] is a generic analysis system for Curry and provides more than 30 different kinds of program analyses, including a determinism analysis. Since CASS is based on a first-order intermediate language, where partial applications are represented as data constructors by defunctionalization [37], the determinism information for higher-order arguments is less precise in CASS. Moreover, CASS is intended to analyze complete programs whereas determinism types can be integrated in the standard type inference of the front end. For instance, determinism types can be used in IDEs to immediately point to problematic uses of non-determinism in I/O operations.

10.2 Extended Type Systems

The idea to encode determinism behavior in a strongly typed language with an extended type system is not new. For instance, Steimann [42] proposed an extension of the lambda calculus, called simply numbered lambda calculus (SNLC), to include strings of objects to represent multiple values. In order to distinguish parameter-passing mechanism for multiple values (comparable to the difference between call-time choice and run-time choice mentioned above), parameters in lambda abstractions are annotated with number specifiers. The basic number specifiers are ! and ∗, which correspond to our determinism types *Det* and *Any*, and their functional combination, called mapping constraints, to cover higher-order functions. In contrast to our approach, the SNLC requires precise numbering annotations for higher-order arguments where a method to infer them is not mentioned (and probably not intended since the numbering annotations are semantically relevant). This is different from our approach where we allow more flexibility by making *Any* a supertype of any function (see Figure 3). This is important to avoid restrictions by adding determinism types and it paves the way towards the determinism typing of polymorphic operations, as discussed in Section 8. Moreover, we consider a non-strict operational semantics which is necessary to use determinism typing for Curry.

A proposal to add determinism information to standard types can be found in [11] where an operation is considered as deterministic if all computed results are identical. This information is used to improve the efficiency of functional logic programs by avoiding irrelevant computations. In contrast to our approach, these determinism annotations are based on trust since methods to check them are not provided.

10.3 Gradual Typing

The requirement that any valid expression can be assigned a determinism type makes our approach similar to gradually typed languages, where types can be added incrementally. For example, *Type Script* has the type any for cases when one does not know

what type a value might be [35]. In contrast, our determinism types are not a fallback type but rather an annotation that can be used to ensure that a function is deterministic. Gradual typing was formally introduced by Siek and Taha [39] for functional languages, allowing programs to evolve from dynamic to static typing incrementally. One of the most mature implementations of this approach is Typed Scheme (now Typed Racket) [43], which demonstrates how untyped Scheme programs can be gradually converted to statically typed programs with minimal disruption. Similar to our approach with determinism types, Typed Scheme maintains backward compatibility while providing stronger guarantees through optional annotations, requiring no changes to the underlying run-time system.

Particularly relevant to our work is research on gradual typing for logic programming languages. For example, Schrijvers et al. [38] developed a gradual typing system for Prolog. Their system allows programmers to specify varying levels of type information, from fully dynamic to fully static typing. While we share the goal of being able to assign a type to any expression, our approach introduces an additional type system, orthogonal to the standard Curry type system.

10.4 Effect Systems

The integration of type systems with effect and determinism tracking is a well-established area. Effect systems in functional programming languages, such as those proposed for ML or Haskell (e.g., monads), encode side effects in types to control and reason about program behavior.

Lucassen and Gifford's foundational work on polymorphic effect systems [32] introduced the concept of annotating types with effects to track and control computational behaviors. This approach enabled precise reasoning about which effects an expression might produce. Similarly, our determinism types can be viewed as a specialized effect system focused on non-deterministic choice effects.

Monads, popularized by Wadler [44], offer another mechanism to encapsulate and reason about effects in functional languages. While monads in Haskell can represent non-determinism (e.g., via list monads), this approach differs from ours in that it requires explicit lifting of operations into the monadic context, whereas our system naturally extends the existing typing framework of Curry where non-determinism is implicit in the standard type of a function.

More recent work on effect systems, such as Leijen's Koka language with row polymorphic effect types [31], provides a framework for tracking diverse computational effects. Koka can express determinism as one of many possible effects but lacks specialized support for the specific semantics of functional logic programming where non-determinism is a first-class feature rather than an optional effect.

While effect systems track side effects such as state or exceptions, determinism typing focuses specifically on the presence or absence of non-deterministic choices. This makes it a lightweight and focused tool for reasoning about program behavior in languages where non-determinism is a core feature.

11 Conclusion

In this paper we presented a type-based approach for analyzing and controlling the determinism behavior of Curry programs. Determinism types can be considered as non-standard types which are attached to defined operations. With determinism typing, we gain a compositional, static method to ensure the safe execution of I/O operations so that run-time errors often present in Curry programs can be caught at compile time.

Our system is lightweight, formally grounded, and compatible with existing Curry implementations so that it does not restrict the set of admissible programs. It improves program reliability by catching an important class of run-time errors, supports optimization, and enhances the developer experience in both interactive and compiled environments.

For future work, it might be interesting to extend determinism types to approximate other behaviors, like partially defined operations. Although there exist methods to infer sufficient conditions so that operations do not fail in Curry [22, 23], explicit annotations might allow the programmer to express intended partiality so that the compiler can suppress standard warnings in this case, similarly to the non-determinism warnings discussed in Section 7. We could even consider using intervals of determinism types to express the number of results computed by an operation.

Another direction to be explored is to capture more nuanced non-deterministic behavior. For instance, we consider the expression `(failed ? True)` as non-deterministic, although it has only one result. There are also situations where more than one result is computed but all of them are identical, as considered in [11]. A challenge of these extensions is to find techniques which ensure the correctness of such a different view of determinism.

References

[1] Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and Germán Vidal. 2005. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation* 40, 1 (2005), 795–829. doi:10.1016/j.jsc.2004.01.001

[2] Abdulla Alqaddoumi, Sergio Antoy, Sebastian Fischer, and Fabian Reck. 2010. The Pull-Tab Transformation. In *Proc. of the Third International Workshop on Graph Computation Models*. Enschede, The Netherlands, 127–132. Available at http://gcm2010.imag.fr/pages/gcm2010-preproceedings.pdf.

[3] Sergio Antoy. 2001. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*. ACM Press, 199–206. doi:10.1145/773184.773205

[4] Sergio Antoy. 2011. On the Correctness of Pull-Tabbing. *Theory and Practice of Logic Programming* 11, 4-5 (2011), 713–730. doi:10.1017/S1471068411000263

[5] Sergio Antoy and Bernd Braßel. 2007. Computing with Subspaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*. ACM Press, 121–130.

[6] Sergio Antoy, Rachid Echahed, and Michael Hanus. 2000. A Needed Narrowing Strategy. *J. ACM* 47, 4 (2000), 776–822. doi:10.1145/347476.347484

[7] Sergio Antoy and Michael Hanus. 2005. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*. Springer LNCS 3901, 6–22. doi:10.1007/11680093_2

[8] Sergio Antoy and Michael Hanus. 2006. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*. Springer LNCS 4079, 87–101. doi:10.1007/11799573_9

[9] Sergio Antoy and Michael Hanus. 2009. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*. ACM Press, 73–82. doi:10.1145/1599410.1599420

[10] Sergio Antoy and Michael Hanus. 2010. Functional Logic Programming. *Commun. ACM* 53, 4 (2010), 74–85. doi:10.1145/1721654.1721675

[11] Sergio Antoy and Michael Hanus. 2017. Eliminating Irrelevant Non-determinism in Functional Logic Programs. In *Proc. of the 19th International Symposium on*

Practical Aspects of Declarative Languages (PADL 2017). Springer LNCS 10137, 1–18. doi:10.1007/978-3-319-51676-9_1

[12] Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Olin Shivers, Guy L. Steele, and Tim Sweeney. 2023. The Verse Calculus: A Core Calculus for Deterministic Functional Logic Programming. In *Proc. ACM International Conference on Functional Programming (ICFP 2023)*. 203:1–203:31. doi:10.1145/3607845

[13] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. doi:10.1007/978-3-662-07964-5

[14] Jonas Böhm, Michael Hanus, and Finn Teegen. 2021. From Non-determinism to Goroutines: A Fair Implementation of Curry in Go. In *Proc. of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021)*. ACM Press, 16:1–16:15. doi:10.1145/3479394.3479411

[15] Bernd Braßel, Michael Hanus, and Frank Huch. 2004. Encapsulating Non-Determinism in Functional Logic Computations. *Journal of Functional and Logic Programming* 2004, 6 (2004).

[16] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. 2011. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*. Springer LNCS 6816, 1–18. doi:10.1007/978-3-642-22531-4_1

[17] Maurice Bruynooghe, Gerda Janssens, Alain Callebaut, and Bart Demoen. 1987. Abstract interpretation: towards the global optimization of Prolog programs. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*. San Francisco, 192–204.

[18] Patrick Cousot and Rhadia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*. 238–252. doi:10.1145/512950.512973

[19] Javier de Dios Castro and Francisco J. López-Fraguas. 2007. Extra variables can be eliminated from functional logic programs. *Electronic Notes in Theoretical Computer Science* 188 (2007), 3–19. doi:10.1016/j.entcs.2006.05.049

[20] Saumya K. Debray and David S. Warren. 1989. Functional Computations in Logic Programs. *ACM Trans. Program. Lang. Syst.* 11, 3 (1989), 451–481. doi:10.1145/65979.65984

[21] Michael Hanus. 2013. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*. Springer LNCS 7797, 123–168. doi:10.1007/978-3-642-37651-1_6

[22] Michael Hanus. 2024. Hybrid Verification of Declarative Programs with Arithmetic Non-fail Conditions. In *Proc. of the 22nd Asian Symposium on Programming Languages and Systems (APLAS 2024)*. Springer LNCS 15194, 109–129. doi:10.1007/978-981-97-8943-6_6

[23] Michael Hanus. 2024. Inferring Non-Failure Conditions for Declarative Programs. In *Proc. of the 17th International Symposium on Functional and Logic Programming (FLOPS 2024)*. Springer LNCS 14659, 167–187. doi:10.1007/978-981-97-2300-3_10

[24] Michael Hanus, Kai-Oliver Prott, and Finn Teegen. 2022. A Monadic Implementation of Functional Logic Programs. In *Proc. of the 24th International Symposium on Principles and Practice of Declarative Programming (PPDP 2022)*. ACM Press, 1:1–1:15. doi:10.1145/3551357.3551370

[25] Michael Hanus and Fabian Skrlac. 2014. A Modular and Generic Analysis Server System for Functional Logic Programs. In *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*. ACM Press, 181–188. doi:10.1145/2543728.2543744

[26] Michael Hanus and Finn Teegen. 2020. Adding `Data` to Curry. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*. Springer LNCS 12057, 230–246. doi:10.1007/978-3-030-46714-2_15

[27] Michael Hanus (ed.). 2016. Curry: An Integrated Functional Logic Language (Vers. 0.9.0). Available at http://www.curry-lang.org.

[28] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. 1996. Determinism analysis in the Mercury compiler. In *Proc. of the Nineteenth Australian Computer Science Conference*. 337–346.

[29] Gérard P. Huet and Jean-Jaques Lévy. 1991. Computations in Orthogonal Rewriting Systems. In *Computational Logic: Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin (Eds.). MIT Press, 395–443.

[30] Heinrich Hussmann. 1992. Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. *Journal of Logic Programming* 12 (1992), 237–255. doi:10.1016/0743-1066(92)90026-Y

[31] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Blain Levy and Neel Krishnaswami (Eds.). 100–126. doi:10.4204/EPTCS.153.8

[32] John M. Lucassen and David K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. doi:10.1145/73560.73564

[33] Connor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. doi:10.1017/S0956796807006326

[34] John Peterson and Mark P. Jones. 1993. Implementing Type Classes. In *Proc. of ACM SIGPLAN SYmposium on Programming Language Design and Implementation (PLDI'93)*. ACM SIGPLAN Notices Vol. 28, No. 6, 227–236. doi:10.1145/155090.155112

[35] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India and New York, NY, USA) *(Popl '15)*. Association for Computing Machinery, 167–180. doi:10.1145/2676726.2676971

[36] Uday S. Reddy. 1985. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*. Boston, 138–151.

[37] John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*. ACM Press, 717–740. doi:10.1145/800194.805852

[38] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. 2008. Towards Typed Prolog. In *Proceedings of the 24th International Conference on Logic Programming* (Berlin, Heidelberg) *(ICLP '08)*. Springer-Verlag, 693–697. doi:10.1007/978-3-540-89982-2_59

[39] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop* (Portland, Oregon, USA). 81–92. http://scheme2006.cs.uchicago.edu/13-siek.pdf

[40] James R. Slagle. 1974. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *J. ACM* 21, 4 (1974), 622–642. doi:10.1145/321850.321859

[41] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *The Journal of Logic Programming* 29, 1-3 (1996), 17–64. doi:10.1016/S0743-1066(96)00068-4

[42] Friedrich Steimann. 2023. A Simply Numbered Lambda Calculus. In *Eelco Visser Commemorative Symposium, EVCS 2023 (OASIcs, Vol. 109)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:12. doi:10.4230/OASICS.EVCS.2023.24

[43] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 395–406. doi:10.1145/1328438.1328486

[44] Philip Wadler. 1992. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/143165.143169

[45] Philip Wadler. 1997. How to Declare an Imperative. *Comput. Surveys* 29, 3 (1997), 240–263. doi:10.1145/262009.262011

[46] Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proc. of the 16th ACM Symposium on Principles of Programming Languages (POPL'89)*. 60–76. doi:10.1145/75277.75283

Recovering Commutation of Logically Constrained Rewriting and Equivalence Transformations

Kanta Takahata
Niigata University
Niigata, Japan

Jonas Schöpf
University of Innsbruck
Innsbruck, Austria
jonas.schoepf@uibk.ac.at

Naoki Nishida
Nagoya University
Nagoya, Japan
nishida@i.nagoya-u.ac.jp

Takahito Aoto
Niigata University
Niigata, Japan
aoto@ie.niigata-u.ac.jp

Abstract

Logically constrained term rewriting is a relatively new rewriting formalism that naturally supports built-in data structures, such as integers and bit vectors. In the analysis of logically constrained term rewrite systems (LCTRSs), rewriting constrained terms plays a crucial role. However, this combines rewrite rule applications and equivalence transformations in a closely intertwined way. This intertwining makes it difficult to establish useful theoretical properties for this kind of rewriting and causes problems in implementations—namely, that impractically large search spaces are often required. To address this issue, we propose in this paper a novel notion of most general constrained rewriting, which operates on existentially constrained terms, a concept recently introduced by the authors. We define a class of left-linear, left-value-free LCTRSs that are general enough to simulate all left-linear LCTRSs and exhibit the desired key property: most general constrained rewriting commutes with equivalence. This property ensures that equivalence transformations can be deferred until after the application of rewrite rules, which helps mitigate the issue of large search spaces in implementations. In addition to that, we show that the original rewriting formalism on constrained terms can be embedded into our new rewriting formalism on existentially constrained terms. Thus, our results are expected to have significant implications for achieving correct and efficient implementations in tools operating on LCTRSs.

CCS Concepts

• **Theory of computation → Equational logic and rewriting**.

Keywords

Logically Constrained Term Rewrite System, Commutation, Equivalence Transformation, Constrained Term

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Kanta Takahata, Jonas Schöpf, Naoki Nishida, and Takahito Aoto. 2025. Recovering Commutation of Logically Constrained Rewriting and Equivalence Transformations . In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The basic formalism of term rewriting is a purely syntactic computational model; due to its simplicity, it is one of the most extensively studied computational models. However, precisely because of this simplicity, it is often not suitable for applications that arise in practical areas, such as in programming languages, formal specifications, etc. One of the main issues of term rewriting and its real-world applications is that the basic formalism lacks painless treatment of built-in data structures, such as integers, bit vectors, etc.

Logically constrained term rewriting [7] is a relatively new extension of term rewriting that intends to overcome such weaknesses of the basic formalism, while keeping succinctness for theoretical analysis as a computational model. Rewrite rules, that are used to model computations, in logically constrained term rewrite systems (LCTRSs) are equipped with constraints over some arbitrary theory, e.g., linear integer arithmetic. Built-in data structures are represented via the satisfiability of constraints within a respective theory. Implementations in LCTRS tools are then able to check these constraints using SMT-solvers and therefore benefit from recent advances in the area of checking *satisfiability modulo theories*. Recent progress on the LCTRS formalism was for example made in confluence analysis [13, 15], (non-)termination analysis [5, 10], completion [18], rewriting induction [3, 8], algebraic semantics [1], and complexity analysis [19].

During the analysis of LCTRSs, not only rewriting of terms but also rewriting of constrained terms, called *constrained rewriting*, is frequently used. Here, a *constrained term* consists of a term and a constraint, which restricts the possibilities in which the term is instantiated. For example, $\mathsf{f}(x)\ [x > 2]$ is a constrained term (in LCTRS notation) which can be intuitively considered as a set of terms $\{\mathsf{f}(x) \mid x > 2\}$. Constrained rewriting is an integral part of many different analysis techniques. For example, in finding a specific joining sequence in confluence analysis you need to deal with two terms under a shared constraint, which results in

working on constrained terms. In rewriting induction, rewriting of constrained terms is used for several of its inference steps.

Unfortunately, supporting constrained rewriting completely for LCTRSs is far from practical, as it involves heavy non-determinism. The situation gets even worse as it is also equipped with equivalence transformations before (and/or after) each rewrite step.

Example 1.1. Consider the rewrite rule $\rho\colon \mathsf{f}(x) \to \mathsf{g}(y)\ [x \geqslant 1 \wedge x + 1 \geqslant y]$ and a constrained term $\mathsf{f}(x)\ [x > 2]$. Because, for any $x > 2$, the instantiation $y := 3$ satisfies the constraint $x \geqslant 1 \wedge x + 1 \geqslant y$ of the rewrite rule, we obtain a rewrite step

$$\mathsf{f}(x)\ [x > 2] \to_\rho \mathsf{g}(3)\ [x > 2].$$

It is also possible to apply the following rewrite step, because the instantiation $y := x$ satisfies the constraint:

$$\mathsf{f}(x)\ [x > 2] \to_\rho \mathsf{g}(x)\ [x > 2]$$

Actually, by using equivalence transformations (denoted by $\sim$), different variations of rewrite steps are possible, e.g.:

$$\begin{aligned} \mathsf{f}(x)\ [x > 2] &\sim \mathsf{f}(x)\ [x > 2 \wedge 0 > y] \\ &\to_\rho \mathsf{g}(y)\ [x > 2 \wedge 0 > y] \\ &\sim \mathsf{g}(y)\ [0 > y] \end{aligned}$$

Note that the resulting constrained terms are often not equivalent, e.g. $\mathsf{g}(3)\ [x > 2] \not\sim \mathsf{g}(x)\ [x > 2]$, as $\{\mathsf{g}(3)\} \neq \{\mathsf{g}(x) \mid x > 2\}$.

The question may arise whether restricting to rewriting without equivalence transformations is a good idea. However, it turns out that for some natural computations, we need an equivalence transformation prior to the actual rewrite step:

Example 1.2. Consider the rewrite rule $\rho\colon \mathsf{h}(x, y) \to \mathsf{g}(z)\ [(x + y) + 1 = z]$ and a constrained term $\mathsf{h}(x, y)\ [x < y]$. It is not possible to take any concrete value or variable of x, y for z, and hence the constrained term using the rule ρ cannot be rewritten. However, after the equivalence transformation, the rule becomes applicable:

$$\begin{aligned} \mathsf{h}(x, y)\ [x < y] &\sim \mathsf{h}(x, y)\ [x < y \wedge x + y + 1 = z] \\ &\to_\rho \mathsf{g}(z)\ [x < y \wedge x + y + 1 = z] \end{aligned}$$

Clearly, it is not feasible to support the full strength of such an equivalence relation in an implementation.

In this paper, we introduce a novel notion of most general constrained rewriting, which operates on existentially constrained terms, a concept recently introduced by the authors [16]. As seen in the example above, a key source of confusion is that the rewrite step heavily relies on variables in the constraint that do not appear in the term itself. The existentially constrained terms distinguish variables that appear solely in the constraint but not in the term itself by using existential quantifiers. Variables appearing only within the constraint are naturally not allowed to appear in any reduction of a term as they are bound to the scope of the constraint. It turns out that this novel way of rewriting covers the "most general part" of the original rewrite relation, which in practice usually suffices for the analysis of LCTRSs.

Additionally, it fulfills not only a form of uniqueness of reducts but also the commutation property of rewrite steps and equivalence transformations. These features are not supported by the original rewrite relation. The latter property about commutation is very important from an implementation perspective, because as a result one can move intermediate equivalence transformations in rewrite sequences to the end of the sequence. This property reduces huge search spaces for the computations of rewrite sequences. Coincidentally, LCTRS tools such as Ctrl [9], Crisys [4], and crest [14, 15] already implement similar approaches to deal with constrained rewriting. However, this has not formally been defined so far. Our results guarantee the correctness of the approaches in these implementations and provide the foundation for their correctness.

The remainder of the paper is organized as follows. After presenting the necessary background in Section 2, we introduce most general rewrite steps and prove its well-definedness in Section 3. In Sections 4 and 5 we focus on the relation between our new formalism of constrained rewriting and the current state-of-the-art. Then in Section 6, we show two useful properties of most general rewriting: uniqueness of reducts and commutation between rewrite steps and equivalence transformations on pattern-general existentially constrained terms. Subsequently we introduce left-value-free rules [6] within our setting and discuss their rewriting behavior in Section 7. In Section 8 we show a general commutation theorem between rewrite steps using left-value-free rules and the equivalence transformation. Before we conclude in Section 10, we discuss some related work in Section 9.

2 Preliminaries

In this section, we briefly recall the basic notions of LCTRSs [1, 7, 13, 15] and fix additional notations used throughout this paper. Familiarity with the basic notions of term rewriting is assumed (e.g. see [2, 11]).

Logically Constrained Terms. Our signature consists of a set $\mathcal{S}$ of sorts and a set $\mathcal{F}$ of function symbols, where each $f \in \mathcal{F}$ is attached with its sort declaration $f\colon \tau_1 \times \cdots \times \tau_n \to \tau_0$. As in [1], we assume that these sets can be partitioned into two disjoint sets, i.e., $\mathcal{S} = \mathcal{S}_{\mathsf{th}} \uplus \mathcal{S}_{\mathsf{te}}$ and $\mathcal{F} = \mathcal{F}_{\mathsf{th}} \uplus \mathcal{F}_{\mathsf{te}}$, where each $f\colon \tau_1 \times \cdots \times \tau_n \to \tau_0 \in \mathcal{F}_{\mathsf{th}}$ satisfies $\tau_i \in \mathcal{S}_{\mathsf{th}}$ for all $0 \leqslant i \leqslant n$. Elements of $\mathcal{S}_{\mathsf{th}}$ ($\mathcal{F}_{\mathsf{th}}$) and $\mathcal{S}_{\mathsf{te}}$ ($\mathcal{F}_{\mathsf{te}}$) are called theory sorts (symbols) and term sorts (symbols). The sets of variables and terms are denoted by $\mathcal{V}$ and $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We assume a special sort $\mathsf{Bool} \in \mathcal{S}_{\mathsf{th}}$, and call the terms in $\mathcal{T}(\mathcal{F}_{\mathsf{th}}, \mathcal{V})^{\mathsf{Bool}}$ *logical constraints*. We denote the set of variables appearing in terms $t_1, \ldots, t_n$ by $\mathcal{V}(t_1, \ldots, t_n)$. Sometimes sequences of variables and terms are written as $\vec{x}$ and $\vec{t}$. The set of variables occurring in $\vec{x}$ is denoted by $\{\vec{x}\}$. The set of sequences of elements of a set T is denoted by T^*, such that $\vec{x} \in \mathcal{V}^*$.

The set of positions in a term t is denoted by $\mathcal{P}\mathrm{os}(t)$. The symbol and subterm occurring at a position $p \in \mathcal{P}\mathrm{os}(t)$ is denoted by $t(p)$ and $t|_p$, respectively. For $U \subseteq \mathcal{F} \cup \mathcal{V}$, we write $\mathcal{P}\mathrm{os}_U(t) = \{p \in \mathcal{P}\mathrm{os}(t) \mid t(p) \in U\}$ for positions with symbols in U. A term obtained from t by replacing subterms at positions $p_1, \ldots, p_n$ by the terms $t_1, \ldots, t_n$, having the same sort as $t|_{p_1}, \ldots, t|_{p_n}$, is written as $t[t_1, \ldots, t_n]_{p_1,\ldots,p_n}$ or just $t[t_1, \ldots, t_n]$ when no confusions arises. Sometimes we consider an expression obtained by replacing those subterms $t|_{p_1}, \ldots, t|_{p_n}$ in t by holes of the same sorts, which is called a multihole context and denoted by $t[\ ]_{p_1,\ldots,p_n}$.

A sort-preserving function σ from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is called a substitution, where it is identified with its homomorphic extension $\sigma\colon \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}, \mathcal{V})$. The domain of a substitution σ is given by $\mathcal{D}\mathrm{om}(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$. A substitution σ is written

as $\sigma\colon U \to T$ if $\mathcal{D}\mathrm{om}(\sigma) \subseteq U$ and $\sigma(U) = \{\sigma(x) \mid x \in U\} \subseteq T$. For a set $U \subseteq \mathcal{V}$, a substitution $\sigma|_U$ is given by $\sigma|_U(x) = \sigma(x)$ if $x \in U$ and $\sigma|_U(x) = x$ otherwise. For substitutions σ_1, σ_2 such that $\mathcal{D}\mathrm{om}(\sigma_1) \cap \mathcal{D}\mathrm{om}(\sigma_2) = \emptyset$, the substitution $\sigma_1 \cup \sigma_2$ is given by $(\sigma_1 \cup \sigma_2)(x) = \sigma_i(x)$ if $x \in \mathcal{D}\mathrm{om}(\sigma_i)$ and $(\sigma_1 \cup \sigma_2)(x) = x$ otherwise. A substitution $\sigma\colon \{x_1, \ldots, x_n\} \to \{t_1, \ldots, t_n\}$ such that $\sigma(x_i) = t_i$ is denoted by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$; for brevity sometimes we write just $\{\vec{x} \mapsto \vec{t}\}$. A bijective substitution $\sigma\colon \mathcal{V} \to \mathcal{V}$ is called a renaming, and its inverse is denoted by σ^{-1}.

A model over a sorted signature $\langle \mathcal{S}_{\mathsf{th}}, \mathcal{F}_{\mathsf{th}} \rangle$ consists of the two interpretations $\mathcal{I}$ for sorts and $\mathcal{J}$ for function symbols such that $\mathcal{I}(\tau)$ is a non-empty set of values and $\mathcal{J}$ assigns any function symbol $f\colon \tau_1 \times \cdots \times \tau_n \to \tau_0 \in \mathcal{F}_{\mathsf{th}}$ to a function $\mathcal{J}(f)\colon \mathcal{I}(\tau_1) \times \cdots \times \mathcal{I}(\tau_n) \to \mathcal{I}(\tau_0)$. We assume a fixed model $\mathcal{M} = \langle \mathcal{I}, \mathcal{J} \rangle$ over the sorted signature $\langle \mathcal{S}_{\mathsf{th}}, \mathcal{F}_{\mathsf{th}} \rangle$ such that any element $a \in \mathcal{I}(\tau)$ appears as a constant a^τ in $\mathcal{F}_{\mathsf{th}}$. These constants are called *values* and the set of all values is denoted by $\mathcal{V}\mathsf{al}$. For a term t, we define $\mathcal{V}\mathsf{al}(t) = \{t(p) \mid p \in \mathcal{P}\mathrm{os}_{\mathcal{V}\mathsf{al}}(t)\}$ and for a substitution γ, we define $\mathcal{V}\mathcal{D}\mathrm{om}(\gamma) = \{x \in \mathcal{V} \mid \gamma(x) \in \mathcal{V}\mathsf{al}\}$. Throughout this paper we assume the standard interpretation for the sort $\mathsf{Bool} \in \mathcal{S}_{\mathsf{th}}$, namely $\mathcal{I}(\mathsf{Bool}) = \mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$, and the existence of necessary standard theory function symbols such as $\neg, \wedge, \Rightarrow, =^\tau$, etc.

A valuation ρ on the model $\mathcal{M} = \langle \mathcal{I}, \mathcal{J} \rangle$ is a mapping that assigns any $x^\tau \in \mathcal{V}$ with $\rho(x) \in \mathcal{I}(\tau)$. The interpretation of a term t in the model $\mathcal{M}$ over the valuation ρ is denoted by $[\![t]\!]_{\mathcal{M},\rho}$. For a logical constraint φ and a valuation ρ over the model $\mathcal{M}$, we write $\vDash_{\mathcal{M},\rho} \varphi$ if $[\![\varphi]\!]_{\mathcal{M},\rho} = \mathsf{true}$, and $\vDash_{\mathcal{M}} \varphi$ if $\vDash_{\mathcal{M},\rho} \varphi$ for all valuations ρ. For $X \subseteq \mathcal{V}$, a substitution γ is said to be X-valued if $\gamma(X) \subseteq \mathcal{V}\mathsf{al}$. We write $\gamma \vDash_{\mathcal{M}} \varphi$ (and say γ respects φ) if the substitution γ is $\mathcal{V}\mathrm{ar}(\varphi)$-valued (or equivalently, $\mathcal{V}\mathrm{ar}(\varphi) \subseteq \mathcal{V}\mathcal{D}\mathrm{om}(\gamma)$, as well as $\gamma(\mathcal{V}\mathrm{ar}(\varphi)) \subseteq \mathcal{V}\mathsf{al}$) and $\vDash_{\mathcal{M}} \varphi\gamma$. If no confusion arises then we drop the subscript $\mathcal{M}$ in these notations.

Rewriting Logically Constrained Terms. The main focus in this paper lies on rewriting constrained terms, hence we focus solely on introducing this concept. A *constrained term* is a tuple of a term s and a logical constraint φ written as $s\ [\varphi]$. Two constrained terms $s\ [\varphi]$, $t\ [\psi]$ are equivalent, written as $s\ [\varphi] \sim t\ [\psi]$, if for all substitutions σ with $\sigma \vDash \varphi$ there exists a substitution γ with $\gamma \vDash \psi$ such that $s\sigma = t\gamma$, and vice versa.

A constrained rewrite rule is a triple $\ell \to r\ [\varphi]$ of terms ℓ, r with the same sort, and a logical constraint φ. The set of *logical variables* of the rule is given by $\mathcal{LV}\mathrm{ar}(\ell \to r\ [\varphi]) = (\mathcal{V}\mathrm{ar}(r) \setminus \mathcal{V}\mathrm{ar}(\ell)) \cup \mathcal{V}\mathrm{ar}(\varphi)$. A *logically constrained term rewrite system* (LCTRS, for short) consists of a signature $\Sigma = \langle \mathcal{S}_{\mathsf{th}} \uplus \mathcal{S}_{\mathsf{te}}, \mathcal{F}_{\mathsf{th}} \uplus \mathcal{F}_{\mathsf{te}} \rangle$, a model $\mathcal{M}$ over $\Sigma_{\mathsf{th}} = \langle \mathcal{S}_{\mathsf{th}}, \mathcal{F}_{\mathsf{th}} \rangle$ (which induces the set $\mathcal{V}\mathsf{al} \subseteq \mathcal{F}_{\mathsf{th}}$ of values) and a set $\mathcal{R}$ of constrained rewrite rules over the signature Σ.

Consider a constrained term $s\ [\varphi]$ with a satisfiable logical constraint. Let $\ell \to r\ [\psi]$ be a constrained rewrite rule. If there exists a position $p \in \mathcal{P}\mathrm{os}_{\mathcal{F}}(s)$, a substitution σ with $\ell\sigma = s|_p$, $\sigma(x) \in \mathcal{V}\mathsf{al} \cup \mathcal{V}\mathrm{ar}(\varphi)$ for all $x \in \mathcal{LV}\mathrm{ar}(\ell \to r\ [\psi])$, and $\vDash_{\mathcal{M}} \varphi \Rightarrow \psi\sigma$, then we have the rewrite step $s\ [\varphi] \to s[r\sigma]_p\ [\varphi]$. The full rewrite relation $\xrightarrow{}$ on constrained terms is defined by $\xrightarrow{} = \sim \cdot \to \cdot \sim$. In [15], it is shown that *calculation steps* defined in [7] can be integrated into this rewrite relation by using the constrained rules $\mathcal{R}_{\mathsf{ca}} = \{f(x_1, \ldots, x_n) \to y\ [y = f(x_1, \ldots, x_n)] \mid f \in \mathcal{F}_{\mathsf{th}} \setminus \mathcal{V}\mathsf{al}\}$, including some additional initial $\sim$-steps.

Existentially Constrained Terms and Their Equivalence. We now explain existentially constrained terms and characterization of their equivalence based on [16].

An *existential constraint* is a pair $\langle \vec{x}, \varphi \rangle$ of a sequence of variables $\vec{x}$ and a constraint φ, written as $\exists\vec{x}.\ \varphi$, such that $\{\vec{x}\} \subseteq \mathcal{V}\mathrm{ar}(\varphi)$. The sets of *free* and *bound* variables of $\exists\vec{x}.\ \varphi$ are given by $\mathcal{FV}\mathrm{ar}(\exists\vec{x}.\ \varphi) = \mathcal{V}\mathrm{ar}(\varphi) \setminus \{\vec{x}\}$ and $\mathcal{BV}\mathrm{ar}(\exists\vec{x}.\ \varphi) = \{\vec{x}\}$. We write $\vDash_{\mathcal{M},\rho} \exists\vec{x}.\ \varphi$ if there exists $\vec{v} \in \mathcal{V}\mathsf{al}^*$ such that $\vDash_{\mathcal{M},\rho} \varphi\kappa$, where $\kappa = \{\vec{x} \mapsto \vec{v}\}$. An existential constraint $\exists\vec{x}.\ \varphi$ is said to be *valid*, written as $\vDash_{\mathcal{M}} \exists\vec{x}.\ \varphi$, if $\vDash_{\mathcal{M},\rho} \exists\vec{x}.\ \varphi$ for any valuation ρ. An existential constraint $\exists\vec{x}.\ \varphi$ is said to be *satisfiable* if $\vDash_{\mathcal{M},\rho} \exists\vec{x}.\ \varphi$ for some valuation ρ. For any substitution σ, we write $\sigma \vDash_{\mathcal{M}} \exists\vec{x}.\ \varphi$ (and say σ respects $\exists\vec{x}.\ \varphi$) if $\sigma(\mathcal{FV}\mathrm{ar}(\exists\vec{x}.\ \varphi)) \subseteq \mathcal{V}\mathsf{al}$ and $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi)\sigma$. An *existentially constrained term* is a triple $\langle X, s, \exists\vec{x}.\ \varphi \rangle$, written as $\Pi X.\ s\ [\exists\vec{x}.\ \varphi]$, of a set X of variables, a term s, and an existential constraint $\exists\vec{x}.\ \varphi$ such that $\mathcal{FV}\mathrm{ar}(\exists\vec{x}.\ \varphi) \subseteq X \subseteq \mathcal{V}\mathrm{ar}(s)$ and $\mathcal{BV}\mathrm{ar}(\exists\vec{x}.\ \varphi) \cap \mathcal{V}\mathrm{ar}(s) = \emptyset$. Variables in X are called *logical variables* (of $\Pi X.\ s\ [\exists\vec{x}.\ \varphi]$). An existentially constrained term $\Pi X.\ s\ [\exists\vec{x}.\ \varphi]$ is said to be *satisfiable* if $\exists\vec{x}.\ \varphi$ is satisfiable. An existentially constrained term $\Pi X.\ s\ [\exists\vec{x}.\ \varphi]$ is said to be *subsumed by* an existentially constrained term $\Pi Y.\ t\ [\exists\vec{y}.\ \psi]$, denoted by $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \lesssim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$, if for all X-valued substitutions σ with $\sigma \vDash_{\mathcal{M}} \exists\vec{x}.\ \varphi$ there exists a Y-valued substitution γ with $\gamma \vDash_{\mathcal{M}} \exists\vec{y}.\ \psi$ such that $s\sigma = t\gamma$. Existentially constrained terms $\Pi X.\ s\ [\exists\vec{x}.\ \varphi], \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ are said to be *equivalent*, denoted by $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$, if $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \lesssim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ and $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \gtrsim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$. Throughout this paper, we will use three characterizations of equivalence for existentially constrained terms, presented in [16]. The first one covers variants under renaming.

Proposition 2.1 ([16]). *Let δ be a renaming. Let $\Pi X.\ s\ [\exists\vec{x}.\ \varphi]$, $\Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ be satisfiable existentially constrained terms such that $s\delta = t$. Then, $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ iff $\delta(X) = Y$ and $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi)\delta \Leftrightarrow (\exists\vec{y}.\ \psi)$.*

The second one considers those that are *pattern-general*, where an existentially constrained term $\Pi X.\ s\ [\exists\vec{x}.\ \varphi]$ is *pattern-general* if s is X-linear and $\mathcal{V}\mathsf{al}(s) = \emptyset$ [16]. Any existentially constrained term can be translated to equivalent pattern-general existentially constrained term by the following translation: $\mathrm{PG}(\Pi X.\ s\ [\exists\vec{x}.\ \varphi]) = \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ where $t = s[w_1, \ldots, w_n]_{p_1,\ldots,p_n}$ with $\mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathsf{al}}(s) = \{p_1, \ldots, p_n\}$ and pairwise distinct fresh variables $w_1, \ldots, w_n$, $Y = \{w_1, \ldots, w_n\}$, $\{\vec{y}\} = \{\vec{x}\} \cup X$, and $\psi = (\varphi \wedge \bigwedge_{i=1}^{n}(s|_{p_i} = w_i))$.

Proposition 2.2 ([16]). *Let $\Pi X.\ s\ [\exists\vec{x}.\ \varphi], \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ be satisfiable pattern-general existentially constrained terms. Then, we have $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ iff there exists a renaming ρ such that $s\rho = t$, $\rho(X) = Y$, and $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi)\rho \Leftrightarrow (\exists\vec{y}.\ \psi)$.*

The last characterization does not impose any restriction on the existentially constrained terms. In order to present this, we need a couple of notions and results from [16]. Let $\Pi X.\ s\ [\exists\vec{x}.\ \varphi]$ be an existentially constrained term. Then, we define a binary relation $\sim_{\mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathsf{al}}(s)}$ over the positions in $\mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathsf{al}}(s)$ as follows: $p \sim_{\mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathsf{al}}(s)} q$ iff $\vDash_{\mathcal{M}} ((\exists\vec{x}.\ \varphi) \Rightarrow s|_p = s|_q)$. Then, $\sim_{\mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathsf{al}}(s)}$ is an equivalence relation over the positions in $\mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathsf{al}}(s)$. The equivalence class of a position $p \in \mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathsf{al}}(s)$ w.r.t. $\sim_{\mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathsf{al}}(s)}$ is denoted by $[p]_{\sim_{\mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathsf{al}}(s)}}$. If it is clear from the context then we

may simply denote it by $[p]_\sim$. We further denote the representative of $[p]_\sim$ by $\hat{p}$. Let $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,]$ be a satisfiable existentially constrained term. We define $\mathcal{P}\mathrm{os}_{\mathcal{V}\mathrm{al}!}(s) = \{p \in \mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathrm{al}}(s) \mid$ there exists $v \in \mathcal{V}\mathrm{al}$ such that $\vDash_{\mathcal{M}} ((\exists\vec{x}.\ \varphi) \Rightarrow (s|_p = v))\}$. For each $p \in \mathcal{P}\mathrm{os}_{\mathcal{V}\mathrm{al}!}(s)$, there exists a unique value v such that $\mathcal{M} \vDash (\exists\vec{x}.\ \varphi) \Rightarrow (s|_p = v)$ [16]; we denote such a v by $\mathcal{V}\mathrm{al}!(p)$. We then define a *representative substitution* $\mu_X : X \to X \cup \mathcal{V}\mathrm{al}$ of $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,]$ as follows:

$$\mu_X(z) = \begin{cases} \mathcal{V}\mathrm{al}!(p) & \text{if } s(p) = z \text{ for some } p \in \mathcal{P}\mathrm{os}_{\mathcal{V}\mathrm{al}!}(s), \\ s(\hat{p}) & \text{otherwise,} \end{cases}$$

where $\hat{p}$ is the representative of the equivalence class $[p]_\sim$.

Proposition 2.3 ([16]). *Let $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,], \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ be satisfiable existentially constrained terms. Then, $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] \sim \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ iff the following statements hold:*

1. *$\mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathrm{al}}(s) = \mathcal{P}\mathrm{os}_{Y\cup\mathcal{V}\mathrm{al}}(t)$ $(= \{p_1, \ldots, p_n\})$,*
2. *there exists a renaming $\rho : \mathcal{V}\mathrm{ar}(s) \setminus X \to \mathcal{V}\mathrm{ar}(t) \setminus Y$ such that $\rho(s[\,]_{p_1,\ldots,p_n}) = t[\,]_{p_1,\ldots,p_n}$,*
3. *for any $i, j \in \{1, \ldots, n\}$, $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (s|_{p_i} = s|_{p_j})$ iff $\vDash_{\mathcal{M}} (\exists\vec{y}.\ \psi) \Rightarrow (t|_{p_i} = t|_{p_j})$,*
4. *for any $i \in \{1, \ldots, n\}$ and $v \in \mathcal{V}\mathrm{al}$, $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (s|_{p_i} = v)$ iff $\vDash_{\mathcal{M}} (\exists\vec{y}.\ \psi) \Rightarrow (t|_{p_i} = v)$, and*
5. *let $\sim = \sim_{\mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathrm{al}}(s)} = \sim_{\mathcal{P}\mathrm{os}_{X\cup\mathcal{V}\mathrm{al}}(t)}$, and μ_X, μ_Y be representative substitutions of $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,]$ and $\Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$, respectively, based on the same representative for each equivalence class $[p_i]_\sim$ $(1 \leqslant i \leqslant n)$, $\theta = \{\langle s|_{p_i}, t|_{p_i}\rangle \mid 1 \leqslant i \leqslant n\}$, $\tilde{X} = \hat{X} \cap X$, and $\tilde{Y} = \hat{Y} \cap Y$. Then, $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi)\mu_X\theta|_{\tilde{X}} \Leftrightarrow (\exists\vec{y}.\ \psi)\mu_Y$, with the renaming $\theta|_{\tilde{X}} : \tilde{X} \to \tilde{Y}$.*

3 Most General Constrained Rewriting

In this section, we present the definition of most general rewriting and prove its well-definedness. Furthermore, we also show some properties needed in later sections.

We start by devising the notion of constrained rewrite rules tailored to our needs. Similar to constrained equations [1], we attach the respective set of logical variables explicitly to constrained rewrite rules. Previously to this, rewriting relied heavily on the notion of logical variables ($\mathcal{LV}\mathrm{ar}(\ldots)$) for this.

Definition 3.1 (Constrained Rewrite Rule). A *constrained rewrite rule* is a quadruple of two terms ℓ, r, a set of variables Z and a logical constraint π, written as $\Pi Z.\ \ell \to r\ [\,\pi\,]$, where ℓ and r are of the same sort and Z satisfies $(\mathcal{V}\mathrm{ar}(r) \setminus \mathcal{V}\mathrm{ar}(\ell)) \cup \mathcal{V}\mathrm{ar}(\pi) \subseteq Z \subseteq \mathcal{V}\mathrm{ar}(\ell, r, \pi)$. When no confusion arises, we abbreviate constrained rewrite rules by *constrained rules* or just *rules*; sometimes we attach a label to it, as $\rho : \Pi Z.\ \ell \to r\ [\,\pi\,]$, in order to ease referencing it. Let $\rho : \Pi Z.\ \ell \to r\ [\,\pi\,]$ be a constrained rule. Then, ρ is *left-linear* if ℓ is linear; the set of *extra-variables* of ρ is given by $\mathcal{E}\mathrm{x}\mathcal{V}\mathrm{ar}(\rho) = \mathcal{V}\mathrm{ar}(r) \setminus \mathcal{V}\mathrm{ar}(\ell)$[1].

It follows naturally, that a constrained rule $\ell \to r\ [\,\varphi\,]$ as defined previously can easily be transformed into $\Pi X.\ \ell \to r\ [\,\varphi\,]$ by taking $X = \mathcal{LV}\mathrm{ar}(\ell \to r\ [\,\varphi\,])$ $(:= \mathcal{V}\mathrm{ar}(\pi) \cup (\mathcal{V}\mathrm{ar}(r) \setminus \mathcal{V}\mathrm{ar}(\ell)))$. We also do not impose the constraint $\ell|_\epsilon \in \mathcal{F}_{\mathrm{te}}$ as it is done in [7], in order to deal with calculation steps via rule steps as in [15].

[1]Do not confuse the notation $\mathcal{E}\mathrm{x}\mathcal{V}\mathrm{ar}(\rho)$ with $\mathcal{E}\mathcal{V}\mathrm{ar}(\rho) = \mathcal{V}\mathrm{ar}(r) \setminus (\mathcal{V}\mathrm{ar}(\ell) \cup \mathcal{V}\mathrm{ar}(\pi))$ from [13, 15].

Example 3.2. For example, $\Pi\{x, y\}.\ \mathrm{f}(x) \to \mathrm{g}(y)\ [\,x \geqslant 1\,]$ and $\Pi\{x\}.\ \mathrm{h}(x, y) \to \mathrm{g}(y)\ [\,x \geqslant 1\,]$ are constrained rewrite rules. However, none of these are: $\Pi\{x, y, z\}.\ \mathrm{f}(x) \to \mathrm{g}(y)\ [\,x \geqslant 1\,]$ (z does not appear in $\mathcal{V}\mathrm{ar}(\mathrm{f}(x), \mathrm{g}(y), x \geqslant 1)$) and $\Pi\{x\}.\ \mathrm{f}(x) \to \mathrm{g}(y)\ [\,x \geqslant 1\,]$ ($y \in \mathcal{V}\mathrm{ar}(r) \setminus \mathcal{V}\mathrm{ar}(\ell)$ does not appear in $\{x\}$). The rewrite rule $\rho : \mathrm{f}(x) \to \mathrm{g}(y)\ [\,x \geqslant 1 \wedge x + 1 \geqslant y\,]$ in Example 1.1 is written as $\Pi\{x, y\}.\ \mathrm{f}(x) \to \mathrm{g}(y)\ [\,x \geqslant 1 \wedge x + 1 \geqslant y\,]$.

It turns out that certain complications arise if we deal with non-left-linear rules, *hence in the following we consider only left-linear constrained rules and leave the extension to non-left-linear rules for future work.* Before presenting the definition of most general constrained rewrite steps, we introduce the notion of ρ-redex for a rule ρ.

Definition 3.3 (ρ-Redex). Let $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,]$ be a satisfiable existentially constrained term. Suppose that $\rho : \Pi Z.\ \ell \to r\ [\,\pi\,]$ is a left-linear constrained rule satisfying $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(s, \varphi) = \emptyset$. We say $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,]$ *has a ρ-redex at position $p \in \mathcal{P}\mathrm{os}(s)$ using substitution γ* if the following is satisfied: (1.) $\mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell)$, (2.) $s|_p = \ell\gamma$, (3.) $\gamma(x) \in \mathcal{V}\mathrm{al} \cup X$ for all $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$, and (4.) $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (\exists\vec{z}.\ \pi\gamma)$, where $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$.

Example 3.4. Consider a constrained rule $\rho : \Pi\{x', y'\}.\ \mathrm{f}(x') \to \mathrm{g}(y')\ [\,x' \geqslant 0 \wedge y' > x'\,]$. The constrained term $\Pi\{x\}.\ \mathrm{f}(x)\ [\,x > 0\,]$ has a ρ-redex at the root position using the substitution $\gamma = \{x' \mapsto x\}$.

Definition 3.5 (Most General Rewrite Steps for Constrained Terms). Let $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,]$ be a satisfiable existentially constrained term. Suppose that $\rho : \Pi Z.\ \ell \to r\ [\,\pi\,]$ is a left-linear constrained rule satisfying $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(s, \varphi) = \emptyset$. Suppose $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,]$ has a ρ-redex at position $p \in \mathcal{P}\mathrm{os}(s)$ using substitution γ. Then we have a rewrite step $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] \to_\rho \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ (or $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] \to^p_{\rho,\gamma} \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ with explicit p and γ) where (1.) $t = s[r\gamma]$, (2.) $\psi = \varphi \wedge \pi\gamma$, (3.) $\{\vec{y}\} = \mathcal{V}\mathrm{ar}(\psi) \setminus \mathcal{V}\mathrm{ar}(t)$, and (4.) $Y = \mathcal{E}\mathrm{x}\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t))$. We call this notion *most general rewrite steps*; the reason behind this naming will become clear in the remainder of the paper.

Example 3.6 (Cont'd of Example 3.4). Observe that $\gamma(x') = x \in \{x\}$ and $\vDash_{\mathcal{M}} x > 0 \Rightarrow (\exists y'.\ x \geqslant 0 \wedge y' > x)$. Therefore, we obtain a rewrite step

$$\begin{aligned} &\Pi\{x\}.\ \mathrm{f}(x)\ [\,x > 0\,] \\ &\to_{\rho,\gamma} \Pi\{y'\}.\ \mathrm{g}(y')\ [\,\exists x.\ x > 0 \wedge x \geqslant 0 \wedge y' > x\,] \end{aligned}$$

Note that $(x' \geqslant 0 \wedge y' > x')\gamma = (x \geqslant 0 \wedge y' > x)$, $\mathcal{V}\mathrm{ar}(x \geqslant 0 \wedge y' > x) \setminus \mathcal{V}\mathrm{ar}(\mathrm{g}(y')) = \{x\}$, and $\mathcal{E}\mathrm{x}\mathcal{V}\mathrm{ar}(\rho) \cup (\{x\} \cap \mathcal{V}\mathrm{ar}(\mathrm{g}(y'))) = \{y'\}$.

In a constrained rewrite step $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] \to \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$, variables of X will sometimes either be moved to Y or $\{\vec{y}\}$, or even be eliminated such that they do not appear in $\Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ anymore. The latter behavior may happen in rewrite rules with eliminating variables, i.e., where $\mathcal{V}\mathrm{ar}(\ell) \setminus \mathcal{V}\mathrm{ar}(r) \neq \emptyset$. The following example illustrates such situations.

Example 3.7. Let $\rho : \Pi\{v', w', x', y', z'\}.\ \mathrm{f}(v', w', x', y', z') \to v'\ [\,y' \geqslant 0\,] (= \Pi Z.\ \ell \to r\ [\,\pi\,])$ be a constrained rule. Consider

a rewrite step

$$\begin{aligned}\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] &= \Pi\{v,w,x,y,z\}.\ \mathrm{g}(\mathrm{f}(v,w,x,y,z),w)\ [\,x \geqslant 0\,]\\ &\to^{1}_{\rho,\gamma} \Pi\{v,w\}.\ \mathrm{g}(v,w)\ [\,\exists x,y.\ x > 1 \wedge y \geqslant 0\,]\\ &= \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]\end{aligned}$$

where and $\gamma = \{v' \mapsto v, \ldots, z' \mapsto z\}$. We trace now where the variables in X move during this rewrite step:

- We have $v \in Y$ because $v' \in \mathcal{V}\mathrm{ar}(r)$, $\gamma(v') = v$ and $v \in \mathcal{V}\mathrm{ar}(t)$.
- We have $w \in Y$ because $s|_2 = t|_2 = w$, hence $w \in \mathcal{V}\mathrm{ar}(t)$.
- We have $x \in \{\vec{y}\}$. Note $x \in \mathcal{V}\mathrm{ar}(s)$, but $x \notin \mathcal{V}\mathrm{ar}(r\gamma)$ and $x \notin \mathcal{V}\mathrm{ar}(t)$; however, since $x \in \mathcal{V}\mathrm{ar}(\varphi)$, we have $x \in \{\vec{y}\}$.
- We have $y \in \{\vec{y}\}$; note that $y \in \mathcal{V}\mathrm{ar}(s)$, but $y \notin \mathcal{V}\mathrm{ar}(r\gamma)$, $y \notin \mathcal{V}\mathrm{ar}(t)$ and $y \notin \mathcal{V}\mathrm{ar}(\varphi)$; however, $y \in \mathcal{V}\mathrm{ar}(\pi\gamma)$ and hence $y \in \mathcal{V}\mathrm{ar}(\psi)$.
- We have $z \notin Y \cup \{\vec{y}\}$; although $z \in \mathcal{V}\mathrm{ar}(s)$, we have that $z \notin \mathcal{V}\mathrm{ar}(t)$, $z \notin \mathcal{V}\mathrm{ar}(r)$ and $z \notin \mathcal{V}\mathrm{ar}(\psi)$.

In the following we show that rewriting existentially constrained terms is well-defined, in the sense that each rewrite step results in an existentially constrained term. Before that we give certain characterizations which are used in later proofs; proofs of these lemmas are presented in the full version of this paper [17].

Lemma 3.8 (Characterization of Free Variables in Reducts). *Let $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,]$ be a satisfiable existentially constrained term. Suppose that $\rho\colon \Pi Z.\ \ell \to r\ [\,\pi\,]$ is a left-linear constrained rule satisfying $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(s,\varphi) = \varnothing$. If $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] \to_\rho \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ then $\mathcal{FV}\mathrm{ar}(\exists\vec{y}.\ \psi) \subseteq \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t))$.*

Lemma 3.9 (Characterization of Bound Variables). *Suppose the satisfiable existentially constrained term $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,]$ such that $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] \to_{\rho,\gamma} \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$, where $\rho\colon \Pi Z.\ \ell \to r\ [\,\pi\,]$ is a left-linear constrained rewrite rule satisfying $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(s,\varphi) = \varnothing$. Then the following statements hold:*

(1) $\mathcal{BV}\mathrm{ar}(\exists\vec{x}.\ \varphi) \subseteq \mathcal{BV}\mathrm{ar}(\exists\vec{y}.\ \psi)$.
(2) $\mathcal{BV}\mathrm{ar}(\exists\vec{x}.\ \varphi) \cap \mathcal{V}\mathrm{ar}(l\gamma, r\gamma, \pi\gamma) = \varnothing$.
(3) $Y \cup \mathcal{BV}\mathrm{ar}(\exists\vec{y}.\ \psi) = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t)) \cup \mathcal{V}\mathrm{ar}(\psi)$.

We are ready to show that our rewrite steps are well-defined.

Theorem 3.10 (Well-Definedness of Rewrite Steps). *Let ρ be a left-linear constrained rewrite rule and $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,]$ a satisfiable existentially constrained term such that $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(s,\varphi) = \varnothing$. If $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] \to_\rho \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ then $\Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ is a satisfiable existentially constrained term.*

Proof. Suppose $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] \to^{p}_{\rho,\gamma} \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$, where $\rho\colon \Pi Z.\ \ell \to r\ [\,\pi\,]$. Then we have (1) $\mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell)$, (2) $s|_p = \ell\gamma$, (3) $\gamma(x) \in \mathcal{V}\mathrm{al} \cup X$ for any $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$, (4) $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (\exists\vec{z}.\ \pi\gamma)$, where $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$, and $t = s[r\gamma]$, $\psi = \varphi \wedge \pi\gamma$, $\{\vec{y}\} = \mathcal{V}\mathrm{ar}(\psi) \setminus \mathcal{V}\mathrm{ar}(t)$, and $Y = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t))$. Clearly, $\exists\vec{y}.\ \psi$ is an existential quantified constraint.

We show $\mathcal{FV}\mathrm{ar}(\exists\vec{y}.\ \psi) \subseteq Y \subseteq \mathcal{V}\mathrm{ar}(t)$ and $\mathcal{BV}\mathrm{ar}(\exists\vec{y}.\ \psi) \cap \mathcal{V}\mathrm{ar}(t) = \varnothing$. From Lemma 3.8, $\mathcal{FV}\mathrm{ar}(\exists\vec{y}.\ \psi) \subseteq \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t)) = Y$ follows. Since $\{\vec{y}\} = \mathcal{V}\mathrm{ar}(\psi) \setminus \mathcal{V}\mathrm{ar}(t)$, $\{\vec{y}\} \cap \mathcal{V}\mathrm{ar}(t) = \varnothing$ clearly holds. It remains to show $Y = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t)) \subseteq \mathcal{V}\mathrm{ar}(t)$. However, as $X \cap \mathcal{V}\mathrm{ar}(t) \subseteq \mathcal{V}\mathrm{ar}(t)$, it suffices to show $\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \subseteq \mathcal{V}\mathrm{ar}(t)$. Let $x \in \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) = \mathcal{V}\mathrm{ar}(r) \setminus \mathcal{V}\mathrm{ar}(\ell)$. Then we obtain by $\gamma(x) = x$ that $x \notin \mathcal{V}\mathrm{ar}(\ell) = \mathcal{D}\mathrm{om}(\gamma)$. Hence, from $x \in \mathcal{V}\mathrm{ar}(r)$, it follows that $x \in \mathcal{V}\mathrm{ar}(r\gamma) \subseteq \mathcal{V}\mathrm{ar}(t)$.

Finally, we show that $\Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ is satisfiable. By the satisfiability of $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,]$, there exists a valuation ξ such that $\vDash_{\mathcal{M},\xi} \exists\vec{x}.\ \varphi$. Hence by (4), $\vDash_{\mathcal{M},\xi} \exists\vec{z}.\ \pi\gamma$. Since $\{\vec{x}\} \subseteq \mathcal{V}\mathrm{ar}(\varphi)$ and $\{\vec{z}\} \subseteq \mathcal{V}\mathrm{ar}(\rho)$, we have $\{\vec{x}\} \cap \{\vec{z}\} = \varnothing$. Thus, one can extend the valuation ξ to ξ' such that $\vDash_{\mathcal{M},\xi'} \varphi \wedge \pi\gamma$. This also gives $\vDash_{\mathcal{M},\xi'} \exists\vec{y}.\ \psi$. Hence it follows that $\Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ is satisfiable. □

Remark 3.11. In our new notion an instance of the constraint of the used rewrite rule is attached to the constraint of the constrained term. Previously, the constraint remained unchanged [7]. This implies that in each rewrite step a fresh variant of a constrained rewrite rule is needed in order to prevent variable clashes. Therefore, the constraint part of a constrained term may grow along rewriting sequences. However, we expect that this does not cause any troubles in actual implementations as rewrite steps can be combined with simplification of constraints. For example, in Example 3.6 one may perform simplifications after the rewrite step:

$$\begin{aligned}&\Pi\{x\}.\ \mathrm{f}(x)\ [\,x > 0\,]\\ &\to_\rho \Pi\{y'\}.\ \mathrm{g}(y')\ [\,\exists x.\ x > 0 \wedge x \geqslant 0 \wedge y' > x\,]\\ &\sim\ \Pi\{y'\}.\ \mathrm{g}(y')\ [\,y' > 1\,]\end{aligned}$$

In the following, as we focus on the theoretical aspects of rewrite steps, we do not consider simplifications of constraints.

4 Simulating the State-of-the-Art of Non-Quantified Constrained Rewriting

Our new formalism for constrained rewriting is closely related to the original definition which was introduced in [7]. In this and the following sections, we formally describe this relation. To reflect the idea behind the original constrained terms of [7] and to prepare it for a suitable comparison to our existentially constrained terms, we introduce the concept of *non-quantified* constrained terms extended with the Π-notation as follows:

Definition 4.1 (Non-Quantified Terms Extended with Π-notation). A *non-quantified term (extended with Π-notation)* is a triple $\langle X, s, \varphi\rangle$, written as $\Pi X.\ s\ [\,\varphi\,]$, of a set X of theory variables, a term s, and a logical constraint φ such that $\mathcal{V}\mathrm{ar}(\varphi) \subseteq X \subseteq \mathcal{V}\mathrm{ar}(\varphi, s)$.

Example 4.2. For example, $\Pi\{x,y\}.\ \mathrm{f}(x)\ [\,x < 2 \wedge 0 > y\,]$ and $\Pi\{x,y,z\}.\ \mathrm{h}(x,y)\ [\,x < y \wedge x + y + 1 = z\,]$ are non-quantified constrained terms.

For non-quantified constrained terms, satisfiability and equivalence are defined in the usual way. A constrained term $s\ [\,\varphi\,]$ without Π-notation can be easily lifted to one with Π-notation as $\Pi\mathcal{V}\mathrm{ar}(\varphi).\ s\ [\,\varphi\,]$. Throughout this paper, we disambiguate the two notions of constrained terms by explicitly naming them *non-quantified constrained terms* and existentially constrained terms.

We focus now on rewrite steps of non-quantified constrained terms (extended with Π-notation), which reflects the original notion of rewriting constrained terms [7]. For equational theories, Aoto et al. in [1] introduced special Π-notations in order to have explicit sets of logical variables which the ones that need to be instantiated by values. The definition of constrained equation $\Pi Z.\ \ell \approx r\ [\,\pi\,]$ is very similar to $\Pi Z.\ \ell \to r\ [\,\pi\,]$, albeit a slight difference is added to the definition of constrained equations in [1]. In particular, we

request that $Z \subseteq \mathcal{V}\mathrm{ar}(\ell, r, \pi)$ for a constrained rule $\Pi Z.\ \ell \to r\ [\pi]$ in order to avoid redundant variables in Z. The variables in Z of $\Pi Z.\ \ell \to r\ [\pi]$ and $\Pi Z.\ \ell \approx r\ [\pi]$ are the ones that need to be instantiated by values in rewrite steps. Rewrite steps using a constrained rewrite rule $\rho\colon \Pi Z.\ \ell \to r\ [\pi]$ on a non-quantified constrained term are performed as follows:

Definition 4.3 (Constrained Rewriting of Non-Quantified Terms). Let $\Pi X.\ s\ [\varphi]$ be a satisfiable non-quantified constrained term and suppose $\mathcal{V}\mathrm{ar}(\rho) \cap (X \cup \mathcal{V}\mathrm{ar}(s, \varphi)) = \varnothing$. Then we obtain the rewrite step $\Pi X.\ s\ [\varphi] \to^{p}_{\rho,\sigma} \Pi Y.\ t\ [\varphi]$ if there exists a position $p \in \mathcal{P}\mathrm{os}_{\mathcal{F}}(s)$, a substitution σ with $\mathcal{D}\mathrm{om}(\sigma) = \mathcal{V}\mathrm{ar}(\ell, r, \pi)$, $\ell\sigma = s|_p$, $\sigma(x) \in \mathcal{V}\mathrm{al} \cup X$ for all $x \in Z$, and $\vDash_{\mathcal{M}} \varphi \Rightarrow \pi\sigma$, where $t = s[r\sigma]_p$ and $Y = X \cap \mathcal{V}\mathrm{ar}(t, \varphi)$.

It is easy to check that non-quantified constrained rules in [7] are covered by taking $X = Y = \mathcal{V}\mathrm{ar}(\varphi)$ and $Z := \mathcal{LV}\mathrm{ar}(\ell \to r\ [\pi])$.

Example 4.4. We revisit Examples 1.1 and 1.2. Consider the constrained rewrite rule $\rho\colon \Pi\{x', y'\}.\ \mathrm{f}(x') \to \mathrm{g}(y')\ [x' \geqslant 1 \wedge x' + 1 \geqslant y']$. Then the rewrite steps in Example 1.1 are encoded as follows.

- Take $p = \varepsilon$ and $\sigma = \{x' \mapsto x, y' \mapsto 3\}$. We obtain the rewrite step $\Pi\{x\}.\ \mathrm{f}(x)\ [x > 2] \to^{p}_{\rho,\sigma} \Pi\{x\}.\ \mathrm{g}(3)\ [x > 2]$. Note here that $\sigma(x') = x \in \mathcal{V}\mathrm{al} \cup X$ and $\sigma(y') = 3 \in \mathcal{V}\mathrm{al} \cup X$. Also, we have $(x' \geqslant 1 \wedge x' + 1 \geqslant y')\sigma = (x \geqslant 1 \wedge x + 1 \geqslant 3)$ and $\vDash_{\mathcal{M}} (x > 2 \Rightarrow (x \geqslant 1 \wedge x + 1 \geqslant 3))$.
- Take $p = \varepsilon$ and $\sigma = \{x' \mapsto x, y' \mapsto x\}$. We obtain the rewrite step $\Pi\{x\}.\ \mathrm{f}(x)\ [x > 2] \to^{p}_{\rho,\sigma} \Pi\{x\}.\ \mathrm{g}(x)\ [x > 2]$. Note here that $\sigma(x') = \sigma(y') = x \in \mathcal{V}\mathrm{al} \cup X$. Also, we have $(x' \geqslant 1 \wedge x' + 1 \geqslant y')\sigma = (x \geqslant 1 \wedge x + 1 \geqslant x)$ and $\vDash_{\mathcal{M}} (x > 2 \Rightarrow (x \geqslant 1 \wedge x + 1 \geqslant x))$.
- We have a rewrite step $\Pi\{x, y\}.\ \mathrm{f}(x)\ [x > 2 \wedge 0 > y] \to^{p}_{\rho,\sigma} \Pi\{x, y\}.\ \mathrm{g}(y)\ [x > 2 \wedge 0 > y]$, where $p = \varepsilon$ and $\sigma = \{x' \mapsto x, y' \mapsto y\}$. Note here that $\sigma(x') = x \in \mathcal{V}\mathrm{al} \cup X$ and $\sigma(y') = y \in \mathcal{V}\mathrm{al} \cup X$. Also, we have $(x' \geqslant 1 \wedge x' + 1 \geqslant y')\sigma = (x \geqslant 1 \wedge x + 1 \geqslant y)$ and $\vDash_{\mathcal{M}} (x > 2 \wedge 0 > y \Rightarrow (x \geqslant 1 \wedge x + 1 \geqslant y))$.

Similarly, let $\rho\colon \Pi\{x', y', z'\}.\ \mathrm{h}(x', y') \to \mathrm{g}(z')\ [(x' + y') + 1 = z']$ from which we obtain $\Pi\{x, y, z\}.\ \mathrm{h}(x, y)\ [x < y \wedge x + y + 1 = z] \to^{p}_{\rho,\sigma} \Pi\{x, y, z\}.\ \mathrm{g}(z)\ [x < y \wedge x + y + 1 = z]$ as in Example 1.2.

Below, we show the relation between the current state-of-the-art of constrained rewrite steps and our new definition. We begin by defining translations between them.

Definition 4.5 (Existential Extension and Existential Removing Translations).

(1) An *existential extension* ext of a non-quantified constrained term is defined as $\mathrm{ext}(\Pi X.\ s\ [\varphi]) = \Pi Y.\ s\ [\exists \vec{x}.\ \varphi]$ where $\{\vec{x}\} = \mathcal{V}\mathrm{ar}(\varphi) \setminus \mathcal{V}\mathrm{ar}(s)$ and $Y = X \setminus \{\vec{x}\}$.
(2) An *existential removing* rmv of an existentially constrained term is defined as $\mathrm{rmv}(\Pi X.\ s\ [\exists \vec{x}.\ \varphi]) = \Pi X \cup \{\vec{x}\}.\ s\ [\varphi]$.

Example 4.6 (Cont'd of Example 4.2). The two non-quantified constrained terms in Example 4.2 are translated to existentially constrained terms as follows: $\mathrm{ext}(\Pi\{x, y\}.\ \mathrm{f}(x)\ [x < 2 \wedge 0 > y]) = \Pi\{x\}.\ \mathrm{f}(x)\ [\exists\{y\}.\ x < 2 \wedge 0 > y]$ and $\mathrm{ext}(\Pi\{x, y, z\}.\ \mathrm{h}(x, y)\ [x < y \wedge x + y + 1 = z]) = \Pi\{x, y\}.\ \mathrm{h}(x, y)\ [\exists\{z\}.\ x < y \wedge x + y + 1 = z]$. Similarly, we have $\mathrm{rmv}(\Pi\{x\}.\ \mathrm{f}(x)\ [\exists\{y\}.\ x < 2 \wedge 0 > y]) = \Pi\{x, y\}.\ \mathrm{f}(x)\ [x < 2 \wedge 0 > y]$ and $\mathrm{rmv}(\Pi\{x, y\}.\ \mathrm{h}(x, y)\ [\exists\{z\}.\ x < y \wedge x + y + 1 = z]) = \Pi\{x, y, z\}.\ \mathrm{h}(x, y)\ [x < y \wedge x + y + 1 = z]$.

The next lemmata straightforwardly follow from Definition 4.5.

Lemma 4.7. *For any non-quantified constrained term $\Pi X.\ s\ [\varphi]$, $\mathrm{ext}(\Pi X.\ s\ [\varphi])$ is an existentially constrained term; for any existentially constrained term $\Pi X.\ s\ [\exists \vec{x}.\ \varphi]$ $\mathrm{rmv}(\Pi X.\ s\ [\exists \vec{x}.\ \varphi])$ is a non-quantified constrained term.*

Lemma 4.8. *The translation* rmv ∘ ext *is the identity translation on non-quantified constrained terms; the translation* ext ∘ rmv *is the identity translation on existentially constrained terms.*

We show that any rewrite step on existentially constrained terms results in a $\tilde{\to}$-rewrite step on non-quantified constrained terms obtained by existential removing. We need the following lemma.

Lemma 4.9. *Let $\Pi X.\ s\ [\varphi]$ be a non-quantified constrained term. Suppose $\vDash_{\mathcal{M}} \varphi \Rightarrow \exists \vec{z}.\ \pi$ with $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus (X \cup \mathcal{V}\mathrm{ar}(\varphi))$ and $\mathcal{V}\mathrm{ar}(s) \cap \{\vec{z}\} = \varnothing$. Then $\Pi X.\ s\ [\varphi] \sim \Pi X \cup \mathcal{V}\mathrm{ar}(\pi).\ s\ [\varphi \wedge \pi]$.*

Proof. ($\subseteq$) Let σ be an X-valued substitution such that $\sigma \vDash_{\mathcal{M}} \varphi$. Then we have $X \cup \mathcal{V}\mathrm{ar}(\varphi) \subseteq \mathcal{VD}\mathrm{om}(\sigma)$ and $\vDash_{\mathcal{M}} \varphi\sigma$. Let γ be a valuation such that $\gamma(x) = \sigma(x)$ for any $x \in X \cup \mathcal{V}\mathrm{ar}(\varphi)$. From the latter, we have $\vDash_{\mathcal{M},\gamma} \varphi$ and hence $\vDash_{\mathcal{M},\gamma} \exists \vec{z}.\ \pi$ by our assumption. Thus, there exists a sequence $\vec{v} \in \mathcal{V}\mathrm{al}^*$ of values so that $\vDash_{\mathcal{M},\gamma} \pi\kappa$, where $\kappa = \{\vec{z} \mapsto \vec{v}\}$. As $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus (X \cup \mathcal{V}\mathrm{ar}(\varphi))$, we have $\mathcal{V}\mathrm{ar}(\pi\kappa\sigma) = \varnothing$ and $\vDash_{\mathcal{M}} \pi\kappa\sigma$. Take a substitution $\delta = \sigma \circ \kappa$. Then, it follows δ is $\mathcal{V}\mathrm{ar}(\pi)$-valued and $\vDash_{\mathcal{M}} \varphi\delta$. For any $x \in X$, if $x \in \mathcal{D}\mathrm{om}(\kappa)$, then $\delta(x) = \sigma(\kappa(x)) = \kappa(x) \in \mathcal{V}\mathrm{al}$. If $x \notin \mathcal{D}\mathrm{om}(\kappa)$, then $\delta(x) = \sigma(\kappa(x)) = \sigma(x) \in \mathcal{V}\mathrm{al}$, because σ is X-valued. Thus, δ is also X-valued. By our assumption $\mathcal{V}\mathrm{ar}(s) \cap \{\vec{z}\} = \varnothing$, and thus $s\delta = s\kappa\sigma = s\sigma$. Also, as $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus (X \cup \mathcal{V}\mathrm{ar}(\varphi))$, we have $\{\vec{z}\} \cap \mathcal{V}\mathrm{ar}(\varphi) = \varnothing$, and therefore, $\varphi\delta = \varphi\kappa\sigma = \varphi\sigma$. Hence, $\mathcal{V}\mathrm{ar}(\varphi\delta) = \varnothing$ and $\vDash_{\mathcal{M}} \varphi\delta$. All in all, δ is $X \cup \mathcal{V}\mathrm{ar}(\pi)$-valued, $s\sigma = s\delta$, and $\delta \vDash_{\mathcal{M}} \varphi \wedge \pi$. The ($\supseteq$)-part is trivial. □

Theorem 4.10 (Simulation of Rewrite Steps by Existential Removing). *Let ρ be a left-linear constrained rewrite rule. If we have $\Pi X.\ s\ [\exists \vec{x}.\ \varphi] \to_{\rho} \Pi Y.\ t\ [\exists \vec{y}.\ \psi]$ then $\mathrm{rmv}(\Pi X.\ s\ [\exists \vec{x}.\ \varphi]) \tilde{\to} \mathrm{rmv}(\Pi Y.\ t\ [\exists \vec{y}.\ \psi])$.*

Proof. Let $\rho\colon \Pi Z.\ \ell \to r\ [\pi]$ be a left-linear constrained rewrite rule and suppose that $\Pi X.\ s\ [\exists \vec{x}.\ \varphi] \to^{p}_{\rho,\gamma} \Pi Y.\ t\ [\exists \vec{y}.\ \psi]$ such that $\mathcal{V}\mathrm{ar}(s, \varphi) \cap \mathcal{V}\mathrm{ar}(\rho) = \varnothing$. Then we have (1) $\mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell)$, (2) $s|_p = \ell\gamma$, (3) $\gamma(x) \in \mathcal{V}\mathrm{al} \cup X$ for any $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$, and (4) $\vDash_{\mathcal{M}} (\exists \vec{x}.\ \varphi) \Rightarrow (\exists \vec{z}.\ \pi\gamma)$, where $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$, $t = s[r\gamma]$, $\psi = \varphi \wedge \pi\gamma$, $\{\vec{y}\} = \mathcal{V}\mathrm{ar}(\psi) \setminus \mathcal{V}\mathrm{ar}(t)$, and $Y = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t))$.

Note that we have $\mathrm{rmv}(\Pi X.\ s\ [\exists \vec{x}.\ \varphi]) = \Pi X \cup \{\vec{x}\}.\ s\ [\varphi]$. We begin by showing the following equivalence: (5) $\Pi X \cup \{\vec{x}\}.\ s\ [\varphi] \sim \Pi X \cup \{\vec{x}\} \cup \mathcal{V}\mathrm{ar}(\pi\gamma) \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho).\ s\ [\varphi \wedge \pi\gamma \wedge \bigwedge_{z \in \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)} (z = z)]$. For this, we use Lemma 4.9. By (4), we have that $\vDash_{\mathcal{M}} \varphi \Rightarrow \exists \vec{z}.\ \pi\gamma$ holds. Since we have $\vDash_{\mathcal{M}} \bigwedge_{z \in \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)} (z = z)$, also $\vDash_{\mathcal{M}} \varphi \Rightarrow \exists \vec{z}.\ \pi\gamma \wedge \bigwedge_{z \in \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)} (z = z)$ holds. Now, we have

$$
\begin{aligned}
&\mathcal{V}\mathrm{ar}(\pi\gamma \wedge \textstyle\bigwedge_{z \in \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)} (z = z)) \setminus (X \cup \{\vec{x}\} \cup \mathcal{V}\mathrm{ar}(\varphi)) \\
&= ((\mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)) \cup (\textstyle\bigcup_{y \in \mathcal{V}\mathrm{ar}(\ell) \cap \mathcal{V}\mathrm{ar}(\pi)} \mathcal{V}\mathrm{ar}(\gamma(y))) \\
&\quad \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)) \setminus (X \cup \{\vec{x}\} \cup \mathcal{V}\mathrm{ar}(\varphi)) \\
&= ((\mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)) \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)) \setminus (X \cup \{\vec{x}\} \cup \mathcal{V}\mathrm{ar}(\varphi)) \\
&= \mathcal{V}\mathrm{ar}(r, \pi) \setminus \mathcal{V}\mathrm{ar}(\ell).
\end{aligned}
$$

Take $\{\vec{z}'\} = \mathcal{V}\mathrm{ar}(r,\pi) \setminus \mathcal{V}\mathrm{ar}(\ell) = \mathcal{V}\mathrm{ar}(\pi\gamma \wedge \bigwedge_{z\in\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)}(z = z)) \setminus (X \cup \{\vec{x}\} \cup \mathcal{V}\mathrm{ar}(\varphi))$. By $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(s) = \varnothing$, we have $\{\vec{z}'\} \cap \mathcal{V}\mathrm{ar}(s) = \varnothing$. Furthermore, since $\{\vec{z}'\} \supseteq \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) = \{\vec{z}\}$, we have $\vDash_{\mathcal{M}} \varphi \Rightarrow \exists\vec{z}'.\ \pi\gamma \wedge \bigwedge_{z\in\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)}(z = z)$. Thus, by Lemma 4.9, we conclude the equivalence (5).

We proceed to show the rewrite step $\Pi X \cup \{\vec{x}\} \cup \mathcal{V}\mathrm{ar}(\pi\gamma) \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho).\ s\ [\varphi \wedge \pi\gamma \wedge \bigwedge_{z\in\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)}(z = z)] \to^{p}_{\rho'} \Pi Y \cup \{\vec{y}\} \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho).\ t\ [\varphi \wedge \pi\gamma \wedge \bigwedge_{z\in\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)}(z = z)]$ over non-quantified constrained terms, by a variant $\rho' \colon \Pi Z'.\ \ell' \to r'\ [\pi']$ of the constrained rewrite rule ρ such that $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(\rho') = \varnothing$. Accordingly, we have a variable renaming ξ such that $Z = \xi(Z')$, $\ell = \ell'\xi$, $r = r'\xi$, and $\pi = \pi'\xi$. We also suppose that $\mathcal{V}\mathrm{ar}(\rho') \cap \mathcal{V}\mathrm{ar}(s, \varphi \wedge \pi\gamma \wedge \bigwedge_{z\in\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)}(z = z)) = \varnothing$.

Take $\delta = \gamma \circ \xi$. Then, by (1) and our assumption on ξ, we have $\mathcal{D}\mathrm{om}(\delta) = \mathcal{V}\mathrm{ar}(\ell', r', \pi')$. Also, $\ell'\delta = \ell'\xi\gamma = \ell\gamma = s|_p$ holds by (2) and the definition of δ.

We now show $\delta(x) \in \mathcal{V}\mathrm{al} \cup X \cup \{\vec{x}\} \cup \mathcal{V}\mathrm{ar}(\pi\gamma) \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)$ for all $x \in Z'$. To this end assume that $x \in Z'$. If $x \in \mathcal{V}\mathrm{ar}(\ell')$, then $\delta(x) = x\xi\gamma = \gamma(y)$ for some $y \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$. Hence, $\delta(x) \in \mathcal{V}\mathrm{al} \cup X$ by (3). Thus, suppose $x \in Z' \setminus \mathcal{V}\mathrm{ar}(\ell')$. Then, because of $Z' \subseteq \mathcal{V}\mathrm{ar}(\ell', r', \pi')$, we know $x \in \mathcal{V}\mathrm{ar}(r', \pi') \setminus \mathcal{V}\mathrm{ar}(\ell')$. Thus, $\delta(x) = x\xi\gamma = \gamma(y)$ for some $y \in \mathcal{V}\mathrm{ar}(r, \pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$. Hence, if $y \in \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$ then $\delta(x) = \gamma(y) \in \mathcal{V}\mathrm{ar}(\pi\gamma)$, and if $y \in \mathcal{V}\mathrm{ar}(r) \setminus \mathcal{V}\mathrm{ar}(\ell)$ then $\delta(x) = \gamma(y) = y \in \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)$.

Furthermore, we have $\vDash_{\mathcal{M}} (\varphi \wedge \pi\gamma \wedge \bigwedge_{z\in\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)}(z = z)) \Rightarrow \pi'\delta$, because $\pi'\delta = \pi'\xi\gamma = \pi\gamma$. We also have $t = s[r\gamma]_p = s[r'\xi\gamma]_p = s[r'\delta]_p$. Thus, it remains to show that $Y \cup \{\vec{y}\} \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) = (X \cup \{\vec{x}\} \cup \mathcal{V}\mathrm{ar}(\pi\gamma) \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)) \cap \mathcal{V}\mathrm{ar}(t, \varphi \wedge \pi\gamma \wedge \bigwedge_{z\in\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)}(z = z))$. We denote the right-hand side of the equation by *rhs*, and obtain: $(rhs) = ((X \cup \{\vec{x}\} \cup \mathcal{V}\mathrm{ar}(\pi\gamma) \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)) \cap \mathcal{V}\mathrm{ar}(t)) \cup ((X \cup \{\vec{x}\} \cup \mathcal{V}\mathrm{ar}(\pi\gamma) \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)) \cap \mathcal{V}\mathrm{ar}(\varphi \wedge \pi\gamma \wedge \bigwedge_{z\in\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)}(z = z))) = (X \cap \mathcal{V}\mathrm{ar}(t)) \cup (\mathcal{V}\mathrm{ar}(\pi\gamma) \cap \mathcal{V}\mathrm{ar}(t)) \cup \mathcal{V}\mathrm{ar}(\varphi \wedge \pi\gamma) \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) = (X \cap \mathcal{V}\mathrm{ar}(t)) \cup (\mathcal{V}\mathrm{ar}(\varphi \wedge \pi\gamma \wedge \bigwedge_{z\in\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)}(z = z))) \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) = Y \cup \{\vec{y}\} \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)$ (by Lemma 3.9). Putting all of this together gives the claimed rewrite step. To complete the proof, we need to show the equivalence $\Pi Y \cup \{\vec{y}\} \cup \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho).\ t\ [\varphi \wedge \pi\gamma \wedge \bigwedge_{z\in\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)}(z = z)] \sim \Pi Y \cup \{\vec{y}\}.\ t\ [\exists\vec{y}.\ \varphi \wedge \pi\gamma]) = \mathrm{rmv}(\Pi Y.\ t\ [\exists\vec{y}.\ \psi])$ which should be trivial to see. □

Example 4.11. Let us consider the most general rewrite step $\Pi\{x\}.\ \mathrm{f}(x)\ [x > 0] \to_\rho \Pi\{y'\}.\ \mathrm{g}(y')\ [\exists x.\ x > 0 \wedge x \geqslant 0 \wedge y' > x]$ of Example 3.6 where $\rho \colon \Pi\{x', y'\}.\ \mathrm{f}(x') \to \mathrm{g}(y')\ [x' \geqslant 0 \wedge y' > x']$. We obtain the rewrite step over the non-quantified constrained terms $\mathrm{rmv}(\Pi\{x\}.\ \mathrm{f}(x)\ [x > 0]) \xrightarrow{\sim} \mathrm{rmv}(\Pi\{y'\}.\ \mathrm{g}(y')\ [\exists x.\ x > 0 \wedge x \geqslant 0 \wedge y' > x])$ as follows:

$$\begin{aligned}
&\mathrm{rmv}(\Pi\{x\}.\ \mathrm{f}(x)\ [x > 0]) \\
&\quad = \Pi\{x\}.\ \mathrm{f}(x)\ [x > 0] \\
&\quad \sim \Pi\{x, y'\}.\ \mathrm{f}(x)\ [x > 0 \wedge x \geqslant 0 \wedge y' > x] \\
&\quad \to_\rho \Pi\{x, y'\}.\ \mathrm{g}(y')\ [x > 0 \wedge x \geqslant 0 \wedge y' > x] \\
&\quad = \mathrm{rmv}(\Pi\{y'\}.\ \mathrm{g}(y')\ [\exists x.\ x > 0 \wedge x \geqslant 0 \wedge y' > x])
\end{aligned}$$

5 Embedding Non-Quantified Constrained Rewriting into the Most General Form

In this section, we concern ourselves with the problem of characterizing the opposite direction. That is, whether any rewrite step on non-quantified constrained terms results in a rewrite step on existentially constrained terms that are obtained by the existential extension. Naively this is not the case as depicted by the following example.

Example 5.1. In Example 4.4, we encoded the rewrite steps of Example 1.1 by rewriting of non-quantified constrained terms with the constrained rewrite rule $\rho \colon \Pi\{x', y'\}.\ \mathrm{f}(x') \to \mathrm{g}(y')\ [x' \geqslant 1 \wedge x' + 1 \geqslant y']$. We had $\Pi\{x\}.\ \mathrm{f}(x)\ [x > 2] \to_\rho \Pi\{x\}.\ \mathrm{g}(3)\ [x > 2]$ and $\Pi\{x\}.\ \mathrm{f}(x)\ [x > 2] \to_\rho \Pi\{x\}.\ \mathrm{g}(x)\ [x > 2]$. Applying the following most general rewrite steps to $\mathrm{ext}(\Pi\{x\}.\ \mathrm{f}(x)\ [x > 2]) = \Pi\{x\}.\ \mathrm{f}(x)\ [x > 2])$ yields $\Pi\{x\}.\ \mathrm{f}(x)\ [x > 2] \to_\rho \Pi\{x, y'\}.\ \mathrm{g}(y')\ [x > 2 \wedge x \geqslant 1 \wedge x + 1 \geqslant y']$. Unfortunately, neither $\mathrm{ext}(\Pi\{x\}.\ \mathrm{g}(3)\ [x > 2]) = \Pi\{x\}.\ \mathrm{g}(3)\ [x > 2])$ nor $\mathrm{ext}(\Pi\{x\}.\ \mathrm{f}(x)\ [x > 2]) = \Pi\{x\}.\ \mathrm{f}(x)\ [x > 2])$ is obtained. However, note that $\Pi\{x, y'\}.\ \mathrm{g}(y')\ [x > 2 \wedge x \geqslant 1 \wedge x + 1 \geqslant y'] \supseteq\!\!\sim \Pi\{x\}.\ \mathrm{g}(3)\ [x > 2]$, as well as $\Pi\{x, y'\}.\ \mathrm{g}(y')\ [x > 2 \wedge x \geqslant 1 \wedge x + 1 \geqslant y'] \supseteq \Pi\{x\}.\ \mathrm{g}(x)\ [x > 2]$ holds.

In fact, as the example above demonstrates, we can give the following characterization of non-quantified constrained rewriting in our formalism of most general rewriting. Namely most general rewrite steps subsume any reduct of non-quantified rewrite steps—this relation serves as the main motivation for us to call our new definition *most general rewrite steps*.

Theorem 5.2 (Simulation of Rewrite Steps by Existential Extension). *Let ρ be a left-linear constrained rewrite rule. If we have $\Pi X.\ s\ [\varphi] \to_\rho \Pi Y.\ t\ [\varphi]$, then $\mathrm{ext}(\Pi X.\ s\ [\varphi]) \to \cdot \supseteq\!\!\sim \mathrm{ext}(\Pi Y.\ t\ [\varphi])$.*

Proof. Assume $\Pi X.\ s\ [\varphi] \to^{p}_{\rho,\gamma} \Pi Y.\ t\ [\varphi]$ where $\rho \colon \Pi Z.\ \ell \to r\ [\pi]$ is a left-linear constrained rewrite rule with $\mathcal{V}\mathrm{ar}(\rho) \cap (X \cup \mathcal{V}\mathrm{ar}(s, \varphi)) = \varnothing$. Note that this is a rewrite step for non-quantified constrained terms. Thus, we have (1) $\mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell, r, \pi)$, (2) $s|_p = \ell\gamma$, (3) $\gamma(x) \in \mathcal{V}\mathrm{al} \cup X$ for all $x \in Z$, and (4) $\vDash_{\mathcal{M}} \varphi \Rightarrow \pi\gamma$. Furthermore, $t = s[r\gamma]_p$ and $Y = X \cap \mathcal{V}\mathrm{ar}(t, \varphi)$. W.l.o.g. we assume $\mathcal{V}\mathrm{ar}(\rho) \cap (Y \cup \mathcal{V}\mathrm{ar}(t, \varphi)) = \varnothing$. Let $\mathrm{ext}(\Pi X.\ s\ [\varphi]) = \Pi X \setminus \{\vec{x}\}.\ s\ [\exists\vec{x}.\ \varphi]$ with $\{\vec{x}\} = \mathcal{V}\mathrm{ar}(\varphi) \setminus \mathcal{V}\mathrm{ar}(s)$ and $\mathrm{ext}(\Pi Y.\ t\ [\psi]) = \Pi Y \setminus \{\vec{y}\}.\ t\ [\exists\vec{y}.\ \psi]$, with $\vec{y} = \mathcal{V}\mathrm{ar}(\psi) \setminus \mathcal{V}\mathrm{ar}(t)$.

We first show that $\Pi X \setminus \{\vec{x}\}.\ s\ [\exists\vec{x}.\ \varphi]$ has a ρ-redex at position p using a substitution $\sigma = \gamma|_{\mathcal{V}\mathrm{ar}(\ell)}$. For this, we need to show (5) $\mathcal{D}\mathrm{om}(\sigma) = \mathcal{V}\mathrm{ar}(\ell)$, (6) $s|_p = \ell\sigma$, (7) $\sigma(x) \in \mathcal{V}\mathrm{al} \cup (X \setminus \{\vec{x}\})$ for any $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$, and (8) $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (\exists\vec{z}.\ \pi\sigma)$, where $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$. (5) is obvious by definition of σ. Similarly, (6) follows as $s|_p = \ell\gamma = \ell\sigma$ by (2). We have $\sigma(x) \in \mathcal{V}\mathrm{al} \cup X$ for any $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$, because $\sigma(x) = \gamma(x)$ for all $x \in \mathcal{V}\mathrm{ar}(\ell)$ and $\gamma(x) \in \mathcal{V}\mathrm{al} \cup X$ by (3); moreover, if $\sigma(x) \in X$, then, as $\sigma(x) \in \mathcal{V}\mathrm{ar}(\ell\sigma) \subseteq \mathcal{V}\mathrm{ar}(s)$ by $x \in \mathcal{V}\mathrm{ar}(\ell)$, we have $\sigma(x) \in X \cap \mathcal{V}\mathrm{ar}(s)$, which implies that $\sigma(x) \notin \{\vec{x}\} = \mathcal{V}\mathrm{ar}(\varphi) \setminus \mathcal{V}\mathrm{ar}(s)$. Hence (7) $\sigma(x) \in \mathcal{V}\mathrm{al} \cup (X \setminus \{\vec{x}\})$ for any $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$.

It remains to show (8) $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (\exists\vec{z}.\ \pi\sigma)$. First of all, note that by definition of σ and $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(s) = \varnothing$, we have $\gamma = \gamma' \circ \sigma$ where $\gamma' = \{z \mapsto \gamma(z) \mid z \in \mathcal{V}\mathrm{ar}(r, \pi) \setminus \mathcal{V}\mathrm{ar}(\ell)\}$. Suppose $\vDash_{\mathcal{M},\xi} \exists\vec{x}.\ \varphi$ for a valuation ξ. Then, for some $\kappa = \{\vec{x} \mapsto \vec{v}\}$ with $\vec{v} \in \mathcal{V}\mathrm{al}^*$ we have $\vDash_{\mathcal{M},\xi\circ\kappa} \varphi$. Hence, from (4) it follows $\vDash_{\mathcal{M},\xi\circ\kappa} \pi\gamma$. Then, by $\gamma = \gamma' \circ \sigma$, we have $\vDash_{\mathcal{M},\xi\circ\kappa} \exists\vec{z}.\ \pi\sigma$ where $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$. Also, $\mathcal{FV}\mathrm{ar}(\exists\vec{z}.\ \pi\sigma) \subseteq \mathcal{V}\mathrm{ar}(\ell\gamma) \subseteq \mathcal{V}\mathrm{ar}(s)$. Since $\mathcal{D}\mathrm{om}(\kappa) = \{\vec{x}\} = \mathcal{V}\mathrm{ar}(\varphi) \setminus \mathcal{V}\mathrm{ar}(s)$, we know $\mathcal{D}\mathrm{om}(\kappa) \cap \mathcal{FV}\mathrm{ar}(\exists\vec{z}.\ \pi\sigma) = \varnothing$,

and thus $(\exists\vec{z}.\ \pi\sigma)\kappa = \exists\vec{z}.\ \pi\sigma$. Hence, $\vDash_{\mathcal{M},\xi} \exists\vec{z}.\ \pi\sigma$ follows and therefore we have shown that (8) $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (\exists\vec{z}.\ \pi\sigma)$ holds.

We conclude that $\Pi X \setminus \{\vec{x}\}.\ s\,[\,\exists\vec{x}.\ \varphi\,]$ has a ρ-redex at position p using the substitution σ. Consequently, we obtain the rewrite step $\Pi X \setminus \{\vec{x}\}.\ s\,[\,\exists\vec{x}.\ \varphi\,] \to^{p}_{\rho,\sigma} \Pi Y'.\ t'\,[\,\exists\vec{y}'.\ \psi'\,]$ where $t' = s[r\sigma]_p$, $\psi' = \varphi \wedge \pi\sigma$, $\{\vec{y}'\} = \mathcal{V}\mathrm{ar}(\psi') \setminus \mathcal{V}\mathrm{ar}(t')$, and $Y' = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup ((X \setminus \{\vec{x}\}) \cap \mathcal{V}\mathrm{ar}(t'))$.

It remains to show $\Pi Y'.\ t'\,[\,\exists\vec{y}'.\ \psi'\,] \supseteq \Pi Y \setminus \{\vec{y}\}.\ t\,[\,\exists\vec{y}.\ \psi\,]$.

Prior to that, let us show $\{\vec{y}\} \subseteq \{\vec{y}'\}$. Let $w \in \{\vec{y}\} = \mathcal{V}\mathrm{ar}(\varphi) \setminus \mathcal{V}\mathrm{ar}(t)$. Since $\{\vec{y}'\} = \mathcal{V}\mathrm{ar}(\varphi \wedge \pi\gamma) \setminus \mathcal{V}\mathrm{ar}(t')$, it suffices to show that for any $w \in \mathcal{V}\mathrm{ar}(\varphi)$, it holds that $w \in \mathcal{V}\mathrm{ar}(t')$ implies $w \in \mathcal{V}\mathrm{ar}(t)$. Since $t' = s[r\sigma]$ and $t = s[r\gamma]$, the case $w \in \mathcal{V}\mathrm{ar}(s[\ ])$ is trivial. Thus, suppose $w \in \mathcal{V}\mathrm{ar}(r\sigma)$. Then either $w \in \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)$ or $w \in \mathcal{V}\mathrm{ar}(x\sigma)$ for some $x \in \mathcal{V}\mathrm{ar}(\ell) \cap \mathcal{V}\mathrm{ar}(r)$. The former case contradicts with $w \in \mathcal{V}\mathrm{ar}(\varphi)$ because $\mathcal{V}\mathrm{ar}(\varphi) \cap \mathcal{V}\mathrm{ar}(\rho) = \emptyset$. In the latter case, $w \in \mathcal{V}\mathrm{ar}(r\gamma)$ follows, as $\sigma(x) = \gamma(x)$ for $x \in \mathcal{V}\mathrm{ar}(\ell)$.

We proceed to show that $\Pi Y'.\ t'\,[\,\exists\vec{y}'.\ \psi'\,] \supseteq \Pi Y \setminus \{\vec{y}\}.\ t\,[\,\exists\vec{y}.\ \psi\,]$. Let θ be a $(Y \setminus \{\vec{y}\})$-valued substitution such that $\theta \vDash_{\mathcal{M}} \exists\vec{y}.\ \psi$, i.e. $((\mathcal{V}\mathrm{ar}(\psi) \cup Y) \setminus \{\vec{y}\}) \subseteq \mathcal{V}\mathcal{D}\mathrm{om}(\theta)$ and $\vDash_{\mathcal{M}} (\exists\vec{y}.\ \psi)\theta$. We will prove there exists a substitution θ' such that it is Y'-valued and $\theta' \vDash \exists\vec{y}'.\ \psi'$, i.e. $(Y' \cup (\mathcal{V}\mathrm{ar}(\psi') \setminus \{\vec{y}'\})) \subseteq \mathcal{V}\mathcal{D}\mathrm{om}(\theta')$ and $\vDash_{\mathcal{M}} (\exists\vec{y}'.\ \psi')\theta'$, which satisfies $t\theta = t'\theta'$. For this purpose, we assume w.l.o.g.

- $\mathcal{D}\mathrm{om}(\theta) \cap \mathcal{V}\mathrm{ar}(\rho) = \emptyset$,
- $\mathcal{D}\mathrm{om}(\theta) \cap \{\vec{y}\} = \emptyset$, and
- $\mathcal{V}\mathrm{ar}(\theta(z)) \cap \{\vec{y}\} = \emptyset$ for any $z \in \mathcal{F}\mathcal{V}\mathrm{ar}(\exists\vec{y}.\ \psi)$.

Note here also $\{\vec{y}\} \cap \mathcal{V}\mathrm{ar}(t) = \emptyset$, as $\{\vec{y}\} = \mathcal{V}\mathrm{ar}(\varphi) \setminus \mathcal{V}\mathrm{ar}(t)$. Let us define $\theta' = \theta \cup \{z \mapsto z\gamma\theta \mid z \in \mathcal{V}\mathrm{ar}(r,\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)\}$. As a first step, we show that $\theta(\gamma(x)) = \theta'(\sigma(x))$ for any $x \in \mathcal{V}\mathrm{ar}(\rho)$. Suppose that $x \in \mathcal{V}\mathrm{ar}(\ell)$. Then $\gamma(x) = \sigma(x)$ and since $x\gamma$ is a subterm of $\ell\gamma$, we have that $\mathcal{V}\mathrm{ar}(\gamma(x)) \cap \mathcal{V}\mathrm{ar}(\rho) = \emptyset$. In particular, $\mathcal{V}\mathrm{ar}(\gamma(x)) \cap (\mathcal{V}\mathrm{ar}(r,\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)) = \emptyset$, thus, by definition of θ', we have $\theta'(\gamma(x)) = \theta(\gamma(x))$. Hence $\theta(\gamma(x)) = \theta'(\gamma(x)) = \theta'(\sigma(x))$. Otherwise, suppose that $x \in (\mathcal{V}\mathrm{ar}(r,\pi) \setminus \mathcal{V}\mathrm{ar}(\ell))$. Then, by definition of θ', we have $\theta'(x) = \theta(\gamma(x))$. Since $\mathcal{D}\mathrm{om}(\sigma) = \mathcal{V}\mathrm{ar}(\ell)$, we have $\sigma(x) = x$ and thus $\theta(\gamma(x)) = \theta'(x) = \theta'(\sigma(x))$.

We conclude that $\theta(\gamma(x)) = \theta'(\sigma(x))$ for any $x \in \mathcal{V}\mathrm{ar}(\rho)$. From this, it follows that $\pi\gamma\theta = \pi\sigma\theta'$ and $r\gamma\theta = r\sigma\theta'$. We further have $\theta'(x) = \theta(x)$ for any $x \in \mathcal{V}\mathrm{ar}(s)$ by definition of θ' as $\mathcal{V}\mathrm{ar}(s) \cap \mathcal{V}\mathrm{ar}(\rho) = \emptyset$. Therefore, we have $t\theta = s[r\gamma]\theta = s\theta[r\gamma\theta] = s\theta'[r\sigma\theta'] = s[r\sigma]\theta' = t'\theta'$.

Let us claim that $\theta'(Y') \subseteq \mathcal{V}\mathrm{al}$. Assume $y \in Y'$. We are now going to show that $\theta'(y) \in \mathcal{V}\mathrm{al}$ by distinguishing two cases.

- Assume $y \in \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)$. Then, $y \in \mathcal{V}\mathrm{ar}(r) \setminus \mathcal{V}\mathrm{ar}(\ell)$ and hence $\theta'(y) = \theta(\gamma(y))$. Since $\mathcal{V}\mathrm{ar}(r,\pi) \setminus \mathcal{V}\mathrm{ar}(\ell) \subseteq Z$ by the definition of ρ, we have that $\gamma(y) \in \mathcal{V}\mathrm{al} \cup X$ by our condition (3). In case $\gamma(y) \in \mathcal{V}\mathrm{al}$, clearly, $\theta'(y) = \theta(\gamma(y)) = \gamma(y) \in \mathcal{V}\mathrm{al}$. Otherwise, we have $\gamma(y) \in X$. As $y \in \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)$ and $\mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell, r, \pi)$, $\gamma(y)$ appears in $t = s[r\gamma]$. Thus, we have $\gamma(y) \in \mathcal{V}\mathrm{ar}(t)$. Moreover, by $\{\vec{y}\} = \mathcal{V}\mathrm{ar}(\varphi) \setminus \mathcal{V}\mathrm{ar}(t)$, we know that $\gamma(y) \notin \{\vec{y}\}$. Thus, $\gamma(y) \in (X \cap \mathcal{V}\mathrm{ar}(t,\varphi)) \setminus \{\vec{y}\} = Y \setminus \{\vec{y}\}$. As θ is $(Y \setminus \{\vec{y}\})$-valued, we conclude $\theta'(y) = \theta(\gamma(y)) \in \mathcal{V}\mathrm{al}$.
- Assume $y \notin \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)$. As $Y' = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup ((X \setminus \{\vec{x}\}) \cap \mathcal{V}\mathrm{ar}(t'))$, we have $y \in ((X \setminus \{\vec{x}\}) \cap \mathcal{V}\mathrm{ar}(t'))$. If $y \notin \mathcal{V}\mathrm{ar}(t) = \mathcal{V}\mathrm{ar}(s[r\gamma]_p)$, then, as $y \in \mathcal{V}\mathrm{ar}(t') = \mathcal{V}\mathrm{ar}(s[r\sigma]_p)$ and $\sigma = \gamma|_{\mathcal{V}\mathrm{ar}(\ell)}$, we have $y \in \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)$. This contradicts our assumption and therefore $y \in \mathcal{V}\mathrm{ar}(t)$. Then, since $y \in X$, we have $y \in (X \cap \mathcal{V}\mathrm{ar}(t,\varphi)) = Y$. Also, by $\{\vec{y}\} = \mathcal{V}\mathrm{ar}(\varphi) \setminus \mathcal{V}\mathrm{ar}(t)$, we know that $y \notin \{\vec{y}\}$. Thus, it follows that $y \in Y \setminus \{\vec{y}\}$, and moreover, that $\theta(y) \in \mathcal{V}\mathrm{al}$ as θ is $(Y \setminus \{\vec{y}\})$-valued. As $y \notin \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)$, we have $\theta'(y) = \theta(y)$ and we obtain $\theta'(y) \in \mathcal{V}\mathrm{al}$.

It remains to show that $\theta' \vDash_{\mathcal{M}} \exists\vec{y}'.\ \psi'$. First, $\mathcal{F}\mathcal{V}\mathrm{ar}(\exists\vec{y}'.\ \psi') \subseteq \mathcal{V}\mathcal{D}\mathrm{om}(\theta')$ follows from $\mathcal{F}\mathcal{V}\mathrm{ar}(\exists\vec{y}'.\ \psi') \subseteq Y'$ and the fact that θ' is Y'-valued as shown above. We are now going to show $\vDash_{\mathcal{M}} (\exists\vec{y}'.\ \psi')\theta'$. From $\vDash_{\mathcal{M}} (\exists\vec{y}.\ \varphi)\theta$, there exists a valuation $\eta = \{\vec{y} \mapsto \vec{v}\}$ with $\vec{v} \in \mathcal{V}\mathrm{al}^*$ such that $\vDash_{\mathcal{M}} \varphi\eta\theta$. Using the second and third assumption on θ which we have assumed above w.l.o.g., it follows that $\vDash_{\mathcal{M}} \varphi\theta\eta$. Thus, from (4), we have that $\vDash_{\mathcal{M}} \pi\gamma\theta\eta$, and hence $\vDash_{\mathcal{M}} \pi\sigma\theta'\eta$. Again, using the second and third assumptions on θ, it follows that $\vDash_{\mathcal{M}} \pi\sigma\eta\theta'$. Also, from $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(\varphi) = \emptyset$, we have that $\varphi\theta = \varphi\theta'$, and using our assumptions on θ it also follows that $\varphi\eta\theta = \varphi\eta\theta'$. Therefore, $\vDash_{\mathcal{M}} \varphi\eta\theta'$ holds and we conclude that $\vDash_{\mathcal{M}} (\varphi \wedge \pi\sigma)\eta\theta'$. We obtain that $\vDash_{\mathcal{M}} \exists\vec{y}.\ (\varphi \wedge \pi\sigma)\theta'$, and $\{\vec{y}\} \subseteq \{\vec{y}'\}$ implies that $\vDash_{\mathcal{M}} \exists\vec{y}'.\ (\varphi \wedge \pi\sigma)\theta'$. We ultimately have $\theta' \vDash_{\mathcal{M}} \exists\vec{y}'.\ \psi'$ which concludes the proof. □

6 Uniqueness of Reducts and Partial Commutation of Rewriting and Equivalence

The aim of this section is to show two useful properties of most general rewriting. First, we show that applying the same constrained rewrite rule at the same position yields a unique term. Second, we show that our new notion of rewriting commutes with the equivalence transformation for pattern-general constrained terms.

Theorem 6.1 (Uniqueness of Reducts). *Let* $\Pi X.\ s\,[\,\exists\vec{x}.\ \varphi\,]$ *be an existentially constrained term and* $p \in \mathcal{P}\mathrm{os}(s)$. *Suppose that* ρ, ρ' *are renamed variants of the same left-linear constrained rule satisfying* $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(s,\varphi) = \emptyset$ *and* $\mathcal{V}\mathrm{ar}(\rho') \cap \mathcal{V}\mathrm{ar}(s,\varphi) = \emptyset$. *If* $\Pi X.\ s\,[\,\exists\vec{x}.\ \varphi\,] \to^{p}_{\rho} \Pi Y.\ t\,[\,\exists\vec{y}.\ \psi\,]$ *and* $\Pi X.\ s\,[\,\exists\vec{x}.\ \varphi\,] \to^{p}_{\rho'} \Pi Y'.\ t'\,[\,\exists\vec{y}'.\ \psi'\,]$ *then* $\Pi Y.\ t\,[\,\exists\vec{y}.\ \psi\,] \sim \Pi Y'.\ t'\,[\,\exists\vec{y}'.\ \psi'\,]$.

Proof. Suppose that $\Pi X.\ s\,[\,\exists\vec{x}.\ \varphi\,] \to^{p}_{\rho,\gamma} \Pi Y.\ t\,[\,\exists\vec{y}.\ \psi\,]$ and $\Pi X.\ s\,[\,\exists\vec{x}.\ \varphi\,] \to^{p}_{\rho',\gamma'} \Pi Y'.\ t'\,[\,\exists\vec{y}'.\ \psi'\,]$, where $\rho\colon \Pi Z.\ \ell \to r\,[\,\pi\,]$ and $\rho'\colon \Pi Z'.\ \ell' \to r'\,[\,\pi'\,]$. Then we have (1) $\mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell)$, (2) $s|_p = \ell\gamma$, (3) $\gamma(x) \in \mathcal{V}\mathrm{al} \cup X$ for any $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$, and (4) $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (\exists\vec{z}.\ \pi\gamma)$, where $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$, $t = s[r\gamma]$, $\psi = \varphi \wedge \pi\gamma$, $\{\vec{y}\} = \mathcal{V}\mathrm{ar}(\psi) \setminus \mathcal{V}\mathrm{ar}(t)$, and $Y = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t))$. Similarly, we have (1') $\mathcal{D}\mathrm{om}(\gamma') = \mathcal{V}\mathrm{ar}(\ell')$, (2') $s|_p = \ell'\gamma'$, (3') $\gamma'(x) \in \mathcal{V}\mathrm{al} \cup X$ for any $x \in \mathcal{V}\mathrm{ar}(\ell') \cap Z'$, and (4') $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (\exists\vec{z}'.\ \pi'\gamma')$, where $\{\vec{z}'\} = \mathcal{V}\mathrm{ar}(\pi') \setminus \mathcal{V}\mathrm{ar}(\ell')$, $t' = s[r'\gamma']$, $\psi' = \varphi \wedge \pi'\gamma'$, $\{\vec{y}'\} = \mathcal{V}\mathrm{ar}(\psi') \setminus \mathcal{V}\mathrm{ar}(t')$, and $Y' = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho') \cup (X \cap \mathcal{V}\mathrm{ar}(t'))$.

Since ρ, ρ' are renamed variants of the same rule, we have renaming $\sigma\colon \mathcal{V}\mathrm{ar}(\rho) \to \mathcal{V}\mathrm{ar}(\rho')$ such that $\sigma(Z) = Z'$, $\sigma(\ell) = \ell'$, $\sigma(r) = r'$, and $\sigma(\pi) = \pi'$.

For each $x \in \mathcal{V}\mathrm{ar}(r,\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$, we have by $x \notin \mathcal{V}\mathrm{ar}(\ell) = \mathcal{D}\mathrm{om}(\gamma)$ that $\gamma(x) = x$, and by $\sigma(x) \notin \mathcal{V}\mathrm{ar}(\ell') = \mathcal{D}\mathrm{om}(\gamma')$ that $\gamma'(\sigma(x)) = \sigma(x)$. Therefore, $\sigma(\gamma(x)) = \sigma(x) = \gamma'(\sigma(x))$ for all $x \in \mathcal{V}\mathrm{ar}(r,\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$. Suppose $x \in \mathcal{V}\mathrm{ar}(\ell)$. Then from $\gamma(x) \in$

$\mathcal{V}\mathrm{ar}(s)$ and $\mathcal{V}\mathrm{ar}(s) \cap \mathcal{V}\mathrm{ar}(\rho) = \emptyset$, we know $\gamma(x) \notin \mathcal{V}\mathrm{ar}(\rho)$, and hence $\gamma(x) \notin \mathcal{D}\mathrm{om}(\sigma)$. Thus, $\sigma(\gamma(x)) = \gamma(x)$. Moreover, since $\ell\gamma = s|_p = \ell'\gamma' = \ell\sigma\gamma'$, by fixing $x = \ell|_q$ for a position q, we obtain $\gamma(x) = s|_{pq} = \gamma'(\sigma(x))$. Thus, $\sigma(\gamma(x)) = \gamma(x) = \gamma'(\sigma(x))$ for all $x \in \mathcal{V}\mathrm{ar}(\ell)$.

Putting all of this together gives $\sigma(\gamma(x)) = \gamma'(\sigma(x))$ for all $x \in \mathcal{V}\mathrm{ar}(\rho)$. Hence, we have $r\gamma\sigma = r\sigma\gamma' = r'\gamma'$ and $\pi\gamma\sigma = \pi\sigma\gamma' = \pi'\gamma'$. Therefore, we have $t\sigma = s[r\gamma]_p\sigma = s[r\gamma\sigma]_p = s[r'\gamma']_p = t'$ and $\psi\sigma = (\varphi \wedge \pi\gamma)\sigma = \varphi \wedge \pi\gamma\sigma = \varphi \wedge \pi'\gamma' = \psi'$. In particular, it also follows that $\sigma(\mathcal{V}\mathrm{ar}(t)) = \mathcal{V}\mathrm{ar}(t\sigma) = \mathcal{V}\mathrm{ar}(t')$ and $\sigma(\mathcal{V}\mathrm{ar}(\psi)) = \mathcal{V}\mathrm{ar}(\psi\sigma) = \mathcal{V}\mathrm{ar}(\psi')$. Thus, as σ is bijective on $\mathcal{V}\mathrm{ar}(\rho)$, we obtain $\sigma(\{\vec{y}\}) = \sigma(\mathcal{V}\mathrm{ar}(\psi) \setminus \mathcal{V}\mathrm{ar}(t)) = \sigma(\{x \in \mathcal{V}\mathrm{ar}(\rho) \mid x \in \mathcal{V}\mathrm{ar}(\psi) \wedge x \notin \mathcal{V}\mathrm{ar}(t)\} = \{\sigma(x) \in \mathcal{V}\mathrm{ar}(\rho) \mid x \in \mathcal{V}\mathrm{ar}(\psi) \wedge x \notin \mathcal{V}\mathrm{ar}(t)\} = \{\sigma(x) \in \mathcal{V}\mathrm{ar}(\rho) \mid \sigma(x) \in \mathcal{V}\mathrm{ar}(\psi') \wedge \sigma(x) \notin \mathcal{V}\mathrm{ar}(t')\} = \mathcal{V}\mathrm{ar}(\psi') \setminus \mathcal{V}\mathrm{ar}(t') = \{\vec{y}'\}$. From $X \cap \mathcal{V}\mathrm{ar}(\rho) = \emptyset$, we know that $X \cap \mathcal{D}\mathrm{om}(\sigma) = \emptyset$. Thus, $\sigma(X) = X$. Therefore, $\sigma(Y) = \sigma(\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t))) = \sigma((\mathcal{V}\mathrm{ar}(r) \setminus \mathcal{V}\mathrm{ar}(\ell)) \cup (X \cap \mathcal{V}\mathrm{ar}(t))) = (\mathcal{V}\mathrm{ar}(r\sigma) \setminus \mathcal{V}\mathrm{ar}(\ell\sigma)) \cup (X \cap \mathcal{V}\mathrm{ar}(t\sigma)) = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho') \cup (X \cap \mathcal{V}\mathrm{ar}(t')) = Y'$. Since σ is a bijective renaming by Proposition 2.1 we have $\Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,] \sim \Pi Y'.\ t'\ [\,\exists\vec{y}'.\ \psi'\,]$. □

Uniqueness of reducts implies that for LCTRSs with a finite number of constrained rewrite rules, there exists only a finite number of reducts for any existentially constrained term (modulo equivalence). This stands in sharp contrast to the original notion of rewriting non-quantified terms and represents a useful property to check convergence of constrained terms.

Theorem 6.2 (Commutation of Rewrite Steps and Equivalence for Pattern-General Terms). *Let ρ be a left-linear constrained rewrite rule and $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,], \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ be existentially constrained terms that are satisfiable and pattern-general. If $\Pi X'.\ s'\ [\,\exists\vec{x}'.\ \varphi'\,]\ {}_\rho\!\leftarrow\ \Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] \sim \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ then $\Pi X'.\ s'\ [\,\exists\vec{x}'.\ \varphi'\,] \sim \Pi Y'.\ t'\ [\,\exists\vec{y}'.\ \psi'\,]\ {}_\rho\!\leftarrow\ \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ for some $\Pi Y'.\ t'\ [\,\exists\vec{y}'.\ \psi'\,]$ (see below).*

$$\begin{array}{ccc} \Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,] & \wedge\!\!\wedge & \Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] \\ \vdots & & \downarrow \\ \Pi Y'.\ t'\ [\,\exists\vec{y}'.\ \psi'\,] & \cdots & \Pi X'.\ s'\ [\,\exists\vec{x}'.\ \varphi'\,] \end{array}$$

Proof. From Proposition 2.2 it follows that there is a renaming $\sigma\colon \mathcal{V}\mathrm{ar}(s,\varphi) \to \mathcal{V}\mathrm{ar}(t,\psi)$ such that (1) $s\sigma = t$, (2) $Y = \sigma(X)$, and (3) $\vDash_{\mathcal{M}} ((\exists\vec{x}.\ \varphi)\sigma \Leftrightarrow (\exists\vec{y}.\ \psi))$.

Suppose that $\Pi X.\ s\ [\,\exists\vec{x}.\ \varphi\,] \to^p_{\rho,\gamma} \Pi X'.\ s'\ [\,\exists\vec{x}'.\ \varphi'\,]$, where $\rho\colon \Pi Z.\ \ell \to r\ [\,\pi\,]$ such that $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(s,\varphi) = \emptyset$. W.l.o.g. we assume $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(s,\varphi,t,\psi) = \emptyset$. By definition, we have (4) $\mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell)$, (5) $s|_p = \ell\gamma$, (6) $\gamma(x) \in \mathcal{V}\mathrm{al} \cup X$ for any $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$, and (7) $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (\exists\vec{z}.\ \pi\gamma)$, where $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$, $s' = s[r\gamma]$, $\varphi' = \varphi \wedge \pi\gamma$, $\{\vec{x}'\} = \mathcal{V}\mathrm{ar}(\varphi') \setminus \mathcal{V}\mathrm{ar}(s')$, and $X' = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(s'))$.

We show a rewrite step $\Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,] \to^p_{\rho,\delta} \Pi Y'.\ t'\ [\,\exists\vec{y}'.\ \psi'\,]$, where $\delta = \{x \mapsto \sigma(\gamma(x)) \mid x \in \mathcal{V}\mathrm{ar}(\ell)\}$. First we show that $\Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,]$ has a ρ-redex at p by δ, that is, (4') $\mathcal{D}\mathrm{om}(\delta) = \mathcal{V}\mathrm{ar}(\ell)$, (5') $t|_p = \ell\delta$, (6') $\delta(x) \in \mathcal{V}\mathrm{al} \cup Y$ for any $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$, and (7') $\vDash_{\mathcal{M}} (\exists\vec{y}.\ \psi) \Rightarrow (\exists\vec{z}.\ \pi\delta)$, where $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$.

(4') is satisfied by definition of δ. (5') follows as $\ell\delta = \ell\gamma\sigma = (s|_p)\sigma = (s\sigma)|_p = t|_p$, using (1) and (5). To show (6'), assume $x \in \mathcal{V}\mathrm{al}(\ell) \cap Z$. Then $\gamma(x) \in \mathcal{V}\mathrm{al} \cup X$ by (6), and $\delta(x) = \sigma(\gamma(x)) \in \mathcal{V}\mathrm{al} \cup \sigma(X) = \mathcal{V}\mathrm{al} \cup Y$, by using (2). It remains to show (7'). From (7) we have $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (\exists\vec{z}.\ \pi\gamma)$, thus $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi)\sigma \Rightarrow (\exists\vec{z}.\ \pi\gamma)\sigma$. From (3) follows that $\vDash_{\mathcal{M}} (\exists\vec{y}.\ \psi) \Rightarrow (\exists\vec{z}.\ \pi\gamma)\sigma$. By $\sigma\colon \mathcal{V}\mathrm{ar}(s,\varphi) \to \mathcal{V}\mathrm{ar}(t,\psi)$ and $\{\vec{z}\} \subseteq \mathcal{V}\mathrm{ar}(\rho)$, we conclude from $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(s,\varphi) = \emptyset$ that $\{\vec{z}\} \cap \mathcal{V}\mathrm{ar}(s,\varphi) = \emptyset$, and further from $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(t,\psi) = \emptyset$ that $\{\vec{z}\} \cap \mathcal{V}\mathrm{ar}(t,\psi) = \emptyset$. Hence, by definition of δ, $(\exists\vec{z}.\ \pi\gamma)\sigma = \exists\vec{z}.\ \pi\gamma\sigma = \exists\vec{z}.\ \pi\delta$, and therefore $\vDash_{\mathcal{M}} (\exists\vec{y}.\ \psi) \Rightarrow (\exists\vec{z}.\ \pi\delta)$ is satisfied. Let $t' = s[r\delta]_p$, $\psi' = \psi \wedge \pi\delta$, $\{\vec{y}'\} = \mathcal{V}\mathrm{ar}(\psi') \setminus \mathcal{V}\mathrm{ar}(t')$, and $Y' = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (Y \cap \mathcal{V}\mathrm{ar}(t'))$, we obtain the rewrite step $\Pi Y.\ t\ [\,\exists\vec{y}.\ \psi\,] \to^p_{\rho,\delta} \Pi Y'.\ t'\ [\,\exists\vec{y}'.\ \psi'\,]$.

It remains to show that $\Pi X'.\ s'\ [\,\exists\vec{x}'.\ \varphi'\,] \sim \Pi Y'.\ t'\ [\,\exists\vec{y}'.\ \psi'\,]$. To that end we use Proposition 2.3. We show (8) $\mathcal{P}\mathrm{os}_{X' \cup \mathcal{V}\mathrm{al}}(s') = \mathcal{P}\mathrm{os}_{Y' \cup \mathcal{V}\mathrm{al}}(t')$ $(= \{p_1, \ldots, p_n\})$, (9) $\rho(s'[\,]_{p_1,\ldots,p_n}) = t'[\,]_{p_1,\ldots,p_n}$ for some renaming $\rho\colon \mathcal{V}\mathrm{ar}(s') \setminus X' \to \mathcal{V}\mathrm{ar}(t') \setminus Y'$, (10) for any $i, j \in \{1, \ldots, n\}$, $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (s'|_{p_i} = s'|_{p_j})$ iff $\vDash_{\mathcal{M}} (\exists\vec{y}.\ \psi) \Rightarrow (t'|_{p_i} = t'|_{p_j})$, (11) for any $i \in \{1, \ldots, n\}$ and $v \in \mathcal{V}\mathrm{al}$, $\vDash_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (s'|_{p_i} = v)$ iff $\vDash_{\mathcal{M}} (\exists\vec{y}.\ \psi) \Rightarrow (t'|_{p_i} = v)$, and (12) let $\sim = {\sim_{\mathcal{P}\mathrm{os}_{X' \cup \mathcal{V}\mathrm{al}}(s')}} = {\sim_{\mathcal{P}\mathrm{os}_{X' \cup \mathcal{V}\mathrm{al}}(t')}}$ and $\mu_{X'}, \mu_{Y'}$ be representative substitutions of $\Pi X'.\ s'\ [\,\exists\vec{x}'.\ \varphi'\,]$ and $\Pi Y'.\ t'\ [\,\exists\vec{y}'.\ \psi'\,]$, respectively, based on the same representative for each equivalence class $[p_i]_\sim$ $(1 \leqslant i \leqslant n)$, and we have $\vDash_{\mathcal{M}} (\exists\vec{x}'.\ \varphi')\mu_{X'}\theta|_{\tilde{X}'} \Leftrightarrow (\exists\vec{y}'.\ \psi')\mu_{Y'}$ with a renaming $\theta|_{\tilde{X}'}\colon \tilde{X}' \to \tilde{Y}'$, where $\theta = \{\langle s'|_{p_i}, t'|_{p_i}\rangle \mid 1 \leqslant i \leqslant n\}$, $\tilde{X}' = \hat{X}' \cap X'$, and $\tilde{Y}' = \hat{Y}' \cap Y'$.

We start by showing (8). Consequently observe that

$$s'\sigma = s[r\gamma]_p\sigma = s\sigma[r\gamma\sigma]_p = t[r\delta]_p = t' \tag{13}$$

Hence, $\sigma(X') = \sigma(\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup \sigma(X \cap \mathcal{V}\mathrm{ar}(s')) = \sigma(\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho)) \cup \sigma(X \cap \mathcal{V}\mathrm{ar}(s')) = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (\sigma(X) \cap \mathcal{V}\mathrm{ar}(s'\sigma)) = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (Y \cap \mathcal{V}\mathrm{ar}(t')) = Y'$ follows. This implies that $\mathcal{P}\mathrm{os}_{X' \cup \mathcal{V}\mathrm{al}}(s') = \mathcal{P}\mathrm{os}_{\sigma(X') \cup \mathcal{V}\mathrm{al}}(s'\sigma) = \mathcal{P}\mathrm{os}_{Y' \cup \mathcal{V}\mathrm{al}}(t')$, which shows (8). In the following let $\mathcal{P}\mathrm{os}_{X' \cup \mathcal{V}\mathrm{al}}(s') = \{p_1, \ldots, p_n\}$.

We proceed to show (9). We let $s' = s'[s'_1, \ldots, s'_n]_{p_1,\ldots,p_n}$ and $t' = t'[t'_1, \ldots, t'_n]_{p_1,\ldots,p_n}$, using the fact (8). Then, from (13), it follows that $s'\sigma[\,]_{p_1,\ldots,p_n} = t'[\,]_{p_1,\ldots,p_n}$ and $s'_i\sigma = t'_i$ for each $1 \leqslant i \leqslant n$. Note here that $s'_i \in \mathcal{V}\mathrm{al} \cup X'$ and $t'_i \in \mathcal{V}\mathrm{al} \cup Y'$. Thus, since σ is a renaming, we have $s'_i \in X'$ iff $t'_i \in Y'$, $s'_i \in \mathcal{V}\mathrm{al}$ iff $t'_i \in \mathcal{V}\mathrm{al}$, and $s'_i = s'_j$ iff $t'_i = t'_j$ for $i, j \in \{1, \ldots, n\}$. Hence $\{s'_1, \ldots, s'_n\} \cap X' = \{t'_1, \ldots, t'_n\} \cap Y'$. Since $\sigma(X') = Y'$, it implies that $\eta = \{x \mapsto \sigma(x) \mid x \in \mathcal{V}\mathrm{ar}(s') \setminus X'\}$ is a bijection from $\mathcal{V}\mathrm{ar}(s') \setminus X'$ to $\mathcal{V}\mathrm{ar}(t') \setminus Y'$ such that $\eta(s'[\,]_{p_1,\ldots,p_n}) = t'[\,]_{p_1,\ldots,p_n}$.

Before we proceed to the remainder of the proof, let us introduce a useful temporary notation and show some related properties. For $\exists\vec{x}.\ \varphi$ and $\exists\vec{x}'.\ \varphi' = \exists\vec{x}'.\ \varphi \wedge \pi\gamma$, we have, by Lemma 3.9, that $\{\vec{x}\} \subseteq \{\vec{x}'\}$. Let $\{\vec{x}''\} = \{\vec{x}'\} \setminus \{\vec{x}\}$ and let us write $\varphi = \varphi(\vec{x}, \vec{x}'')$. Also, for $\pi\gamma$, because $\{\vec{x}\} \cap \mathcal{V}\mathrm{ar}(\pi\gamma) = \emptyset$ by Lemma 3.9, we have $\mathcal{V}\mathrm{ar}(\pi\gamma) \cap \{\vec{x}'\} = \{\vec{x}'''\} \subseteq \{\vec{x}''\}$. Thus, using the subsequence $\vec{x}'''$ of $\vec{x}''$, we write $\pi\gamma = \pi\gamma(\vec{x}''')$. Similarly, for $\exists\vec{y}.\ \psi$ and $\exists\vec{y}'.\ \psi' = \exists\vec{y}'.\ \psi \wedge \pi\delta$, let us write $\psi = \psi(\vec{y}, \vec{y}'')$ and $\pi\delta = \pi\delta(\vec{y}''')$ where $\{\vec{y}''\} = \{\vec{y}'\} \setminus \{\vec{y}\}$ and $\{\vec{y}'''\} \subseteq \{\vec{y}''\}$. Note that $\mathcal{V}\mathrm{ar}(\pi\gamma) = (\bigcup_{x \in \mathcal{V}\mathrm{ar}(\ell) \cap \mathcal{V}\mathrm{ar}(\pi)} \mathcal{V}\mathrm{ar}(x\gamma)) \cup (\mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell))$. Moreover, we obtain that $\sigma(\{\vec{x}'''\}) = \sigma(\mathcal{V}\mathrm{ar}(\pi\gamma) \cap \{\vec{x}'\}) = \sigma(((\bigcup_{x \in \mathcal{V}\mathrm{ar}(\ell) \cap \mathcal{V}\mathrm{ar}(\pi)} \mathcal{V}\mathrm{ar}(x\gamma)) \cup (\mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell))) \setminus \mathcal{V}\mathrm{ar}(s')) = ((\bigcup_{x \in \mathcal{V}\mathrm{ar}(\ell) \cap \mathcal{V}\mathrm{ar}(\pi)} \mathcal{V}\mathrm{ar}(x\delta)) \cup (\mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell))) \setminus \mathcal{V}\mathrm{ar}(t') = \mathcal{V}\mathrm{ar}(\pi\delta) \cap \{\vec{y}\} = \{\vec{y}'''\}$.

From (3) $\vDash_{\mathcal{M}} ((\exists\vec{x}.\ \varphi)\sigma \Leftrightarrow (\exists\vec{y}.\ \psi))$, we have that, for any valuation ξ, $\vDash_{\mathcal{M},\xi} \varphi\sigma(\vec{a}, \vec{x}''\sigma)$ for some $\vec{a} \in |\mathcal{M}|^*$ iff $\vDash_{\mathcal{M},\xi} \psi(\vec{b}, \vec{y}'')$

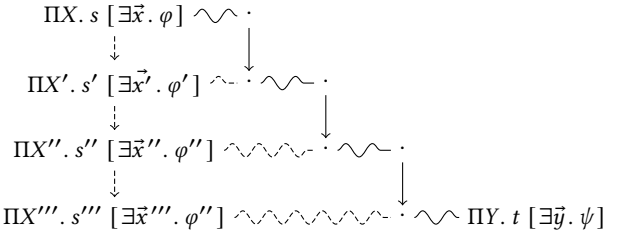
for some $\vec{b} \in |\mathcal{M}|^*$. Thus, for any valuation ξ, $\vDash_{\mathcal{M},\xi} \varphi\sigma(\vec{a}, \vec{x}''\sigma) \wedge \pi\gamma\sigma(\vec{x}'''\sigma)$ for some $\vec{a} \in |\mathcal{M}|^*$ iff $\vDash_{\mathcal{M},\xi} \psi(\vec{b}, \vec{y}'') \wedge \pi\delta(\vec{y}''')$ for some $\vec{b} \in |\mathcal{M}|^*$, as we know $\pi\gamma\sigma = \pi\delta$ and $\sigma(\{\vec{x}'''\}) = \{\vec{y}'''\}$. In particular, we have

$$\vDash_{\mathcal{M}} ((\exists \vec{x}'.\ \varphi')\sigma \Leftrightarrow (\exists \vec{y}'.\ \psi')) \tag{14}$$

Using (14) and our assumption, $\vDash_{\mathcal{M}} \exists x'.\ \varphi' \Rightarrow (s'|_{p_i} = s'|_{p_j})$ iff $\vDash_{\mathcal{M}} \exists x'.\ \varphi'\sigma \Rightarrow (\sigma(s'|_{p_i}) = \sigma(s'|_{p_j}))$ iff $\vDash_{\mathcal{M}} \exists y'.\ \psi' \Rightarrow (s'\sigma|_{p_i} = s'\sigma|_{p_j})$ iff $\vDash_{\mathcal{M}} \exists y'.\ \psi' \Rightarrow (t'|_{p_i}) = t'|_{p_j})$ for $i, j \in \{1, \ldots, n\}$. Similarly, $\vDash_{\mathcal{M}} \exists x'.\ \varphi' \Rightarrow (s'|_{p_i} = v)$ iff $\vDash_{\mathcal{M}} \exists x'.\ \varphi'\sigma \Rightarrow (\sigma(s'|_{p_i}) = v\sigma)$ iff $\vDash_{\mathcal{M}} \exists y'.\ \psi' \Rightarrow (s'\sigma|_{p_i} = v)$ iff $\vDash_{\mathcal{M}} \exists y'.\ \psi' \Rightarrow (t'|_{p_i} = v)$ for $i, j \in \{1, \ldots, n\}$. This shows (10) and (11).

Finally, take $\theta = \{\langle s'|_{p_i}, t'|_{p_i}\rangle \mid 1 \leqslant i \leqslant n\}$. As $\theta|_{\tilde{X}'} \subseteq \sigma$, from (10), (11) and $s'\sigma = t'$, it follows that $\theta|_{\tilde{X}'} \circ \mu_{X'} = \mu_{Y'} \circ \sigma$. From (14), we have $\vDash_{\mathcal{M}} (\exists \vec{x}'.\ \varphi')\sigma\mu_{Y'} \Leftrightarrow (\exists \vec{y}'.\ \psi')\mu_{Y'}$, and thus, $\vDash_{\mathcal{M}} (\exists \vec{x}'.\ \varphi')\mu_{X'}\theta|_{\tilde{X}'} \Leftrightarrow (\exists \vec{y}'.\ \psi')\mu_{Y'}$. As $\mu_{X'}, \mu_{Y'}$ are based on the same representative for each equivalence class $[p_i]_\sim$ $(1 \leqslant i \leqslant n)$, $\theta|_{\tilde{X}'}$ is a variable renaming from $\tilde{X}'$ to $\tilde{Y}'$. This shows (12) and completes the proof. □

In general, the commutation property allows us to postpone all equivalence translations, depicted by the following diagram:



Here, the rewrite steps $\Pi X.\ s\ [\exists \vec{x}.\ \varphi] \rightarrow^* \Pi Y.\ t\ [\exists \vec{y}.\ \psi]$ can be obtained via the rewrite steps $\Pi X.\ s\ [\exists \vec{x}.\ \varphi] \rightarrow \Pi X'.\ s'\ [\exists \vec{x}'.\ \varphi'] \rightarrow \Pi X''.\ s''\ [\exists \vec{x}''.\ \varphi''] \rightarrow \Pi X'''.\ s'''\ [\exists \vec{x}'''.\ \varphi'''] \sim \Pi Y.\ t\ [\exists \vec{y}.\ \psi]$ if repeated applications of commutation are possible. However, Theorem 6.2 does not allow repeated applications, because, in general, reducts of most general rewrite steps do not necessarily lead again to pattern-general constrained terms. Hence, its restriction to pattern-general terms prevents the possibility of repeated applications.

7 Simulation via Left-Value-Free Rules

In Theorem 6.2, we assume that the equivalent existentially constrained terms are pattern-general. A natural question is, what happens if we drop this condition?

Example 7.1. Let $\rho\colon \Pi\varnothing.\ \mathrm{f}(0) \to 0\ [\mathrm{true}]$ be a constrained rule. Consider the equivalence $\Pi\varnothing.\ \mathrm{f}(0) \sim \Pi\varnothing.\ \mathrm{f}(x)\ [x = 0]$ and a rewrite step $\Pi\varnothing.\ \mathrm{f}(0) \to_\rho \Pi\varnothing.\ 0$. However, $\Pi\varnothing.\ \mathrm{f}(x)\ [x = 0]$ cannot be rewritten by ρ, because there does not exist a γ such that $\mathrm{f}(x) = \mathrm{f}(0)\gamma$. Thus constrained terms that are not pattern-general, may not commute with rewrite steps in general.

However, we can recover commutation using a reasonable transformation on constrained rewrite rules. In this section, we introduce this transformation and show that any constrained rewrite rule can be simulated by using it.

The counterexample of commutation in Example 7.1 uses the rule $\rho\colon \Pi\varnothing.\ \mathrm{f}(0) \to 0\ [\mathrm{true}]$, which is not able to reduce the constrained term $\Pi\varnothing.\ \mathrm{f}(x)\ [x = 0]$. However, the rule $\rho'\colon \Pi\{x\}.\ \mathrm{f}(x) \to 0\ [x = 0]$ has the same effect as ρ, but can reduce the term as follows: $\Pi\varnothing.\ \mathrm{f}(x)\ [x = 0] \to_{\rho'} \Pi\varnothing.\ 0\ [\exists x.\ x = 0] \sim \Pi\varnothing.\ 0$. In the following, we show how this can be achieved for arbitrary constrained rewrite rules. We define left-value-free rewrite rules [6] in our setting as follows.

Definition 7.2 (Left-Value-Free Rewrite Rule). A left-linear constrained rewrite rule $\Pi Z.\ \ell \to r\ [\pi]$ is *left-value-free* if $\mathcal{V}\mathrm{al}(\ell) = \varnothing$.

Definition 7.3 (Left-Value-Free Transformation). Let $\rho\colon \Pi Z.\ \ell \to r\ [\pi]$ be a left-linear constrained rule, $\mathcal{P}\mathrm{os}_{\mathcal{V}\mathrm{al}}(\ell) = \{p_1, \ldots, p_n\}$, and $\ell|_{p_i} = v_i$ for $1 \leqslant i \leqslant n$. Taking the fresh variables $x_1, \ldots, x_n$, we define the *left-value-free transformation* of ρ as $\mathrm{lvf}(\rho) = \Pi\hat{Z}.\ \hat{\ell} \to r\ [\hat{\pi}]$ where $\hat{Z} = Z \cup \{x_1, \ldots, x_n\}$, $\hat{\ell} = \ell[x_1, \ldots, x_n]_{p_1,\ldots,p_n}$, and $\hat{\pi} = \pi \wedge \bigwedge_{i=1}^n (x_i = v_i)$.

We demonstrate that restricting rewrite rules to left-value-free ones preserves generality.

Theorem 7.4. *Let $\rho\colon \Pi Z.\ \ell \to r\ [\pi]$ be a left-linear constrained rewrite rule, $\mathrm{lvf}(\rho) = \hat{\rho}\colon \Pi\hat{Z}.\ \hat{\ell} \to r\ [\hat{\pi}]$, and $\Pi X.\ s\ [\exists \vec{x}.\ \varphi]$ an existentially constrained term such that $\mathcal{V}\mathrm{ar}(\hat{\rho}) \cap \mathcal{V}\mathrm{ar}(s, \varphi) = \varnothing$. If $\Pi X.\ s\ [\exists \vec{x}.\ \varphi] \to^p_\rho \Pi Y.\ t\ [\exists \vec{y}.\ \psi]$ then there exists a constrained term $\Pi Y'.\ t'\ [\exists \vec{y}'.\ \psi']$ such that $\Pi X.\ s\ [\exists \vec{x}.\ \varphi] \to^p_{\hat{\rho}} \Pi Y'.\ t'\ [\exists \vec{y}'.\ \psi'] \sim \Pi Y.\ t\ [\exists \vec{y}.\ \psi]$.*

Proof. Suppose $\ell = \ell[v_1, \ldots, v_n]_{p_1,\ldots,p_n}$, where $\{p_1, \ldots, p_n\} = \mathcal{P}\mathrm{os}_{\mathcal{V}\mathrm{al}}(\ell)$. Let $y_1, \ldots, y_n$ are fresh variables, $\hat{Y} = \{y_1, \ldots, y_n\}$, $\hat{Z} = Z \cup \hat{Y}$, $\hat{\ell} = \ell[y_1, \ldots, y_n]_{p_1,\ldots,p_n}$, and $\hat{\pi} = \pi \wedge \bigwedge_{i=1}^n (y_i = v_i)$. We obtain $\mathcal{V}\mathrm{ar}(\hat{\ell}) \cap \hat{Z} = (\mathcal{V}\mathrm{ar}(\ell) \cup \hat{Y}) \cap (Z \cup \hat{Y}) = (\mathcal{V}\mathrm{ar}(\ell) \cap Z) \cup \hat{Y}$ and $\mathcal{V}\mathrm{ar}(\hat{\pi}) \setminus \mathcal{V}\mathrm{ar}(\hat{\ell}) = (\mathcal{V}\mathrm{ar}(\pi) \cup \hat{Y}) \setminus (\mathcal{V}\mathrm{ar}(\ell) \cup \hat{Y}) = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$. Suppose $\Pi X.\ s\ [\exists \vec{x}.\ \varphi] \to^p_{\rho,\gamma} \Pi Y.\ t\ [\exists \vec{y}.\ \psi]$. Then we have (1) $\mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell)$, (2) $s|_p = \ell\gamma$, (3) $\gamma(x) \in \mathcal{V}\mathrm{al} \cup X$ for any $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$, and (4) $\vDash_{\mathcal{M}} (\exists \vec{x}.\ \varphi) \Rightarrow (\exists \vec{z}.\ \pi\gamma)$, where $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$, $t = s[r\gamma]$, $\psi = \varphi \wedge \pi\gamma$, $\{\vec{y}\} = \mathcal{V}\mathrm{ar}(\psi) \setminus \mathcal{V}\mathrm{ar}(t)$, and $Y = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t))$.

Let $\delta = \gamma \cup \{y_i \mapsto v_i \mid 1 \leqslant i \leqslant n\}$. We first show that the term $\Pi X.\ s\ [\exists \vec{x}.\ \varphi]$ has a $\hat{\rho}$-redex at $p \in \mathcal{P}\mathrm{os}(s)$ using δ. By (1), we have $\mathcal{D}\mathrm{om}(\delta) = \mathcal{D}\mathrm{om}(\gamma) \cup \hat{Y} = \mathcal{V}\mathrm{ar}(\ell) \cup \hat{Y}$. From (2), we have $s|_p = \ell\gamma = \ell[v_1, \ldots, v_n]\gamma = \ell\gamma[v_1, \ldots, v_n] = \ell\delta[\delta(y_1), \ldots, \delta(y_n)] = \ell[y_1, \ldots, y_n]\delta = \hat{\ell}\delta$. We proceed to show that $\delta(x) \in \mathcal{V}\mathrm{al} \cup X$ for any $x \in \mathcal{V}\mathrm{ar}(\hat{\ell}) \cap \hat{Z}$. Let $x \in \mathcal{V}\mathrm{ar}(\hat{\ell}) \cap \hat{Z} = (\mathcal{V}\mathrm{ar}(\ell) \cap Z) \cup \{y_1, \ldots, y_n\}$. If $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$, then $\delta(x) = \gamma(x) \in \mathcal{V}\mathrm{al} \cup X$. Otherwise, $x \in \hat{Y}$, and thus $\delta(x) = \delta(y_i) = v_i \in \mathcal{V}\mathrm{al} \subseteq \mathcal{V}\mathrm{al} \cup X$ for some $1 \leqslant i \leqslant n$. Using the fact that $\mathcal{V}\mathrm{ar}(\hat{\pi}) \setminus \mathcal{V}\mathrm{ar}(\hat{\ell}) = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$, we show that $\vDash_{\mathcal{M}} (\exists \vec{x}.\ \varphi) \Rightarrow (\exists \vec{z}.\ \hat{\pi}\delta)$. Here we have $\hat{\pi}\gamma = (\pi \wedge \bigwedge_{i=1}^n (y_i = v_i))\gamma = \pi\gamma \wedge \bigwedge_{i=1}^n (v_i = v_i) = \pi\delta \wedge \bigwedge_{i=1}^n (v_i = v_i)$. Clearly $\vDash_{\mathcal{M}} \pi\gamma \Leftrightarrow \hat{\pi}\delta$. Hence, from (4), we have $\vDash_{\mathcal{M}} (\exists \vec{x}.\ \varphi) \Rightarrow (\exists \vec{z}.\ \hat{\pi}\delta)$. We conclude that $\Pi X.\ s\ [\exists \vec{x}.\ \varphi]$ has a $\hat{\rho}$-redex at $p \in \mathcal{P}\mathrm{os}(s)$ using δ.

Hence, we obtain $\Pi X.\ s\ [\exists \vec{x}.\ \varphi] \to^p_{\hat{\rho},\delta} \Pi Y'.\ t'\ [\exists \vec{y}'.\ \psi']$, where $t' = s[r\delta]$, $\psi' = \varphi \wedge \hat{\pi}\delta$, $\{\vec{y}'\} = \mathcal{V}\mathrm{ar}(\psi') \setminus \mathcal{V}\mathrm{ar}(t')$, and $Y' = \mathcal{E}x\mathcal{V}\mathrm{ar}(\hat{\rho}) \cup (X \cap \mathcal{V}\mathrm{ar}(t'))$. From $\mathcal{E}x\mathcal{V}\mathrm{ar}(\hat{\rho}) \cap \hat{Y} = \varnothing$, we have $r\delta = r\gamma$ and $\pi\delta = \pi\gamma$. Thus, (5) $t' = s[r\delta] = s[r\gamma] = t$. It also follows from $\psi = \varphi \wedge \pi\gamma$, $\psi' = \varphi \wedge \hat{\pi}\delta$, and $\vDash_{\mathcal{M}} \pi\gamma \Leftrightarrow \hat{\pi}\delta$ (shown above), that (6)

$\models_{\mathcal{M}} \psi' \Leftrightarrow \psi$. Using (5) and $\mathcal{V}\mathrm{ar}(\hat{\pi}\delta) = \mathcal{V}\mathrm{ar}(\pi\delta \wedge \bigwedge_{i=1}^{n}(v_i = v_i)) = \mathcal{V}\mathrm{ar}(\pi\gamma)$, we obtain that, (7) $\{\vec{y}'\} = \mathcal{V}\mathrm{ar}(\psi') \setminus \mathcal{V}\mathrm{ar}(t') = \mathcal{V}\mathrm{ar}(\varphi \wedge \hat{\pi}\delta) \setminus \mathcal{V}\mathrm{ar}(t') = \mathcal{V}\mathrm{ar}(\varphi \wedge \pi\gamma) \setminus \mathcal{V}\mathrm{ar}(t) = \mathcal{V}\mathrm{ar}(\psi) \setminus \mathcal{V}\mathrm{ar}(t) = \{\vec{y}\}$. By $\mathcal{E}x\mathcal{V}\mathrm{ar}(\hat{\rho}) \cap \hat{Y} = \varnothing$, we have $\mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) = \mathcal{E}x\mathcal{V}\mathrm{ar}(\hat{\rho})$. (8) $Y' = \mathcal{E}x\mathcal{V}\mathrm{ar}(\hat{\rho}) \cup (X \cap \mathcal{V}\mathrm{ar}(t')) = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t)) = Y$. Having (5)–(8), it is easy to check from the definition of equivalence that $\Pi Y'.\ t'\ [\exists\vec{y}.\ \psi']$ $\sim$ $\Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ holds. □

Example 7.5. Let us revisit Example 7.1. There we consider the constrained rewrite rule $\rho\colon \Pi\varnothing.\ \mathrm{f}(0) \to 0\ [\mathrm{true}]$ and the equivalence $\Pi\varnothing.\ \mathrm{f}(0) \sim \Pi\varnothing.\ \mathrm{f}(x)\ [x = 0]$. Let $\hat{\rho} = \mathrm{lvf}(\rho) = \Pi\{y\}.\ \mathrm{f}(y) \to 0\ [\mathrm{true} \wedge y = 0]$. Using the rule $\hat{\rho}$, instead of ρ, yields $\Pi\varnothing.\ \mathrm{f}(0) \to_{\hat{\rho}} \Pi\varnothing.\ 0\ [\mathrm{true} \wedge 0 = 0]$ and $\Pi\varnothing.\ \mathrm{f}(x)\ [x = 0] \to_{\hat{\rho}} \Pi\varnothing.\ 0\ [\exists x.\ \mathrm{true} \wedge (x = 0) \wedge (x = 0)]$. In this case, clearly, $\Pi\varnothing.\ 0\ [\mathrm{true} \wedge 0 = 0] \sim \Pi\varnothing.\ 0\ [\exists x.\ \mathrm{true} \wedge (x = 0) \wedge (x = 0)]$.

In the next section we show that, if we only consider left-value-free constrained rewrite rules, then the commutation property holds for all existentially constrained terms that are not necessarily pattern-general. Considering left-value-free rewrite rules does not weaken the applicability of the commutation property, because every constrained rewrite rule can be translated to a left-value-free rule, while preserving its effect, using Theorem 7.4.

8 General Commutation for Rewrite Steps with Left-Value-Free Rules and Equivalence

By focusing on pattern-general terms, we have shown in Theorem 6.2 that rewrite steps commute with the equivalence transformation. However, this commutation property inhabits issues w.r.t. to repeated applications as explained at the end of Section 6.

In the following we show that commutation holds without that restriction if the employed rule is left-value-free. Since any rule has an equivalent left-value-free rule, by Theorem 7.4, one can repeatedly apply the commutation property. This guarantees that the equivalence transformations can be postponed till the end of the rewrite sequence. We first introduce a useful notation that is used in the following proofs.

Definition 8.1. Let X be a set of variables and s a term. Suppose $\{p_1, \ldots, p_n\} = \mathcal{P}\mathrm{os}_{X \cup \mathcal{V}\mathrm{al}}(s)$. We define $s^{\bullet_X} = s[x_1, \ldots, x_n]_{p_1,\ldots,p_n}$ for some pairwise distinct fresh variables $x_1, \ldots, x_n$. We denote by $u^{\bullet_X} = s^{\bullet_X}|_p$ each subterm $u = s|_p$ of s. Consider a linear term ℓ together with a substitution γ such that $s = \ell\gamma$ with $\mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell)$. We define a substitution $\gamma^{\bullet_X}$ by $\gamma^{\bullet_X}(x) = \gamma(x)^{\bullet_X}$. When no confusion arises, the superscript $\bullet_X$ will be abbreviated by $\bullet$.

We now develop several properties of $\bullet$-translation; all proofs are given in the full version of this paper [17].

Lemma 8.2. *Let X be a set of variables and s a term. Let $s^{\bullet_X} = s[x_1, \ldots, x_n]_{p_1,\ldots,p_n}$ and $\sigma = \{x_i \mapsto s|_{p_i} \mid 1 \leqslant i \leqslant n\}$. Then, $u^{\bullet_X}\sigma = u$ holds for any subterm u of s.*

Lemma 8.3. *Let X be a set of variables, s a term, and ℓ a linear term such that $s|_p = \ell\gamma$ with $\mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell)$ for a substitution γ. If ℓ is value-free then $s^{\bullet_X}|_p = \ell\gamma^{\bullet_X}$.*

Lemma 8.4. *Let X be a set of variables and s a term. Suppose that $\mathcal{P}\mathrm{os}_{X \cup \mathcal{V}\mathrm{al}}(s) = \{p_1, \ldots, p_n\}$ and $s^{\bullet_X} = s[x_1, \ldots, x_n]_{p_1,\ldots,p_n}$. Then, for any subterm u of s, we have $\models_{\mathcal{M}} (\bigwedge_{i=1}^{n}(s(p_i) = x_i)) \Rightarrow u = u^{\bullet_X}$.*

Below we use yet another specific characterization of equivalence given as follows, whose proof is given in the full version of this paper [17].

Lemma 8.5. *Let $\Pi X.\ s\ [\exists\vec{x}.\ \varphi], \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ be satisfiable existentially constrained terms. Suppose (1) $\sigma\colon V \to \mathcal{V} \cup \mathcal{V}\mathrm{al}$ with $V \subseteq X$ and (2) $s\sigma = t$, (3) for any $x, y \in V$, $\sigma(x) = \sigma(y)$ implies $\models_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow x = y$. (4) $X \setminus V = Y \setminus \sigma(V)$ and (5) $\sigma(V) \cap \mathcal{V} \subseteq Y$, (6) $\mathcal{FV}\mathrm{ar}((\exists\vec{x}.\ \varphi)\sigma) = \mathcal{FV}\mathrm{ar}(\exists\vec{y}.\ \psi)$ and $\models_{\mathcal{M}} (\exists\vec{x}.\ \varphi)\sigma \Leftrightarrow (\exists\vec{y}.\ \psi)$. Then $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$.*

We show a key lemma for the theorem that follows.

Lemma 8.6. *Let ρ be a left-value-free constrained rewrite rule. If $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \to_{\rho} \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ then $\mathrm{PG}(\Pi X.\ s\ [\exists\vec{x}.\ \varphi]) \to_{\rho} \Pi Y'.\ t'\ [\exists\vec{y}'.\ \psi'] \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ for some $\Pi Y'.\ t'\ [\exists\vec{y}'.\ \psi']$.*

Proof. Let $p \in \mathcal{P}\mathrm{os}(s)$ and $\rho\colon \Pi Z.\ \ell \to r\ [\pi]$ be a left-linear constrained rewrite rule such that $\mathcal{V}\mathrm{al}(\ell) = \varnothing$ and $\mathcal{V}\mathrm{ar}(\rho) \cap \mathcal{V}\mathrm{ar}(\varphi, s) = \varnothing$. Suppose $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \to^{p}_{\rho,\gamma} \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$. Then we have (1) $\mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell)$, (2) $s|_p = \ell\gamma$, (3) $\gamma(x) \in \mathcal{V}\mathrm{al} \cup X$ for any $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$, and (4) $\models_{\mathcal{M}} (\exists\vec{x}.\ \varphi) \Rightarrow (\exists\vec{z}.\ \pi\gamma)$, where $\{\vec{z}\} = \mathcal{V}\mathrm{ar}(\pi) \setminus \mathcal{V}\mathrm{ar}(\ell)$, $t = s[r\gamma]$, $\psi = \varphi \wedge \pi\gamma$, $\{\vec{y}\} = \mathcal{V}\mathrm{ar}(\psi) \setminus \mathcal{V}\mathrm{ar}(t)$, and $Y = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X \cap \mathcal{V}\mathrm{ar}(t))$.

Using the notation of Definition 8.1, define $\mathrm{PG}(\Pi X.\ s\ [\exists\vec{x}.\ \varphi]) = \Pi X'.\ s^{\bullet_X}\ [\exists\vec{x}'.\ \varphi']$, i.e. $s^{\bullet_X} = s[x_1, \ldots, x_n]_{p_1,\ldots,p_n}$ for set of positions $\{p_1, \ldots, p_n\} = \mathcal{P}\mathrm{os}_{X \cup \mathcal{V}\mathrm{al}}(s)$, some pairwise distinct fresh variables $x_1, \ldots, x_n$, $X' = \{x_1, \ldots, x_n\}$ and $\{\vec{x}'\} = \{\vec{x}\} \cup X$ and $\varphi' = (\varphi \wedge \bigwedge_{i=1}^{n}(s|_{p_i} = x_i))$.

We first show $\Pi X'.\ s^{\bullet}\ [\exists\vec{x}'.\ \varphi']$ has a ρ-redex at p by $\gamma^{\bullet}$. Clearly, (1') $\mathcal{D}\mathrm{om}(\gamma^{\bullet}) = \mathcal{D}\mathrm{om}(\gamma) = \mathcal{V}\mathrm{ar}(\ell)$. By Lemma 8.3, we have (2') $s^{\bullet}|_p = \ell\gamma^{\bullet}$. Suppose $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$. Then $\gamma(x) \in \mathcal{V}\mathrm{al} \cup X$, which implies $\gamma(x) = s|_{p_i}$ for some $1 \leqslant i \leqslant n$. Then, $\gamma^{\bullet}(x) = \gamma(x)^{\bullet} = (s|_{p_i})^{\bullet} = s^{\bullet}|_{p_i} = (s[x_1, \ldots, x_n]_{p_1,\ldots,p_n})|_{p_i} = x_i \in X' \subseteq \mathcal{V}\mathrm{al} \cup X'$. Thus, (3') $\gamma^{\bullet}(x) \in \mathcal{V}\mathrm{al} \cup X'$ for any $x \in \mathcal{V}\mathrm{ar}(\ell) \cap Z$ follows. It remains to show (4') $\models_{\mathcal{M}} (\exists\vec{x}'.\ \varphi') \Rightarrow (\exists\vec{z}.\ \pi\gamma^{\bullet})$.

For this, let $\models_{\mathcal{M},\xi} \exists\vec{x}'.\ \varphi'$ for a valuation ξ. This means, $\models_{\mathcal{M},\xi} \exists\vec{x}, \vec{x}''.\ \varphi \wedge \bigwedge_{i=1}^{n}(s|_{p_i} = x_i))$, where $\{\vec{x}''\} = X$. By Lemma 8.4, $\models_{\mathcal{M}} \bigwedge_{i=1}^{n}(s(p_i) = x_i)) \Rightarrow (\gamma(x) = \gamma(x)^{\bullet})$ holds for any $x \in \mathcal{V}$. For, it trivially follows if $x \notin \mathcal{D}\mathrm{om}(\gamma)$, and otherwise, $\gamma(x)$ is a subterm of s, and thus Lemma 8.4 applies. Note that $s(p_i) = s|_{p_i}$ for all $1 \leqslant i \leqslant n$. This implies that $\models_{\mathcal{M},\xi} \exists\vec{x}, \vec{x}''.\ \varphi \wedge (\bigwedge_{i=1}^{n}(s|_{p_i} = x_i)) \wedge (\bigwedge_{x \in \mathcal{V}\mathrm{ar}(\pi)}(\gamma(x) = \gamma(x)^{\bullet}))$. Hence, we obtain $\models_{\mathcal{M},\xi} \exists\vec{x}, \vec{x}''.\ \varphi \wedge (\bigwedge_{i=1}^{n}(s|_{p_i} = x_i)) \wedge (\pi\gamma \Leftrightarrow \pi\gamma^{\bullet})$. Now, by definition, there exists a sequence $\vec{v} \in \mathcal{V}\mathrm{al}^*$ such that $\models_{\mathcal{M},\xi} (\exists\vec{x}.\ \varphi \wedge (\bigwedge_{i=1}^{n}(s|_{p_i} = x_i)) \wedge (\pi\gamma \Leftrightarrow \pi\gamma^{\bullet}))\kappa$, where $\kappa = \{\vec{x}'' \mapsto \vec{v}\}$. By $\mathcal{D}\mathrm{om}(\kappa) = \{\vec{x}''\} = X$, it follows that $\models_{\mathcal{M},\xi} \exists\vec{x}.\ \varphi\kappa \wedge (\bigwedge_{i=1}^{n}(s|_{p_i}\kappa = x_i)) \wedge (\pi\gamma\kappa = \pi\gamma^{\bullet})$. In particular, $\models_{\mathcal{M},\xi} \exists\vec{x}.\ \varphi\kappa$ holds and then by $\mathcal{D}\mathrm{om}(\kappa) \cap \{\vec{x}\} = \varnothing$, we have $\models_{\mathcal{M},\xi} (\exists\vec{x}.\ \varphi)\kappa$, i.e., $\models_{\mathcal{M},\xi\circ\kappa} \exists\vec{x}.\ \varphi$. Then, by (4), it follows $\models_{\mathcal{M},\xi\circ\kappa} \exists\vec{z}.\ \pi\gamma$, i.e., $\models_{\mathcal{M},\xi} (\exists\vec{z}.\ \pi\gamma)\kappa$. As $\{\vec{z}\} \cap X = \varnothing$, this implies $\models_{\mathcal{M},\xi} \exists\vec{z}.\ \pi\gamma\kappa$. Moreover, by $\{\vec{x}\} \cap \mathcal{V}\mathrm{ar}(\pi\gamma\kappa, \pi\gamma^{\bullet}) = \varnothing$, from $\models_{\mathcal{M},\xi} \exists\vec{x}.\ (\pi\gamma\kappa \Leftrightarrow \pi\gamma^{\bullet})$ it follows that $\models_{\mathcal{M},\xi} (\pi\gamma\kappa \Leftrightarrow \pi\gamma^{\bullet})$. Thus, we obtain $\models_{\mathcal{M},\xi} \exists\vec{z}.\ \pi\gamma^{\bullet}$ and have proven (4').

Hence, we obtain the rewrite step $\Pi X'.\ s^{\bullet}\ [\exists\vec{x}'.\ \varphi'] \to^{p}_{\rho,\gamma^{\bullet}} \Pi Y'.\ t'\ [\exists\vec{y}'.\ \psi']$, where $t' = s^{\bullet}[r\gamma^{\bullet}]$, $\psi' = \varphi' \wedge \pi\gamma^{\bullet}$, $\{\vec{y}'\} = \mathcal{V}\mathrm{ar}(\psi') \setminus \mathcal{V}\mathrm{ar}(t')$, and $Y' = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X' \cap \mathcal{V}\mathrm{ar}(t'))$.

It remains to show $\Pi Y'.\ t'\ [\exists\vec{y}'.\ \psi'] \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$. For this, we use Lemma 8.5 by taking $\sigma = \{x_i \mapsto s|_{p_i} \mid 1 \leqslant i \leqslant$

$n, x_i \in Y'\}$ and $V = \{x_i \mid 1 \leqslant i \leqslant n, x_i \in Y'\}$. We refer to the conditions (1)–(6) of Lemma 8.5 by (C1)–(C6). Clearly, (C1) $\sigma\colon V \to \mathcal{V} \cup \mathcal{V}\mathrm{al}$ with $V \subseteq Y'$. (C2) holds as $t'\sigma = s^\bullet[r\gamma^\bullet]\sigma = s^\bullet\sigma[r\gamma^\bullet\sigma] = s[r\gamma] = t$, using Lemma 8.2. Let $x_i, x_j \in Y'$ and $\sigma(x_i) = \sigma(x_j)$. Then, $s|_{p_i} = s|_{p_j}$ and hence $\vDash_{\mathcal{M}} \psi' \Rightarrow (s|_{p_i} = s|_{p_j})$. By $x_i, x_j \in Y'$, we know that $x_i, x_j \notin \{\vec{y}'\}$, and hence it follows $\vDash_{\mathcal{M}} \exists\vec{y}'.\ \psi' \Rightarrow (x_i = x_j)$. From this we have (C3). Note that $Y' = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (X' \cap \mathcal{V}\mathrm{ar}(t')) = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (\{x_1, \ldots, x_n\} \cap \mathcal{V}\mathrm{ar}(t'))$ and hence $V = \{x_1, \ldots, x_n\} \cap \mathcal{V}\mathrm{ar}(t')$ and that $Y = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \cup (\{s|_{p_1}, \ldots, s|_{p_n}\} \cap X \cap \mathcal{V}\mathrm{ar}(t))$. Since $Y' \backslash V = \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) = Y \backslash \sigma(V)$, we have (C4). For (C5), let $x \in Y'$ such that $\sigma(x) \in \mathcal{V}$. If $x \in V$ then $\sigma(x) \in Y$. Otherwise $\sigma(x) = x$, and by $s\sigma = t$, we know $x \in \mathcal{V}\mathrm{ar}(t)$. Hence $x \in \mathcal{E}x\mathcal{V}\mathrm{ar}(\rho) \subseteq Y$. Thus, (C5) holds. In order to show (C6), note first that $\mathcal{FV}\mathrm{ar}(\exists\vec{y}'.\ \psi') = \mathcal{V}\mathrm{ar}(\psi') \cap \mathcal{V}\mathrm{ar}(t')$ and $\mathcal{FV}\mathrm{ar}(\exists\vec{y}.\ \psi) = \mathcal{V}\mathrm{ar}(\psi) \cap \mathcal{V}\mathrm{ar}(t)$. Note that $\psi'\sigma = (\varphi' \wedge \pi\gamma^\bullet)\sigma = (\varphi \wedge \bigwedge_i (s|_{p_i} = x_i) \wedge \pi\gamma^\bullet)\sigma = (\varphi \wedge \bigwedge_i (s|_{p_i} = s|_{p_i}) \wedge \pi\gamma)$ and $\psi = (\varphi \wedge \pi\gamma)$, and hence $\mathcal{V}\mathrm{ar}(\psi'\sigma) = \mathcal{V}\mathrm{ar}(\psi)$. By $t'\sigma = t$ it follows $\mathcal{FV}\mathrm{ar}((\exists\vec{y}'.\ \psi')\sigma) = \sigma(\mathcal{FV}\mathrm{ar}(\exists\vec{y}'.\ \psi')) = \mathcal{FV}\mathrm{ar}(\exists\vec{y}.\ \psi)$. Finally,

$$
\begin{aligned}
&\vDash_{\mathcal{M}} (\exists\vec{y}'.\ \psi')\sigma \\
&\iff \vDash_{\mathcal{M}} (\exists\vec{y}'.\ (\varphi \wedge (\textstyle\bigwedge_{i=1}^{n}(s|_{p_i} = x_i))) \wedge \pi\gamma^\bullet)\sigma \\
&\iff \vDash_{\mathcal{M}} \exists\vec{y}'.\ (\varphi \wedge (\textstyle\bigwedge_{i=1}^{n}(s|_{p_i} = x_i\sigma))) \wedge \pi\gamma^\bullet\sigma \\
&\iff \vDash_{\mathcal{M}} \exists\vec{y}'.\ (\varphi \wedge (\textstyle\bigwedge_{i=1}^{n}(s|_{p_i} = s|_{p_i})) \wedge \pi\gamma \\
&\iff \vDash_{\mathcal{M}} \exists\vec{y}'.\ (\psi \wedge (\textstyle\bigwedge_{i=1}^{n}(s|_{p_i} = s|_{p_i})) \\
&\iff \vDash_{\mathcal{M}} \exists\vec{y}'.\ \psi
\end{aligned}
$$

Thus, we have $\vDash_{\mathcal{M}} (\exists\vec{y}'.\ \psi')\sigma \Leftrightarrow (\exists\vec{y}'.\ \psi)$. Since we can eliminate bound variables which do not appear in the constraint, we conclude $\vDash_{\mathcal{M}} (\exists\vec{y}'.\ \psi')\sigma \Leftrightarrow (\exists\vec{y}.\ \psi)$. □

Finally, we are able to prove the main result.

Theorem 8.7 (Commutation of Rewrite Steps and Equivalence by Left-Value-Free Rules). *Let ρ be a left-value-free constrained rewrite rule, and $\Pi X.\ s\ [\exists\vec{x}.\ \varphi], \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ be satisfiable existentially constrained terms. If $\Pi X'.\ s'\ [\exists\vec{x}'.\ \varphi'] \leftarrow_\rho \Pi X.\ s\ [\exists\vec{x}.\ \varphi] \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$, then we have $\Pi X'.\ s'\ [\exists\vec{x}'.\ \varphi'] \sim \Pi Y'.\ t'\ [\exists\vec{y}'.\ \psi'] \leftarrow_\rho \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ for some $\Pi Y'.\ t'\ [\exists\vec{y}'.\ \psi']$.*

Proof. Suppose that $\Pi X'.\ s'\ [\exists\vec{x}'.\ \varphi'] \leftarrow_\rho \Pi X.\ s\ [\exists\vec{x}.\ \varphi] \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$. By Lemma 8.6, there exists an existentially constrained term $\Pi X''.\ s''\ [\exists\vec{x}''.\ \varphi'']$ such that $\Pi X'.\ s'\ [\exists\vec{x}'.\ \varphi'] \sim \Pi X''.\ s''\ [\exists\vec{x}''.\ \varphi''] \leftarrow_\rho \mathrm{PG}(\Pi X.\ s\ [\exists\vec{x}.\ \varphi])$. By $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$, $\Pi X''.\ s''\ [\exists\vec{x}''.\ \varphi''] \leftarrow_\rho \mathrm{PG}(\Pi X.\ s\ [\exists\vec{x}.\ \varphi]) \sim \mathrm{PG}(\Pi Y.\ t\ [\exists\vec{y}.\ \psi])$. Hence, by Theorem 6.2, $\Pi X''.\ s''\ [\exists\vec{x}''.\ \varphi''] \sim \Pi Y'.\ t'\ [\exists\vec{y}'.\ \psi'] \leftarrow_\rho \mathrm{PG}(\Pi Y.\ t\ [\exists\vec{y}.\ \psi])$ for some existentially constrained term $\Pi Y'.\ t'\ [\exists\vec{y}'.\ \psi']$. Hence, $\Pi X'.\ s'\ [\exists\vec{x}'.\ \varphi'] \sim \Pi Y'.\ t'\ [\exists\vec{y}'.\ \psi'] \leftarrow_\rho \mathrm{PG}(\Pi Y.\ t\ [\exists\vec{y}.\ \psi])$. □

Theorem 8.7 enables us to defer the equivalence transformations until after the application of rewrite rules.

Corollary 8.8. *Let $\mathcal{R}$ be an LCTRS consisting of left-value-free constrained rewrite rules. If $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \mathrel{\tilde{\to}}_{\mathcal{R}} \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$, then $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \to^*_{\mathcal{R}} \Pi X'.\ s'\ [\exists\vec{x}'.\ \varphi'] \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$ for some $\Pi X'.\ s'\ [\exists\vec{x}'.\ \varphi']$.*

Proof Sketch. The claim can be proved by induction on n to show that if we have $\Pi X.\ s\ [\exists\vec{x}.\ \varphi](\sim \cdot \to_{\mathcal{R}})^n \cdot \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$, then there exists $\Pi X'.\ s'\ [\exists\vec{x}'.\ \varphi']$ such that $\Pi X.\ s\ [\exists\vec{x}.\ \varphi] \to^*_{\mathcal{R}} \Pi X'.\ s'\ [\exists\vec{x}'.\ \varphi'] \sim \Pi Y.\ t\ [\exists\vec{y}.\ \psi]$. □

9 Related Work

As mentioned earlier, tools working with LCTRSs, like Ctrl or crest, do already implement (mitigated) versions of the results in this paper. Implementation details w.r.t. crest are discussed in [13, 14]. This includes transformations like [13, Definition 12] that produces left-value-free rules, or to check joinability of $s \approx t\ [\varphi]$ crest does not compute any intermediate equivalences but checks triviality by using [13, Lemma 3] at the end of the rewrite sequence. Ctrl [9] makes constrained rewrite rules left-value-free in advance [3, 6], and thus, equivalence transformations are not used in applying constrained rewrite rules to constrained terms. Crisys [4] does not modify the given LCTRSs but uses an extension of matching during rewriting constrained terms, which can be seen as an alternative to equivalence transformations before rewrite steps: When matching a term ℓ with s in a constrained term $s\ [\varphi]$, a value v can match a logical variable x if $x = v$ is guaranteed by φ, i.e., $\vDash_{\mathcal{M}} \varphi \Rightarrow (x = v)$. However, the results of this paper do show promising evidence that tools incorporating these approximations do not loose much power.

Regarding rewriting of constrained terms, the *symbolic rewriting module SMT* has been proposed in [12]. The rewriting formalism there is different from constrained rewriting in this paper. We use the original definition of LCTRSs and constrained rewriting on them with the only difference on the presentation to ease its analysis. In particular, constrained rewriting reduces a constrained term to another one such that every instance in the former has its reduct in the latter. On the other hand, symbolic rewriting in [12] does not guarantee this, hence this is similar to narrowing of constrained terms.

10 Conclusion

In this paper, we have revisited the formalism of constrained rewriting in LCTRSs. We have introduced the new notion of most general rewriting on existentially constrained terms for left-linear LCTRSs. It was discussed in which way our new formalism of most general rewriting extracts the so called "most general" part of the corresponding original formalism. We have shown the uniqueness of reducts for our formalism of constrained rewriting, and commutation between rewrite steps and equivalence for pattern-general constrained terms. Then, by using the notion of non-left-value-free rewrite rules we showed that left-value-free rewrite rules can simulate non-left-value-free rules. Finally, we did recover general commutation between rewrite steps with left-value-free rules and equivalent transformations.

Because of the expected complications, our focus in this paper was on rewriting with left-linear rules. To deal with non-left-linear rules, we need to extend the matching mechanism underlying rewrite steps. However, this complicates the definition of rewrite steps drastically. This extension remains as future work.

Acknowledgments

This work was partially supported by FWF (Austrian Science Fund) project I 5943-N and JSPS KAKENHI Grant Numbers JP24K02900 and JP24K14817.

References

[1] Takahito Aoto, Naoki Nishida, and Jonas Schöpf. 2024. Equational Theories and Validity for Logically Constrained Term Rewriting. In *Proceedings of the 9th FSCD (LIPIcs, Vol. 299)*, Jakob Rehof (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 31:1–31:21. doi:10.4230/LIPIcs.FSCD.2024.31

[2] Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press. doi:10.1145/505863.505888

[3] Carsten Fuhs, Cynthia Kop, and Naoki Nishida. 2017. Verifying Procedural Programs via Constrained Rewriting Induction. *ACM Transactions on Computational Logic* 18, 2 (2017), 14:1–14:50. doi:10.1145/3060143

[4] Misaki Kojima, Naoki Nishida, and Yutaka Matsubara. 2025. Transforming concurrent programs with semaphores into logically constrained term rewrite systems. *Journal of Logical and Algebraic Methods in Programming* 143 (2025), 1–23. doi:10.1016/j.jlamp.2024.101033

[5] Cynthia Kop. 2016. Termination of LCTRSs. *CoRR* abs/1601.03206 (2016), 1–5. doi:10.48550/ARXIV.1601.03206

[6] Cynthia Kop. 2017. Quasi-reductivity of Logically Constrained Term Rewriting Systems. *CoRR* abs/1702.02397 (2017), 1–8. doi:10.48550/arXiv.1702.02397

[7] Cynthia Kop and Naoki Nishida. 2013. Term Rewriting with Logical Constraints. In *Proceedings of the 9th FroCoS (Lecture Notes in Computer Science, Vol. 8152)*, Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 343–358. doi:10.1007/978-3-642-40885-4_24

[8] Cynthia Kop and Naoki Nishida. 2014. Automatic Constrained Rewriting Induction towards Verifying Procedural Programs. In *Proceedings of the 12th APLAS (Lecture Notes in Computer Science, Vol. 8858)*, Jacques Garrigue (Ed.). Springer International Publishing, Cham, 334–353. doi:10.1007/978-3-319-12736-1_18

[9] Cynthia Kop and Naoki Nishida. 2015. ConsTrained Rewriting tooL. In *Proceedings of the 20th LPAR (Lecture Notes in Computer Science, Vol. 9450)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 549–557. doi:10.1007/978-3-662-48899-7_38

[10] Naoki Nishida and Sarah Winkler. 2018. Loop Detection by Logically Constrained Term Rewriting. In *Proceedings of the 10th VSTTE (Lecture Notes in Computer Science, Vol. 11294)*, Ruzica Piskac and Philipp Rümmer (Eds.). Springer International Publishing, Cham, 309–321. doi:10.1007/978-3-030-03592-1_18

[11] Enno Ohlebusch. 2002. *Advanced Topics in Term Rewriting.* Springer New York, New York. doi:10.1007/978-1-4757-3661-8

[12] Camilo Rocha, José Meseguer, and César A. Muñoz. 2017. Rewriting modulo SMT and open system analysis. *Journal of Logic and Algebraic Programming* 86, 1 (2017), 269–297. doi:10.1016/J.JLAMP.2016.10.001

[13] Jonas Schöpf and Aart Middeldorp. 2023. Confluence Criteria for Logically Constrained Rewrite Systems. In *Proceedings of the 29th CADE (Lecture Notes in Artificial Intelligence, Vol. 14132)*, Brigitte Pientka and Cesare Tinelli (Eds.). Springer Nature Switzerland, Cham, 474–490. doi:10.1007/978-3-031-38499-8_27

[14] Jonas Schöpf and Aart Middeldorp. 2025. Automated Analysis of Logically Constrained Rewrite Systems using crest. In *Proceedings of the 31st TACAS (Lecture Notes in Computer Science, Vol. 15696)*. Springer Nature Switzerland, Cham, 124–144. doi:10.1007/978-3-031-90643-5_7

[15] Jonas Schöpf, Fabian Mitterwallner, and Aart Middeldorp. 2024. Confluence of Logically Constrained Rewrite Systems Revisited. In *Proceedings of the 12th IJCAR (Lecture Notes in Artificial Intelligence, Vol. 14740)*, Christoph Benzmüller, Marijn J.H. Heule, and Renate A. Schmidt (Eds.). Springer Nature Switzerland, Cham, 298–316. doi:10.1007/978-3-031-63501-4_16

[16] Kanta Takahata, Jonas Schöpf, Naoki Nishida, and Takahito Aoto. 2025. Characterizing Equivalence of Logically Constrained Terms via Existentially Constrained Terms. In *Proceedings of the 35th International Symposium on Logic-Based Program Synthesis and Transformation (Lecture Notes in Computer Science, Vol. aa)*, Santiago Escobar and Laura Titolo (Eds.). Springer. to appear.

[17] Kanta Takahata, Jonas Schöpf, Naoki Nishida, and Takahito Aoto. 2025. Recovering Commutation of Logically Constrained Rewriting and Equivalence Transformations (Full Version). *CoRR* abs/2507.09326 (2025), 1–15. doi:10.48550/arXiv.2507.09326

[18] Sarah Winkler and Aart Middeldorp. 2018. Completion for Logically Constrained Rewriting. In *Proceedings of the 3rd FSCD (LIPIcs, Vol. 108)*, Hélène Kirchner (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:18. doi:10.4230/LIPIcs.FSCD.2018.30

[19] Sarah Winkler and Georg Moser. 2021. Runtime Complexity Analysis of Logically Constrained Rewriting. In *Proceedings of the 30th LOPSTR (Lecture Notes in Computer Science, Vol. 12561)*, Maribel Fernández (Ed.). Springer International Publishing, Cham, 37–55. doi:10.1007/978-3-030-68446-4_2

Dependency Pairs for Expected Innermost Runtime Complexity and Strong Almost-Sure Termination of Probabilistic Term Rewriting

Jan-Christoph Kassing
kassing@cs.rwth-aachen.de
RWTH Aachen University
Aachen, Germany

Leon Spitzer
leon.spitzer@rwth-aachen.de
RWTH Aachen University
Aachen, Germany

Jürgen Giesl
giesl@informatik.rwth-aachen.de
RWTH Aachen University
Aachen, Germany

Abstract

The dependency pair (DP) framework is one of the most powerful techniques for automatic termination and complexity analysis of term rewrite systems. While DPs were extended to prove almost-sure termination of *probabilistic* term rewrite systems (PTRSs), automatic complexity analysis for PTRSs is largely unexplored. We introduce the first DP framework for analyzing expected complexity and for proving *positive* or *strong* almost-sure termination (SAST) of innermost rewriting with PTRSs, i.e., finite expected runtime. We implemented our framework in the tool AProVE and demonstrate its power compared to existing techniques for proving SAST.

CCS Concepts

• **Theory of computation → Equational logic and rewriting**; **Probabilistic computation**; **Program analysis**.

Keywords

Term Rewriting, Dependency Pairs, Probabilistic Programs, Termination Analysis, Complexity Analysis

ACM Reference Format:
Jan-Christoph Kassing, Leon Spitzer, and Jürgen Giesl. 2025. Dependency Pairs for Expected Innermost Runtime Complexity and Strong Almost-Sure Termination of Probabilistic Term Rewriting. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Probabilistic programming integrates probabilistic branching into traditional computer models, with applications in many areas [24]. Probabilities do not only handle uncertainty in data, but they can also be used to decrease the expected runtime of algorithms.

Term rewriting [9] is a fundamental concept to transform and evaluate expressions, which is used, e.g., for symbolic computation, automated theorem proving, and automatic program analysis. There exist many approaches to prove termination or infer bounds on the runtime complexity of TRSs, for example, via *ranking functions* like polynomial interpretations [45]. One of the most powerful approaches to analyze termination and runtime complexity of TRSs is the *dependency pair* (DP) framework, see, e.g., [3, 4, 21, 26, 54]. It uses a divide-and-conquer approach to transform termination or complexity problems into simpler subproblems repeatedly (via *DP processors*). Indeed, DPs are used in essentially all current termination and complexity tools for TRSs, e.g., AProVE [22], MU-TERM [25], NaTT [58], TcT [5], $\text{T}_{\text{T}}\text{T}_2$ [41], etc.

Probabilistic TRSs (PTRSs) have been introduced in [7, 13, 14]. A PTRS $\mathcal{R}$ is *almost-surely terminating* (AST) if every evaluation (or "reduction") terminates with probability 1. A strictly stronger notion is *positive* AST (PAST), where every reduction must consist of a finite expected number of rewrite steps. An even stronger notion is *strong* AST (SAST) which requires that for every term t, the supremum over the expected number of rewrite steps of all possible reductions starting in t is finite. It is well known that SAST implies PAST and that PAST implies AST. To see the difference between PAST and SAST, note that a term t can start infinitely many reductions of different expected numbers of rewrite steps due to non-determinism. So even if all reductions are finite, the supremum of their expected lengths may be infinite. In this paper, we develop an approach to prove SAST for PTRSs under an innermost evaluation strategy where we only consider reductions starting with *basic* terms (which represent the application of an algorithm to data objects). Moreover, our approach computes upper bounds on the *expected innermost runtime complexity* of PTRSs. *Runtime complexity* is one of the standard notions of complexity for non-probabilistic TRSs [27], and it was adapted to *expected runtime complexity* for PTRSs in [35].

Related Work: There are numerous techniques to prove (P)AST for *imperative programs on numbers*, e.g., [1, 2, 8, 15, 16, 18, 29–31, 47–53]. In particular, there also exist several *tools* to analyze (P)AST and expected costs for imperative probabilistic programs, e.g., Absynth [53], Amber [52], Eco-Imp [8], and KoAT [47, 51]. In addition, there are also several related approaches for recursive programs, e.g., to analyze probabilistic higher-order programs based on types or martingales [6, 11, 39, 42–44, 55], or probabilistic imperative languages with recursion [40]. However, only few approaches analyze probabilistic programs on recursive *data structures*, e.g., [10, 46, 57]. While [10] uses pointers to represent data structures like tables and lists, [46, 57] consider a probabilistic programming language with matching similar to term rewriting and develop an automatic amortized resource analysis via fixed template potential

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

functions. However, these works are mostly targeted towards specific data structures, whereas our aim is a fully automatic approach for general PTRSs that can model arbitrary data structures.

Currently, the only approach to analyze SAST of PTRSs automatically is the direct application of polynomial or matrix interpretations [17] to the whole PTRS [7], implemented in NaTT. However, already for non-probabilistic TRSs such a direct application of orderings is limited in power. For a powerful approach, one should combine orderings in a modular way, as in the DP framework.

Therefore, we already adapted the DP framework to the probabilistic setting in order to prove AST, both for innermost [32, 36] and full rewriting [37]. Moreover, in the non-probabilistic setting, DPs were extended to analyze complexity instead of just termination, see, e.g., [4, 54]. But up to now there did not exist any DP framework to prove SAST or PAST, or to infer bounds on the expected runtime of PTRSs. In this paper we show that the DP framework for AST from [36] which uses *annotated dependency pairs* can be lifted to a novel DP framework for expected complexity of PTRSs.

Moreover, in [34] we presented criteria for classes of PTRSs where (P)AST for innermost rewriting implies (P)AST for full rewriting, and in [35] these criteria were extended to SAST and expected runtime complexity. Thus, they can also be used in order to infer SAST and expected runtime complexity for full instead of innermost rewriting via our novel DP framework, see Sect. 5.

Main Results of the Paper:

- We develop the first DP framework for SAST and expected innermost runtime complexity of probabilistic TRSs.
- We introduce several processors for our novel DP framework.
- To evaluate the power of our novel framework, we implemented it in the tool AProVE.

Structure: We recapitulate (probabilistic) term rewriting in Sect. 2. In Sect. 3, we introduce our novel DP framework for SAST and expected runtime complexity. Afterwards, we present several processors that can be used in our framework in Sect. 4. In Sect. 5, we give experimental results, also in comparison to the technique of [7]. We refer to [38] for the proofs of all our results.

2 Preliminaries

We recapitulate ordinary and probabilistic TRSs in Sect. 2.1 and 2.2.

2.1 Term Rewriting

We assume some familiarity with term rewriting [9], but recapitulate all needed notions. For any relation $\to \subseteq A \times A$ on a set A and $n \in \mathbb{N}$, we define $\to^n$ as $\to^0 = \{(a,a) \mid a \in A\}$ and $\to^{n+1} = \to^n \circ \to$, where "$\circ$" denotes composition of relations. Moreover, $\to^* = \bigcup_{n\in\mathbb{N}} \to^n$, i.e., $\to^*$ is the reflexive and transitive closure of $\to$.

The set $\mathcal{T}(\Sigma,\mathcal{V})$ of all *terms* over a finite set of *function symbols* $\Sigma = \biguplus_{k\in\mathbb{N}} \Sigma_k$ and a (possibly infinite) set of *variables* $\mathcal{V}$ is the smallest set with $\mathcal{V} \subseteq \mathcal{T}(\Sigma,\mathcal{V})$, and if $f \in \Sigma_k$ and $t_1,\dots,t_k \in \mathcal{T}(\Sigma,\mathcal{V})$ then $f(t_1,\dots,t_k) \in \mathcal{T}(\Sigma,\mathcal{V})$. If Σ and $\mathcal{V}$ are clear from the context, then we just write $\mathcal{T}$ instead of $\mathcal{T}(\Sigma,\mathcal{V})$. The *arity* of a function symbol $f \in \Sigma_k$ is k. For example, consider the signature $\Sigma_{\mathsf{q}} = \{\mathsf{q}, \mathsf{start}, \mathsf{s}, 0\}$, where 0 has arity 0, s has arity 1, start has arity 2, and q has arity 3. Then, for $x \in \mathcal{V}$, $\mathsf{start}(\mathsf{s}(0),\mathsf{s}(0))$ and $\mathsf{q}(0,x,x)$ are terms in $\mathcal{T}(\Sigma_{\mathsf{q}},\mathcal{V})$. For a non-variable term $t = f(t_1,\dots,t_k)$, we define $\mathrm{root}(t) = f$. A term without variables is called a *ground* term. The *size* $|t|$ of a term t is the number of occurrences of function symbols and variables in t, i.e., $|t| = 1$ if $t \in \mathcal{V}$, and $|t| = 1 + \sum_{j=1}^{k} |t_j|$ if $t = f(t_1,\dots,t_k)$. Thus, $|\mathsf{start}(\mathsf{s}(0),\mathsf{s}(0))| = 5$ and $|\mathsf{q}(0,x,x)| = 4$. A *substitution* is a function $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma,\mathcal{V})$ with $\sigma(x) = x$ for all but finitely many $x \in \mathcal{V}$, and we often write $x\sigma$ instead of $\sigma(x)$. Substitutions homomorphically extend to terms: if $t = f(t_1,\dots,t_k) \in \mathcal{T}$ then $t\sigma = f(t_1\sigma,\dots,t_k\sigma)$. Thus, for a substitution σ with $\sigma(x) = \mathsf{s}(x)$ we obtain $\mathsf{q}(0,x,x)\sigma = \mathsf{q}(0,\mathsf{s}(x),\mathsf{s}(x))$. For any term $t \in \mathcal{T}$, the set of *positions* $\mathrm{Pos}(t)$ is the smallest subset of $\mathbb{N}^*$ satisfying $\varepsilon \in \mathrm{Pos}(t)$, and if $t = f(t_1,\dots,t_k)$ then for all $1 \le j \le k$ and all $\pi \in \mathrm{Pos}(t_j)$ we have $j.\pi \in \mathrm{Pos}(t)$. A position π_1 is *above* π_2 if π_1 is a prefix of π_2. If $\pi \in \mathrm{Pos}(t)$ then $t|_\pi$ denotes the subterm starting at position π and $t[r]_\pi$ denotes the term that results from replacing the subterm $t|_\pi$ at position π with the term $r \in \mathcal{T}$. We write $s \trianglelefteq t$ if s is a subterm of t and $s \triangleleft t$ if s is a *proper* subterm of t (i.e., if $s \trianglelefteq t$ and $s \neq t$). For example, we have $\mathrm{Pos}(\mathsf{q}(0,x,x)) = \{\varepsilon,1,2,3\}$, $\mathsf{q}(0,x,x)|_2 = x$, $\mathsf{q}(0,x,x)[\mathsf{s}(x)]_2 = \mathsf{q}(0,\mathsf{s}(x),x)$, and $\mathsf{s}(x) \triangleleft \mathsf{q}(0,\mathsf{s}(x),x)$.

A *rewrite rule* $\ell \to r$ is a pair of terms $(\ell,r) \in \mathcal{T} \times \mathcal{T}$ such that $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$ and $\ell \notin \mathcal{V}$, where $\mathcal{V}(t)$ denotes the set of all variables occurring in $t \in \mathcal{T}$. A *term rewrite system* (TRS) is a finite set of rewrite rules. As an example, consider the following TRS $\mathcal{R}_{\mathsf{q}}$ that is used to compute the rounded **q**uotient of two natural numbers (represented by the successor function s and 0) [3].

$$\begin{array}{ll} \mathsf{start}(x,y) \to \mathsf{q}(x,y,y) & \mathsf{q}(x,0,\mathsf{s}(z)) \to \mathsf{s}(\mathsf{q}(x,\mathsf{s}(z),\mathsf{s}(z))) \\ \mathsf{q}(\mathsf{s}(x),\mathsf{s}(y),z) \to \mathsf{q}(x,y,z) & \mathsf{q}(0,\mathsf{s}(y),\mathsf{s}(z)) \to 0 \end{array}$$

A TRS $\mathcal{R}$ induces a *rewrite relation* $\to_{\mathcal{R}} \subseteq \mathcal{T} \times \mathcal{T}$ on terms where $s \to_{\mathcal{R}} t$ holds if there is a position $\pi \in \mathrm{Pos}(s)$, a rule $\ell \to r \in \mathcal{R}$, and a substitution σ such that $s|_\pi = \ell\sigma$ and $t = s[r\sigma]_\pi$. Let $\mathsf{NF}_{\mathcal{R}}$ denote the set of all terms that are in *normal form* w.r.t. $\to_{\mathcal{R}}$, i.e., $s \in \mathsf{NF}_{\mathcal{R}}$ if there is no term t with $s \to_{\mathcal{R}} t$.

A rewrite step $s \to_{\mathcal{R}} t$ is an *innermost* step (denoted $s \xrightarrow{\mathrm{i}}_{\mathcal{R}} t$) if $\ell\sigma \in \mathsf{ANF}_{\mathcal{R}}$, where $\mathsf{ANF}_{\mathcal{R}}$ is the set of all terms in *argument normal form* w.r.t. $\to_{\mathcal{R}}$, i.e., $t \in \mathsf{ANF}_{\mathcal{R}}$ iff $t' \in \mathsf{NF}_{\mathcal{R}}$ for all proper subterms $t' \triangleleft t$. For example, $\mathsf{start}(\mathsf{s}(0),\mathsf{s}(0)) \in \mathsf{ANF}_{\mathcal{R}_{\mathsf{q}}}$ and $\mathsf{start}(\mathsf{start}(0,0),\mathsf{s}(0)) \notin \mathsf{ANF}_{\mathcal{R}_{\mathsf{q}}}$ since it contains the proper subterm $\mathsf{start}(0,0)$, which is not a normal form. Thus, the rewrite step $\mathsf{start}(\mathsf{start}(0,0),\mathsf{s}(0)) \to_{\mathcal{R}_{\mathsf{q}}} \mathsf{q}(\mathsf{start}(0,0),\mathsf{s}(0),\mathsf{s}(0))$ is not an innermost rewrite step, as it takes place at the position of a subterm that is not in argument normal form. The TRS $\mathcal{R}_{\mathsf{q}}$ computes $\lfloor \frac{n}{m} \rfloor$ when starting with the term $\mathsf{start}(\mathsf{s}^n(0),\mathsf{s}^m(0))$, where $\mathsf{s}^n(\dots)$ denotes $n \in \mathbb{N}$ successive s-function symbols. For instance, we have $\lfloor \frac{1}{1} \rfloor = 1$ and

$$\begin{array}{l} \mathsf{start}(\mathsf{s}(0),\mathsf{s}(0)) \xrightarrow{\mathrm{i}}_{\mathcal{R}_{\mathsf{q}}} \mathsf{q}(\mathsf{s}(0),\mathsf{s}(0),\mathsf{s}(0)) \xrightarrow{\mathrm{i}}_{\mathcal{R}_{\mathsf{q}}} \mathsf{q}(0,0,\mathsf{s}(0)) \\ \qquad\qquad \xrightarrow{\mathrm{i}}_{\mathcal{R}_{\mathsf{q}}} \mathsf{s}(\mathsf{q}(0,\mathsf{s}(0),\mathsf{s}(0))) \xrightarrow{\mathrm{i}}_{\mathcal{R}_{\mathsf{q}}} \mathsf{s}(0). \end{array}$$

Essentially, innermost rewriting corresponds to a call-by-value strategy. Note that innermost rewriting allows us to reduce arbitrary subterms in argument normal form, i.e., there is no fixed preference among several parallel subterms in argument normal form, as in a leftmost-innermost strategy.

Already for non-probabilistic TRSs, the techniques for termination and complexity analysis of *innermost* rewriting are significantly stronger than the ones for "full" rewriting where arbitrary rewrite sequences are allowed (the same holds for the probabilistic DP framework for AST in [32, 36, 37]). Moreover, innermost evaluation

is the standard strategy for most programming languages. Hence, in the remainder, we restrict ourselves to innermost rewriting.

The *derivation height* [28] of a term t is the length of the longest $\xrightarrow{i}_{\mathcal{R}}$-sequence starting with t, i.e.,

$$\mathrm{dh}_{\mathcal{R}}(t) = \sup\{n \in \mathbb{N} \mid \exists t' \in \mathcal{T} \text{ such that } t \xrightarrow{i}{}^{n}_{\mathcal{R}} t'\} \in \mathbb{N} \cup \{\omega\}.$$

For example, $\mathrm{dh}_{\mathcal{R}_q}(\mathsf{start}(\mathsf{s}(0),\mathsf{s}(0))) = 4$. We have $\mathrm{dh}_{\mathcal{R}}(t) = 0$ iff $t \in \mathsf{NF}_{\mathcal{R}}$ and $\mathrm{dh}_{\mathcal{R}}(t) = \omega$ iff t starts an infinite sequence of $\xrightarrow{i}_{\mathcal{R}}$-steps, as we restricted ourselves to finite TRSs.

We decompose the signature $\Sigma = \mathcal{D}_{\mathcal{R}} \uplus \mathcal{C}_{\mathcal{R}}$ into *defined symbols* $\mathcal{D}_{\mathcal{R}} = \{\mathrm{root}(\ell) \mid \ell \to r \in \mathcal{R}\}$ and *constructors* $\mathcal{C}_{\mathcal{R}}$. If $\mathcal{R}$ is clear from the context, we just write $\mathcal{C}$ and $\mathcal{D}$. A term $f(t_1,\dots,t_k)$ is *basic* if $f \in \mathcal{D}_{\mathcal{R}}$ and $t_1,\dots,t_k \in \mathcal{T}(\mathcal{C}_{\mathcal{R}},\mathcal{V})$, i.e., $t_1,\dots,t_k$ do not contain defined symbols. Thus, basic terms represent an algorithm f that is applied to data t_1, ..., t_k. So for $\mathcal{R}_q$, $\mathsf{q}(0,x,x)$ is basic, but $\mathsf{q}(\mathsf{q}(0,x,x),x,x)$ is not. Let $\mathcal{BT}_{\mathcal{R}}$ denote the set of basic terms for the TRS $\mathcal{R}$.

The *runtime complexity* $\mathrm{rc}_{\mathcal{R}}$ is a function that maps any $n \in \mathbb{N}$ to the maximum derivation height for basic terms of size $\leq n$.

Definition 2.1 (Runtime Complexity, $\mathrm{rc}_{\mathcal{R}}$ [27]). For a TRS $\mathcal{R}$, its *runtime complexity function* $\mathrm{rc}_{\mathcal{R}} : \mathbb{N} \to \mathbb{N} \cup \{\omega\}$ is:

$$\mathrm{rc}_{\mathcal{R}}(n) = \sup\{\mathrm{dh}_{\mathcal{R}}(t) \mid t \in \mathcal{BT}_{\mathcal{R}}, |t| \leq n\}$$

Given a TRS $\mathcal{R}$, our goal is to determine an upper bound on the *asymptotic complexity* of the function $\mathrm{rc}_{\mathcal{R}}$.

Definition 2.2 (Asymptotic Complexities). We consider a set of complexities $\mathfrak{C} = \{\mathrm{Pol}_0, \mathrm{Pol}_1, \mathrm{Pol}_2, \dots, \mathrm{Exp}, 2\text{-}\mathrm{Exp}, \mathrm{Fin}, \omega\}$ with the order $\mathrm{Pol}_0 \sqsubset \mathrm{Pol}_1 \sqsubset \mathrm{Pol}_2 \sqsubset \dots \sqsubset \mathrm{Exp} \sqsubset 2\text{-}\mathrm{Exp} \sqsubset \mathrm{Fin} \sqsubset \omega$, where $\sqsubseteq$ is the reflexive closure of $\sqsubset$. For any function $f : \mathbb{N} \to \mathbb{N} \cup \{\omega\}$, we define its *complexity* $\iota(f) \in \mathfrak{C}$ as follows:

$$\iota(f) = \begin{cases} \mathrm{Pol}_a & \text{if } a \in \mathbb{N} \text{ is the smallest number with } f(n) \in O(n^a) \\ \mathrm{Exp} & \text{if no such } a \text{ exists, but there is a} \\ & \text{polynomial } \mathrm{pol}(n) \text{ such that } f(n) \in O(2^{\mathrm{pol}(n)}) \\ 2\text{-}\mathrm{Exp} & \text{if no such polynomial exists, but there is a} \\ & \text{polynomial } \mathrm{pol}(n) \text{ such that } f(n) \in O(2^{2^{\mathrm{pol}(n)}}) \\ \mathrm{Fin} & \text{if no such polynomial exists,} \\ & \text{but there is no } n \in \mathbb{N} \text{ with } f(n) = \omega \\ \omega & \text{if there is an } n \in \mathbb{N} \text{ with } f(n) = \omega \end{cases}$$

For any TRS $\mathcal{R}$, its (asymptotic) *runtime complexity* $\iota_{\mathcal{R}}$ is $\iota(\mathrm{rc}_{\mathcal{R}})$.

The TRS $\mathcal{R}_q$ has linear runtime complexity, i.e., $\iota_{\mathcal{R}_q} = \iota(\mathrm{rc}_{\mathcal{R}_q}) = \mathrm{Pol}_1$. For example, any rewrite sequence starting with the basic term $\mathsf{start}(\mathsf{s}^n(0),\mathsf{s}^m(0))$ has at most $2n+2$ rewrite steps.

Finally, we recapitulate a first approach to prove termination and to infer upper bounds on the runtime complexity via polynomial[1] interpretations. A *polynomial interpretation* is a Σ-algebra $\mathcal{I} : \Sigma \to \mathbb{N}[\mathcal{V}]$ that maps every function symbol $f \in \Sigma_k$ to a polynomial $\mathcal{I}_f$ over k variables with natural coefficients. As usual, $\mathcal{I}$ is homomorphically extended to terms. $\mathcal{I}$ is *monotonic* if $x > y$ implies $\mathcal{I}_f(\dots,x,\dots) > \mathcal{I}_f(\dots,y,\dots)$ for all $f \in \Sigma$ and $x, y \in \mathbb{N}$. We call $\mathcal{I}$ a *complexity polynomial interpretation* (CPI) if for all constructors $f \in \mathcal{C}$ we have $\mathcal{I}_f(x_1,\dots,x_k) = a_1x_1 + \dots + a_kx_k + b$, where $b \in \mathbb{N}$ and $a_j \in \{0,1\}$ for all $1 \leq j \leq k$.[2] While arbitrary monotonic polynomial interpretations can be used to prove termination of TRSs [45], monotonic CPIs are needed to infer a polynomial runtime bound from such a termination proof [12, 28]. So the reason for the name CPI is that only such polynomial interpretations give rise to polynomial upper bounds when analyzing runtime complexity.

More precisely, if there is a monotonic polynomial interpretation $\mathcal{I}$ such that $\mathcal{I}(\ell) > \mathcal{I}(r)$ holds for every rule $\ell \to r \in \mathcal{R}$, then the TRS $\mathcal{R}$ is terminating and $\iota_{\mathcal{R}} \sqsubseteq 2\text{-}\mathrm{Exp}$, and if all constructors are interpreted by linear polynomials, then we have $\iota_{\mathcal{R}} \sqsubseteq \mathrm{Exp}$, see [38].

But if $\mathcal{I}$ is a monotonic CPI, we even have $\iota_{\mathcal{R}} \sqsubseteq \mathrm{Pol}_a$ if for all $f \in \mathcal{D}$, the polynomial $\mathcal{I}_f$ has at most degree a. The reason is that this implies $\mathcal{I}(t) \in O(|t|^a)$ for all basic ground terms $t \in \mathcal{BT}_{\mathcal{R}}$. (More precisely, the function that maps any $n \in \mathbb{N}$ to $\sup\{\mathcal{I}(t) \mid t \in \mathcal{BT}_{\mathcal{R}}, \mathcal{V}(t) = \varnothing, |t| \leq n\}$ is in $O(n^a)$.) Since every rewrite step decreases the interpretation of the term by at least 1, the length of each rewrite sequence starting with a basic ground term of size $\leq n$ is in $O(n^a)$.

This direct approach is only feasible for simple examples like the TRS $\mathcal{R}_{\mathsf{plus}} = \{\mathsf{plus}(0,y) \to y,\ \mathsf{plus}(\mathsf{s}(x),y) \to \mathsf{s}(\mathsf{plus}(x,y))\}$, computing the addition of two natural numbers. Let $\mathcal{I}$ be a monotonic CPI with $\mathcal{I}_0 = 0$, $\mathcal{I}_{\mathsf{s}}(x) = x+1$, and $\mathcal{I}_{\mathsf{plus}}(x,y) = 2x+y+1$. We have $\mathcal{I}(\mathsf{plus}(\mathsf{s}(x),y)) = 2x+y+3 > 2x+y+2 = \mathcal{I}(\mathsf{s}(\mathsf{plus}(x,y)))$, and $\mathcal{I}(\mathsf{plus}(0,y)) = y+1 > y = \mathcal{I}(y)$. Thus, $\mathcal{R}_{\mathsf{plus}}$ is terminating and has at most linear runtime complexity, i.e., $\iota_{\mathcal{R}_{\mathsf{plus}}} \sqsubseteq \mathrm{Pol}_1$ (in fact, we have $\iota_{\mathcal{R}_{\mathsf{plus}}} = \mathrm{Pol}_1$). To automate this approach, one can use SMT solvers to search for a suitable CPI $\mathcal{I}$. However, such a direct application of polynomials fails to prove termination or to infer a polynomial upper bound on $\iota_{\mathcal{R}_q}$. In contrast, $\iota_{\mathcal{R}_q} \sqsubseteq \mathrm{Pol}_1$ can be proved by more elaborate techniques like dependency pairs, see, e.g., [54]. Indeed, we will show how to analyze a probabilistic version of $\mathcal{R}_q$ with our novel DP framework.

2.2 Probabilistic Rewriting

A probabilistic TRS has finite multi-distributions on the right-hand sides of its rewrite rules. A finite *multi-distribution* μ on a set $A \neq \varnothing$ is a finite multiset of pairs $(p : a)$, where $0 < p \leq 1$ is a probability and $a \in A$, such that $\sum_{(p:a)\in\mu} p = 1$. Let $\mathrm{FDist}(A)$ denote the set of all finite multi-distributions on A. For $\mu \in \mathrm{FDist}(A)$, its *support* is the multiset $\mathrm{Supp}(\mu) = \{a \mid (p:a) \in \mu \text{ for some } p\}$. A *probabilistic rewrite rule* $\ell \to \mu$ is a pair $(\ell,\mu) \in \mathcal{T} \times \mathrm{FDist}(\mathcal{T})$ such that $\ell \notin \mathcal{V}$ and $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$ for every $r \in \mathrm{Supp}(\mu)$. A *probabilistic TRS* (PTRS) is a finite set of probabilistic rewrite rules. Similar to TRSs, a PTRS $\mathcal{R}$ induces a *(probabilistic) rewrite relation* $\to_{\mathcal{R}} \subseteq \mathcal{T} \times \mathrm{FDist}(\mathcal{T})$ where $s \to_{\mathcal{R}} \{p_1 : t_1, \dots, p_k : t_k\}$ if there is a position $\pi \in \mathrm{Pos}(s)$, a rule $\ell \to \{p_1 : r_1, \dots, p_k : r_k\} \in \mathcal{R}$, and a substitution σ such that $s|_\pi = \ell\sigma$ and $t_j = s[r_j\sigma]_\pi$ for all $1 \leq j \leq k$. We call $s \to_{\mathcal{R}} \mu$ an *innermost* rewrite step (denoted $s \xrightarrow{i}_{\mathcal{R}} \mu$) if $\ell\sigma \in \mathsf{ANF}_{\mathcal{R}}$. Consider the PTRS $\mathcal{R}_{\mathsf{geo}}$ with the only rule $\mathsf{geo}(x) \to \{1/2 : \mathsf{geo}(\mathsf{s}(x)),\ 1/2 : x\}$. When starting with the term $\mathsf{geo}(0)$, it computes the representation $\mathsf{s}^k(0)$ of the number $k \in \mathbb{N}$ with a probability of $(1/2)^{k+1}$, i.e., a geometric distribution.

To track innermost rewrite sequences with their probabilities, we consider (potentially infinite) *rewrite sequence trees (RSTs)* [36]. The nodes v of an $\mathcal{R}$-RST are labeled by pairs $(p_v : t_v)$ of a prob-

[1] In this paper, we focus on polynomial interpretations with natural coefficients for simplicity, but our results can be extended to other interpretations where one can define an addition and an expected value operation, e.g., matrix interpretations [17].

[2] For monotonic CPIs one must have $a_j = 1$, but in Sect. 4.3 we will consider weakly monotonic CPIs, where $a_j \in \{0,1\}$ is possible.

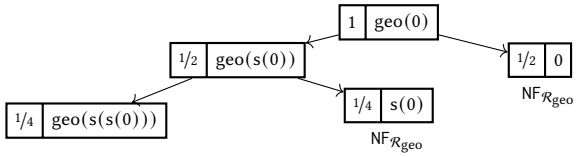


Figure 1: $\mathcal{R}_{geo}$-Rewrite sequence tree starting with $\mathrm{geo}(0)$

ability $p_v \in (0, 1]$ and a term t_v, where the root always has the probability 1. For each node v with successors $w_1, \ldots, w_k$, the edge relation represents an innermost rewrite step, i.e., $t_v \xrightarrow{i}_{\mathcal{R}} \{\frac{p_{w_1}}{p_v} : t_{w_1}, \ldots, \frac{p_{w_k}}{p_v} : t_{w_k}\}$. For an $\mathcal{R}$-RST $\mathfrak{T}$, $V^{\mathfrak{T}}$ denotes its set of nodes, $\mathrm{root}(\mathfrak{T})$ is the term at its root, and $\mathrm{Leaf}^{\mathfrak{T}}$ denotes its set of leaves. A finite RST for $\mathcal{R}_{geo}$ is shown in Fig. 1, but of course this RST could be extended further to an infinite one.

A PTRS $\mathcal{R}$ is *almost-surely terminating* (AST) if $\sum_{v \in \mathrm{Leaf}^{\mathfrak{T}}} p_v = 1$ holds for all $\mathcal{R}$-RSTs $\mathfrak{T}$, i.e., if the probability of termination is always 1. This notion of AST for PTRSs is equivalent to the ones in [7, 14, 32] where AST is defined via a lifting of $\xrightarrow{i}_{\mathcal{R}}$ to multisets or via stochastic processes. However, AST is not sufficient to guarantee that the expected runtime complexity of a PTRS is finite. To define this concept formally, we first introduce the *expected derivation length* of an $\mathcal{R}$-RST $\mathfrak{T}$ as

$$\mathrm{edl}(\mathfrak{T}) = \textstyle\sum_{v \in V^{\mathfrak{T}} \setminus \mathrm{Leaf}^{\mathfrak{T}}} p_v.$$

So $\mathrm{edl}(\mathfrak{T})$ adds up the probabilities of all rewrite steps in $\mathfrak{T}$. Thus, for the RST $\mathfrak{T}$ in Fig. 1 we obtain $\mathrm{edl}(\mathfrak{T}) = 1 + 1/2 = 3/2$, i.e., in expectation we perform $3/2$ rewrite steps in $\mathfrak{T}$. Then a PTRS $\mathcal{R}$ is *positively almost-surely terminating* (PAST) if $\mathrm{edl}(\mathfrak{T})$ is finite for all $\mathcal{R}$-RSTs $\mathfrak{T}$. Again, this notion of PAST for PTRSs is equivalent to the ones in [7, 14]. Clearly, PAST implies AST, but not vice versa (e.g., a PTRS with the rule $\mathrm{g} \to \{1/2 : \mathrm{c}(\mathrm{g}, \mathrm{g}),\ 1/2 : 0\}$ which represents a symmetric random walk is AST, but not PAST).

To adapt the notions for complexity from TRSs to PTRSs, recall that in the non-probabilistic setting, the *derivation height* of a term t does not consider a fixed rewrite sequence, but all possible rewrite sequences starting with t and takes the supremum of their lengths. Similarly, while edl considers a fixed RST $\mathfrak{T}$, for the *expected derivation height* of a term t, we consider all possible RSTs with root t and take the supremum of their expected derivation lengths. So the *expected derivation height* of a term t is

$$\mathrm{edh}_{\mathcal{R}}(t) = \sup\{\mathrm{edl}(\mathfrak{T}) \mid \mathfrak{T} \text{ is an } \mathcal{R}\text{-RST with } \mathrm{root}(\mathfrak{T}) = t\}.$$

Now we can adapt the notion of runtime complexity to PTRSs.

Definition 2.3 (Expected Runtime Complexity, $\mathrm{erc}_{\mathcal{R}}$). For a PTRS $\mathcal{R}$, its *expected runtime complexity function* $\mathrm{erc}_{\mathcal{R}} : \mathbb{N} \to \mathbb{N} \cup \{\omega\}$ is:

$$\mathrm{erc}_{\mathcal{R}}(n) = \sup\{\mathrm{edh}_{\mathcal{R}}(t) \mid t \in \mathcal{BT}_{\mathcal{R}}, |t| \leq n\}$$

Moreover, we define $\mathcal{R}$'s *expected runtime complexity* $\iota_{\mathcal{R}}$ as $\iota(\mathrm{erc}_{\mathcal{R}})$.

A PTRS $\mathcal{R}$ is *strongly* or *bounded almost-surely terminating* (SAST) if $\mathrm{edh}_{\mathcal{R}}(t)$ is finite for every term t. So in contrast to PAST, here one requires a finite bound on the expected derivation lengths of all RSTs with the same term at the root. Such notions of SAST were defined in, e.g., [7, 19, 48]. SAST implies PAST, but not vice versa (a PTRS with finitely many rules that is PAST but not SAST is given in [35]). However, as also shown in [35], SAST and PAST are "almost always" equivalent for finite PTRSs (e.g., whenever the signature contains a function symbol of arity ≥ 2).

Notation	Description
$\mathcal{T}$	Set of all terms
$\mathcal{BT}_{\mathcal{R}}$	Set of all basic terms for a PTRS $\mathcal{R}$
$\mathrm{edl}(\mathfrak{T})$	Expected derivation length of an RST $\mathfrak{T}$ (Sum of all probabilities of inner nodes)
$\mathrm{edh}_{\mathcal{R}}(t)$	Expected derivation height of a term t w.r.t. a PTRS $\mathcal{R}$ (Supremum over all $\mathrm{edl}(\mathfrak{T})$ for $\mathcal{R}$-RSTs $\mathfrak{T}$ starting with t)
$\mathrm{erc}_{\mathcal{R}}(n)$	Expected runtime complexity function (Supremum over all $\mathrm{edh}_{\mathcal{R}}(t)$ for basic terms t of size $\leq n$)
$\iota_{\mathcal{R}}$	Expected runtime complexity $\iota(\mathrm{erc}_{\mathcal{R}})$ of a PTRS $\mathcal{R}$
AST	Sum of all probabilities of leaves is 1 for every $\mathcal{R}$-RST
PAST	$\mathrm{edl}(\mathfrak{T})$ is finite for every $\mathcal{R}$-RST $\mathfrak{T}$
SAST	$\mathrm{edh}_{\mathcal{R}}(t)$ is finite for every basic term t

Table 1: Key notions

As mentioned, in this paper we restrict ourselves to innermost reductions that start with *basic* terms. So we regard a PTRS $\mathcal{R}$ to be SAST if $\mathrm{edh}_{\mathcal{R}}(t)$ is finite for all *basic* terms t, or equivalently, $\mathrm{erc}_{\mathcal{R}}(n) \neq \omega$ for all $n \in \mathbb{N}$. Thus, we use the following definition of SAST.

Definition 2.4 (Strong Almost-Sure Termination, SAST). A PTRS $\mathcal{R}$ is called *strongly almost-surely terminating* (SAST) if $\iota_{\mathcal{R}} \sqsubseteq \mathrm{Fin}$.

We summarized all introduced key notions in Table 1.

Example 2.5 (Leading Examples). Consider the following PTRS $\mathcal{R}_1$, which is based on the previously defined systems $\mathcal{R}_{\mathrm{q}}$ and $\mathcal{R}_{geo}$.

$$\begin{aligned}
\mathcal{R}_1 : \quad \mathrm{start}(x, y) &\to \{1 : \mathrm{q}(\mathrm{geo}(x), y, y)\} \\
\mathrm{geo}(x) &\to \{1/2 : \mathrm{geo}(\mathrm{s}(x)),\ 1/2 : x\} \\
\mathrm{q}(\mathrm{s}(x), \mathrm{s}(y), z) &\to \{1 : \mathrm{q}(x, y, z)\} \\
\mathrm{q}(x, 0, \mathrm{s}(z)) &\to \{1 : \mathrm{s}(\mathrm{q}(x, \mathrm{s}(z), \mathrm{s}(z)))\} \\
\mathrm{q}(0, \mathrm{s}(y), \mathrm{s}(z)) &\to \{1 : 0\}
\end{aligned}$$

When starting with $\mathrm{start}(\mathrm{s}^n(0), \mathrm{s}^m(0))$, $\mathcal{R}_1$ computes $\lfloor \frac{n+\mathrm{geo}(0)}{m} \rfloor$, i.e., it first increases n according to a geometric distribution, and then computes the quotient like $\mathcal{R}_{\mathrm{q}}$. Thus, $\iota_{\mathcal{R}_1} = \mathrm{Pol}_1$, since $\mathcal{R}_{\mathrm{q}}$ has linear runtime complexity and the geometric distribution only increases n by 2 in expectation. So in particular, $\mathcal{R}_1$ is SAST.

Moreover, consider the PTRS $\mathcal{R}_2$ with the rules:

$$\begin{aligned}
\mathcal{R}_2 : \quad \mathrm{start} &\to \{1 : \mathrm{f}(\mathrm{geo}(0))\} \\
\mathrm{geo}(x) &\to \{1/2 : \mathrm{geo}(\mathrm{s}(x)),\ 1/2 : x\} \\
\mathrm{f}(\mathrm{s}(x)) &\to \{1 : \mathrm{f}(\mathrm{c}(x, x))\} \\
\mathrm{f}(\mathrm{c}(x, y)) &\to \{1 : \mathrm{c}(\mathrm{f}(x), \mathrm{f}(y))\}
\end{aligned}$$

The two f-rules have exponential runtime complexity, as a reduction starting in $\mathrm{f}(\mathrm{s}^n(0))$ creates a full binary tree of height n and visits every inner node once. When beginning with the term start, $\mathcal{R}_2$ first generates the term $\mathrm{f}(\mathrm{s}^k(0))$ with probability $(1/2)^{k+1}$ and then takes at least 2^k steps to terminate. The expected derivation length of the corresponding RST is at least $\sum_{k=0}^{\infty} (1/2)^{k+1} \cdot 2^k = \sum_{k=0}^{\infty} 1/2 = \omega$. Hence, $\iota_{\mathcal{R}_2} = \omega$, i.e., $\mathcal{R}_2$ is not SAST.

3 Annotated Dependency Pairs

In Sect. 3.1 we define *annotated dependency pairs*. While such dependency pairs were used to prove AST in [36] and relative termination of TRSs in [33], we now develop a new criterion in order to use

them for complexity analysis of PTRSs. Afterwards, in Sect. 3.2 we introduce the general idea of our novel framework in order to derive upper bounds on the expected runtime complexity of PTRSs.

3.1 ADP Problems

The core idea of the *dependency pair framework* for termination of TRSs [3, 21] is the following: *a function is terminating iff the arguments of each recursive function call are decreasing w.r.t. some well-founded ordering*. Hence, for every defined symbol $f \in \mathcal{D}$ one introduces a fresh *tuple* or *annotated* symbol $f^\sharp$ that is used to compare the arguments of two successive calls of f. Let $\Sigma^\sharp = \Sigma \cup \mathcal{D}^\sharp$ with $\mathcal{D}^\sharp = \{f^\sharp \mid f \in \mathcal{D}\}$, and for any $\Sigma' \subseteq \Sigma \cup \mathcal{V}$, let $\mathrm{Pos}_{\Sigma'}(t)$ be all positions of t with symbols or variables from Σ'. For any $t = f(t_1, \ldots, t_k) \in \mathcal{T}$ with $f \in \mathcal{D}$, let $t^\sharp = f^\sharp(t_1, \ldots, t_k)$. For termination analysis, one considers each function call in a right-hand side of a rewrite rule on its own, i.e., for each rule $\ell \to r$ with $\mathrm{Pos}_{\mathcal{D}} = \{\pi_1, \ldots, \pi_n\}$, one obtains n *dependency pairs* $\ell^\sharp \to r|^\sharp_{\pi_i}$ for all $1 \le i \le n$. However, for complexity analysis, one has to consider all function calls in a right-hand side simultaneously. Thus, when adapting DPs for complexity analysis in [54], a single *dependency tuple* (DT) $\ell^\sharp \to [r|^\sharp_{\pi_1}, \ldots, r|^\sharp_{\pi_n}]$ is constructed instead of the n dependency pairs. By analyzing the dependency tuples (together with the original rewrite rules), [54] presented a modular *DT framework* that can be used to infer an upper bound on the runtime complexity. However, in contrast to the *chain criterion* of dependency pairs (which states that termination of a TRS is equivalent to the absence of infinite chains of DPs), the *chain criterion* of this dependency tuple framework yields an over-approximation. More precisely, the upper bounds on the runtime complexity obtained via dependency tuples are only tight for *confluent* TRSs.

Recently, we introduced *annotated dependency pairs* (ADPs) to analyze *almost-sure termination* of PTRSs [36]. We now show that by using ADPs instead of dependency tuples, the corresponding chain criterion for (expected) complexity becomes an equivalence again, i.e., it can be used to compute *tight* complexity bounds (irrespective of confluence). Instead of extracting the function calls of right-hand sides and coupling them together in a fresh dependency tuple, in ADPs we annotate these function calls in the original rewrite rule directly, i.e., we keep the original structure of the rule.

Definition 3.1 (Annotations). For $t \in \mathcal{T}^\sharp = \mathcal{T}(\Sigma^\sharp, \mathcal{V})$ and a set of positions $\Phi \subseteq \mathrm{Pos}_{\mathcal{D}\cup\mathcal{D}^\sharp}(t)$, let $\sharp_\Phi(t)$ be the variant of t where the symbols at positions from Φ in t are annotated and all other annotations are removed. So $\mathrm{Pos}_{\mathcal{D}^\sharp}(\sharp_\Phi(t)) = \Phi$ and $\sharp_\varnothing(t)$ removes all annotations from t. We often write $\sharp_{\mathcal{D}}(t)$ instead of $\sharp_{\mathrm{Pos}_{\mathcal{D}}(t)}(t)$ to annotate all defined symbols in t, and $\flat(t)$ instead of $\sharp_\varnothing(t)$, where we extend $\flat$ to multi-distributions, rules, and sets of rules by removing the annotations of all occurring terms. Moreover, $\flat^\uparrow_\pi(t)$ results from removing all annotations from t that are strictly above the position π. We write $t \trianglelefteq^\pi_\sharp s$ if $\pi \in \mathrm{Pos}_{\mathcal{D}^\sharp}(s)$ and $t = \flat(s|_\pi)$, i.e., t results from a subterm of s with annotated root symbol by removing its annotation. If π is not of interest, we just write $t \trianglelefteq_\sharp s$. We often write F instead of $\mathsf{f}^\sharp$ for $\mathsf{f} \in \mathcal{D}$ (e.g., Geo instead of $\mathsf{geo}^\sharp$).

Example 3.2 (Annotations). If $\mathsf{f} \in \mathcal{D}$, then we have $\sharp_{\{1\}}(\mathsf{f}(\mathsf{f}(x))) = \sharp_{\{1\}}(\mathsf{F}(\mathsf{F}(x))) = \mathsf{f}(\mathsf{F}(x))$, $\sharp_{\mathcal{D}}(\mathsf{f}(\mathsf{f}(x))) = \sharp_{\{\varepsilon,1\}}(\mathsf{f}(\mathsf{f}(x))) = \mathsf{F}(\mathsf{F}(x))$, $\flat(\mathsf{F}(\mathsf{F}(x))) = \mathsf{f}(\mathsf{f}(x))$, $\flat^\uparrow_1(\mathsf{F}(\mathsf{F}(x))) = \mathsf{f}(\mathsf{F}(x))$, and $\mathsf{f}(x) \trianglelefteq_\sharp \mathsf{f}(\mathsf{F}(x))$.

The annotations indicate which function calls need to be regarded for complexity analysis. To transform a PTRS into ADPs, initially we annotate all defined symbols in the right-hand sides of rules, since all function calls need to be considered at the start of our analysis. The left-hand side of an ADP is just the left-hand side of the original rule (i.e., in contrast to the DPs of [3, 21], we do not annotate symbols in left-hand sides). The DP and the DT framework work on pairs $\langle \mathcal{P}, \mathcal{R}\rangle$, where $\mathcal{R}$ contains the original rewrite rules and $\mathcal{P}$ is the set of dependency pairs or tuples. In contrast, ADPs already represent the original rewrite rules themselves. We simply add a Boolean flag $m \in \{\mathsf{true}, \mathsf{false}\}$ to indicate whether we still need to consider the corresponding original rewrite rule for our analysis. Initially, the flag is true for all ADPs.

Definition 3.3 (Annotated Dependency Pairs). An *annotated dependency pair* (ADP) has the form $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\}^m$, where $\ell \in \mathcal{T}$ with $\ell \notin \mathcal{V}$, $m \in \{\mathsf{true}, \mathsf{false}\}$, and for all $1 \le j \le k$ we have $r_j \in \mathcal{T}^\sharp$ with $\mathcal{V}(r_j) \subseteq \mathcal{V}(\ell)$.

The *canonical ADP* of a probabilistic rule $\ell \to \mu = \{p_1 : r_1, \ldots, p_k : r_k\}$ is $\mathcal{A}(\ell \to \mu) = \ell \to \{p_1 : \sharp_{\mathcal{D}}(r_1), \ldots, p_k : \sharp_{\mathcal{D}}(r_k)\}^{\mathsf{true}}$. The canonical ADPs of a PTRS $\mathcal{R}$ are $\mathcal{A}(\mathcal{R}) = \{\mathcal{A}(\ell \to \mu) \mid \ell \to \mu \in \mathcal{R}\}$.

For a set of ADPs, the defined symbols, constructors, and basic terms are defined as for a TRS, because the left-hand sides of the ADPs are the left-hand sides of the original rewrite rules.

Example 3.4 (Canonical Annotated Dependency Pairs). The canonical ADPs $\mathcal{A}(\mathcal{R}_1)$ and $\mathcal{A}(\mathcal{R}_2)$ of $\mathcal{R}_1$ and $\mathcal{R}_2$ from Ex. 2.5 are:

$$\begin{aligned}
\mathcal{A}(\mathcal{R}_1):\quad \mathsf{start}(x,y) &\to \{1 : \mathsf{Q}(\mathsf{Geo}(x), y, y)\}^{\mathsf{true}} && (1)\\
\mathsf{geo}(x) &\to \{1/2 : \mathsf{Geo}(\mathsf{s}(x)), 1/2 : x\}^{\mathsf{true}} && (2)\\
\mathsf{q}(\mathsf{s}(x), \mathsf{s}(y), z) &\to \{1 : \mathsf{Q}(x, y, z)\}^{\mathsf{true}} && (3)\\
\mathsf{q}(x, 0, \mathsf{s}(z)) &\to \{1 : \mathsf{s}(\mathsf{Q}(x, \mathsf{s}(z), \mathsf{s}(z)))\}^{\mathsf{true}} && (4)\\
\mathsf{q}(0, \mathsf{s}(y), \mathsf{s}(z)) &\to \{1 : 0\}^{\mathsf{true}} && (5)\\
\mathcal{A}(\mathcal{R}_2):\quad \mathsf{start} &\to \{1 : \mathsf{F}(\mathsf{Geo}(0))\}^{\mathsf{true}}\\
\mathsf{geo}(x) &\to \{1/2 : \mathsf{Geo}(\mathsf{s}(x)), 1/2 : x\}^{\mathsf{true}}\\
\mathsf{f}(\mathsf{s}(x)) &\to \{1 : \mathsf{F}(\mathsf{c}(x, x))\}^{\mathsf{true}} && (6)\\
\mathsf{f}(\mathsf{c}(x, y)) &\to \{1 : \mathsf{c}(\mathsf{F}(x), \mathsf{F}(y))\}^{\mathsf{true}}
\end{aligned}$$

Since the original rule and all corresponding dependency pairs are encoded in a single ADP, when rewriting with ADPs we have to distinguish whether we intend to rewrite with the original rule or with a dependency pair. This is important as our analysis should only focus on the complexity of rewriting at annotated positions, i.e., of those function calls that we still need to analyze.

Definition 3.5 ($\overset{\mathrm{i}}{\hookrightarrow}_{\mathcal{P}}$). Let $\mathcal{P}$ be a finite set of ADPs. A term $s \in \mathcal{T}^\sharp$ rewrites with $\mathcal{P}$ to $\mu = \{p_1 : t_1, \ldots, p_k : t_k\}$ (denoted $s \overset{\mathrm{i}}{\hookrightarrow}_{\mathcal{P}} \mu$) if there is a position $\pi \in \mathrm{Pos}_{\mathcal{D}\cup\mathcal{D}^\sharp}(s)$, a rule $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\}^m \in \mathcal{P}$, and a substitution σ such that $\flat(s|_\pi) = \ell\sigma \in \mathsf{ANF}_{\mathcal{P}}$ (i.e., all proper subterms are in normal form w.r.t. $\overset{\mathrm{i}}{\hookrightarrow}_{\mathcal{P}}$), and for all $1 \le j \le k$, t_j is defined as follows, depending on the flag m and on whether $\pi \in \mathrm{Pos}_{\mathcal{D}^\sharp}(s)$ holds:

	$\pi \in \mathrm{Pos}_{\mathcal{D}^\sharp}(s)$	$\pi \notin \mathrm{Pos}_{\mathcal{D}^\sharp}(s)$
$m = \mathsf{true}$	$t_j = s[r_j\sigma]_\pi$ **(at)**	$t_j = s[\flat(r_j)\sigma]_\pi$ **(nt)**
$m = \mathsf{false}$	$t_j = \flat^\uparrow_\pi(s[r_j\sigma]_\pi)$ **(af)**	$t_j = \flat^\uparrow_\pi(s[\flat(r_j)\sigma]_\pi)$ **(nf)**

Rewriting with $\mathcal{P}$ is like ordinary probabilistic term rewriting while considering and modifying annotations. We distinguish between **a**-steps (**a**nnotation) and **n**-steps (**n**o annotation). Similar

to the DP and the DT framework for non-probabilistic TRSs, for complexity we only "count" **a**-steps (on positions with **a**nnotated symbols) that apply dependency pairs, and between two **a**-steps there can be several **n**-steps where rules are applied below the position of the next **a**-step (in order to evaluate the arguments of the function call to normal forms). The flag $m \in \{\mathsf{true}, \mathsf{false}\}$ indicates whether the ADP may be used for such **n**-steps on the arguments before an **a**-step on an annotated symbol above.

During an (**at**)-step (for **a**nnotation and **t**rue), all annotations are kept except those in subterms that correspond to variables in the applied rule. Those subterms are normal forms as we consider innermost rewriting. An (**at**)-step at a position π represents an **a**-step as it rewrites at the position of an annotation, but in addition, it can also represent an **n**-step if an annotated symbol is later rewritten at a position above π. To ease readability, we illustrate Def. 3.5 using a rule with non-probabilistic structure, i.e., a probabilistic rewrite rule of the form $\ell \to \{1 : r\}$. An example for an (**at**)-step is:

$$\mathsf{F}(\mathsf{s}(\mathsf{F}(\mathsf{s}(0)))) \overset{\mathsf{i}}{\hookrightarrow}_{\mathcal{A}(\mathcal{R}_2)} \{1 : \mathsf{F}(\mathsf{s}(\mathsf{F}(\mathsf{c}(0,0))))\}$$

using ADP (6). Here, we have $\pi = 1.1$, $\flat(s|_{1.1}) = \mathsf{f}(\mathsf{s}(0)) = \ell\sigma$, where σ instantiates x with the normal form 0, and $r_1 = \mathsf{F}(\mathsf{c}(0,0))$.

A step of the form (**nt**) (for **n**o annotation and **t**rue) performs a rewrite step at the position of a non-annotated defined symbol. This represents only an **n**-step, and thus all annotations on the right-hand side r_j are removed. An example for such a step is:

$$\mathsf{F}(\mathsf{s}(\mathsf{f}(\mathsf{s}(0)))) \overset{\mathsf{i}}{\hookrightarrow}_{\mathcal{A}(\mathcal{R}_2)} \{1 : \mathsf{F}(\mathsf{s}(\mathsf{f}(\mathsf{c}(0,0))))\}$$

using ADP (6). Here, we have the same π, $\ell\sigma$, and σ as above, but use the right-hand side $\flat(r_1) = \mathsf{f}(\mathsf{c}(0,0))$ without annotations.

An (**af**)-step (for **a**nnotation and **f**alse) at a position π only represents an **a**-step, but not an **n**-step, i.e., it does not rewrite the arguments of a function that is evaluated later on an annotated position above. Therefore, we remove all annotations above π, as no **a**-step is allowed to occur above π afterwards. If $\mathcal{A}(\mathcal{R}_2)'$ contains $\mathsf{f}(\mathsf{s}(x)) \to \{1 : \mathsf{F}(\mathsf{c}(x,x))\}^{\mathsf{false}}$, then a step of the form (**af**) would be:

$$\mathsf{F}(\mathsf{s}(\mathsf{F}(\mathsf{s}(0)))) \overset{\mathsf{i}}{\hookrightarrow}_{\mathcal{A}(\mathcal{R}_2)'} \{1 : \mathsf{f}(\mathsf{s}(\mathsf{F}(\mathsf{c}(0,0))))\}$$

Finally, a step of the form (**nf**) (for **n**o annotation and **f**alse) is irrelevant for proving an upper bound on the expected runtime complexity since there can never be another **a**-step at a position above. These steps are only included to ensure that the innermost evaluation strategy is not affected if one modifies the annotations or the flag of ADPs (such modifications will be done by our ADP processors in Sect. 4). An example would be

$$\mathsf{F}(\mathsf{s}(\mathsf{f}(\mathsf{s}(0)))) \overset{\mathsf{i}}{\hookrightarrow}_{\mathcal{A}(\mathcal{R}_2)'} \{1 : \mathsf{f}(\mathsf{s}(\mathsf{f}(\mathsf{c}(0,0))))\}$$

with $\mathsf{f}(\mathsf{s}(x)) \to \{1 : \mathsf{F}(\mathsf{c}(x,x))\}^{\mathsf{false}}$ at Position 1.1 again.

Next, we lift RSTs to *chain trees* that consider rewriting with $\overset{\mathsf{i}}{\hookrightarrow}_{\mathcal{P}}$ instead of $\overset{\mathsf{i}}{\to}_{\mathcal{R}}$.

Definition 3.6 (Chain Tree). Let $\mathfrak{T} = (V, E, L)$ be a (possibly infinite) labeled and directed tree with nodes $V \neq \emptyset$ and edges $E \subseteq V \times V$, where $vE = \{w \mid (v,w) \in E\}$ is finite for every $v \in V$. We say that $\mathfrak{T}$ is a *$\mathcal{P}$-chain tree* ($\mathcal{P}$-CT) if

- $L : V \to (0,1] \times \mathcal{T}^\sharp$ labels every node v by a probability p_v and a term t_v. For the root $v \in V$ of the tree, we have $p_v = 1$.
- If $vE = \{w_1, \ldots, w_k\}$, then $t_v \overset{\mathsf{i}}{\hookrightarrow}_{\mathcal{P}} \{\frac{p_{w_1}}{p_v} : t_{w_1}, \ldots, \frac{p_{w_k}}{p_v} : t_{w_k}\}$.

For every inner node v, let $\mathcal{P}(v) \in \mathcal{P} \times \{(\mathbf{at}), (\mathbf{af}), (\mathbf{nt}), (\mathbf{nf})\}$ be the ADP and the kind of step used for rewriting t_v.

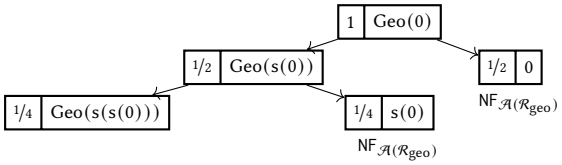


Figure 2: $\mathcal{A}(\mathcal{R}_{\mathrm{geo}})$-Chain tree starting with $\mathrm{Geo}(0)$

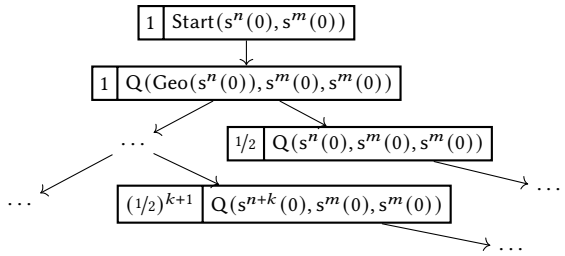
The $\mathcal{A}(\mathcal{R}_{\mathrm{geo}})$-CT in Fig. 2 corresponds to the $\mathcal{R}_{\mathrm{geo}}$-RST in Fig. 1.

In contrast to the *expected derivation length* of RSTs, for the expected derivation length of CTs, we only consider (**at**)- or (**af**)-steps, i.e., steps at the position of an annotated symbol.[3] Moreover, sometimes we do not want to count the application of *all* ADPs, but only the ADPs from some subset. Thus, similar to the adaption of DPs for complexity analysis in [54], in our ADP framework we do not consider a single set of ADPs $\mathcal{P}$, but we use a second set $\mathcal{S} \subseteq \mathcal{P}$ of those ADPs that (still) have to be taken into account. So one only has to add the probabilities of **a**-steps with ADPs from $\mathcal{S}$ in the chain tree to determine its *expected derivation length*. Thus, our ADP framework uses *ADP problems* $\langle \mathcal{P}, \mathcal{S} \rangle$ where $\mathcal{S} \subseteq \mathcal{P}$, and our analysis ends once we have $\mathcal{S} = \emptyset$. Such ADP problems are called *solved*. In the remainder, we fix an arbitrary ADP problem $\langle \mathcal{P}, \mathcal{S} \rangle$.

Definition 3.7 (Exp. Derivation Length for Chain Trees, $\mathrm{edl}_{\langle \mathcal{P},\mathcal{S} \rangle}$*).* Let $\mathfrak{T} = (V, E, L)$ be a $\mathcal{P}$-chain tree. The *expected derivation length* of $\mathfrak{T}$, where we only count steps with $\mathcal{S}$ at annotated symbols, is

$$\mathrm{edl}_{\langle \mathcal{P},\mathcal{S} \rangle}(\mathfrak{T}) = \textstyle\sum_{v \in V \setminus \mathrm{Leaf}^{\mathfrak{T}},\ \mathcal{P}(v) \in \mathcal{S} \times \{(\mathbf{at}),(\mathbf{af})\}} p_v$$

Example 3.8 (Expected Derivation Length for Chain Trees). Reconsider the PTRS $\mathcal{R}_1$ and the following $\mathcal{A}(\mathcal{R}_1)$-chain tree $\mathfrak{T}$.



Let $\mathcal{S}_{\mathrm{geo}} = \{(2)\}$ contain only the geo-ADP and let $\mathcal{S}_{\mathrm{q}} = \{(3), (4), (5)\}$ contain the q-ADPs. Computing the expected derivation length of $\mathfrak{T}$ w.r.t. $\mathcal{S}_{\mathrm{geo}}$ or w.r.t. all ADPs results in

$$\mathrm{edl}_{\langle \mathcal{A}(\mathcal{R}_1), \mathcal{S}_{\mathrm{geo}} \rangle}(\mathfrak{T}) = 1 + \tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8} + \ldots = 1 + \textstyle\sum_{k=1}^{\infty} \frac{1}{2^k} = 2$$

$$\begin{aligned} \mathrm{edl}_{\langle \mathcal{A}(\mathcal{R}_1), \mathcal{A}(\mathcal{R}_1) \rangle}(\mathfrak{T}) &= 1 + \mathrm{edl}_{\langle \mathcal{A}(\mathcal{R}_1), \mathcal{S}_{\mathrm{geo}} \rangle}(\mathfrak{T}) + \mathrm{edl}_{\langle \mathcal{A}(\mathcal{R}_1), \mathcal{S}_{\mathrm{q}} \rangle}(\mathfrak{T}) \\ &\leq 1 + 2 + \textstyle\sum_{k=1}^{\infty} (1/2)^{k+1} \cdot (2(n+k)+2) \\ &= 3 + \textstyle\sum_{k=1}^{\infty} k/2^k + \sum_{k=1}^{\infty} (1/2)^k \cdot (n+1) \\ &= 3 + 2 + 1 \cdot (n+1) = n + 6 \end{aligned}$$

[3] Since (**nt**)- and (**nf**)-steps are disregarded for the expected derivation length of CTs, in contrast to the chain trees used for proving AST in [36], we do not have to require that every infinite path contains infinitely many (**at**)- or (**af**)-steps.

Next, we define *expected derivation height* via chain trees by considering all possible CTs with the root $t^\sharp$ for a basic term t, and taking the supremum of their expected derivation lengths.

Definition 3.9 (Exp. Derivation Height via Chain Trees, $\mathrm{edh}_{\langle\mathcal{P},\mathcal{S}\rangle}$). For $t \in \mathcal{BT}_\mathcal{P}$, the *expected derivation height* $\mathrm{edh}_{\langle\mathcal{P},\mathcal{S}\rangle}(t) \in \mathbb{N} \cup \{\omega\}$ is the supremum obtained when adding all probabilities for **a**-steps with $\mathcal{S}$ in any chain tree $\mathfrak{T}$ with root $t^\sharp$:

$$\mathrm{edh}_{\langle\mathcal{P},\mathcal{S}\rangle}(t) = \sup\{\mathrm{edl}_{\langle\mathcal{P},\mathcal{S}\rangle}(\mathfrak{T}) \mid \mathfrak{T} \text{ is a } \mathcal{P}\text{-chain tree with } \mathrm{root}(\mathfrak{T}) = t^\sharp\}$$

Example 3.10 (Expected Derivation Height w.r.t. Chain Trees). Consider the term $t = \mathrm{start}(s^n(0), s^m(0))$ and its corresponding $\mathcal{A}(\mathcal{R}_1)$-CT from Ex. 3.8. As this is the only $\mathcal{A}(\mathcal{R}_1)$-CT with root $t^\sharp$, we obtain $\mathrm{edh}_{\langle\mathcal{A}(\mathcal{R}_1),\mathcal{A}(\mathcal{R}_1)\rangle}(t) \leq n + 6$ and $\mathrm{edh}_{\langle\mathcal{A}(\mathcal{R}_1),\mathcal{S}_{\mathrm{geo}}\rangle}(t) = 2$.

Now we can define expected runtime complexity for ADP problems.

Definition 3.11 (Expected Runtime Complexity for ADP Problems, $\mathrm{erc}_{\langle\mathcal{P},\mathcal{S}\rangle}$). The *expected runtime complexity function* of an ADP problem $\langle\mathcal{P},\mathcal{S}\rangle$ is defined as

$$\mathrm{erc}_{\langle\mathcal{P},\mathcal{S}\rangle}(n) = \sup\{\mathrm{edh}_{\langle\mathcal{P},\mathcal{S}\rangle}(t) \mid t \in \mathcal{BT}_\mathcal{P}, |t| \leq n\}$$

and we define the *runtime complexity* $\iota_{\langle\mathcal{P},\mathcal{S}\rangle}$ of $\langle\mathcal{P},\mathcal{S}\rangle$ as $\iota(\mathrm{erc}_{\langle\mathcal{P},\mathcal{S}\rangle})$.

Example 3.12 (Expected Runtime Complexity for ADP Problems). For a basic term $\mathrm{start}(t_1, t_2)$, $\mathcal{A}(\mathcal{R}_1)$ first computes a geometric distribution starting in t_1. This needs 2 steps in expectation, and increases t_1 by only 2 in expectation. In the resulting term $\mathrm{q}(t_{\mathrm{geo}}, t_2, t_2)$, where t_{geo} is the normal form resulting from $\mathrm{geo}(t_1)$, we decrease t_{geo} until we reach 0. Therefore, the expected derivation height is linear in the size of the start term, i.e., $\iota_{\langle\mathcal{A}(\mathcal{R}_1),\mathcal{A}(\mathcal{R}_1)\rangle} = \mathrm{Pol}_1$. If we only consider $\mathcal{S}_{\mathrm{geo}}$ for the complexity, then $\iota_{\langle\mathcal{A}(\mathcal{R}_1),\mathcal{S}_{\mathrm{geo}}\rangle} = \mathrm{Pol}_0$.

With our new concepts, we obtain the following novel *chain criterion* for complexity analysis of PTRSs. It shows that to analyze the expected runtime complexity of a PTRS $\mathcal{R}$, it suffices to analyze the expected runtime complexity of its *canonical ADP problem* $\langle\mathcal{A}(\mathcal{R}),\mathcal{A}(\mathcal{R})\rangle$, i.e., in the beginning all ADPs are considered for complexity. In the canonical ADP problem, all defined symbols in right-hand sides are annotated. Thus, one can only perform (**at**)-steps, because due to the innermost strategy, annotations are only removed from subterms in normal form. Hence, the rewrite steps with $\mathcal{R}$ and the ones with $\mathcal{A}(\mathcal{R})$ directly correspond to each other.

Theorem 3.13 (Chain Criterion). *Let $\mathcal{R}$ be a PTRS. Then for all basic terms $t \in \mathcal{BT}_\mathcal{R}$ we have*

$$\mathrm{edh}_\mathcal{R}(t) = \mathrm{edh}_{\langle\mathcal{A}(\mathcal{R}),\mathcal{A}(\mathcal{R})\rangle}(t)$$

and therefore $\iota_\mathcal{R} = \iota_{\langle\mathcal{A}(\mathcal{R}),\mathcal{A}(\mathcal{R})\rangle}$.

So in contrast to the chain criterion of [54] for complexity analysis in the non-probabilistic setting, Thm. 3.13 yields a *tight* bound ($\iota_\mathcal{R} = \iota_{\langle\mathcal{A}(\mathcal{R}),\mathcal{A}(\mathcal{R})\rangle}$) for arbitrary PTRSs due to the usage of ADPs instead of dependency tuples (with dependency tuples one would only obtain an upper bound, i.e., $\iota_\mathcal{R} \sqsubseteq \iota_{\langle\mathcal{A}(\mathcal{R}),\mathcal{A}(\mathcal{R})\rangle}$).

3.2 ADP Framework

Like the original DP framework of [21], our ADP framework is a *divide-and-conquer* approach which applies *processors* to simplify ADP problems until all subproblems are solved. As in [54], a processor also returns a complexity $c \in \mathfrak{C}$.

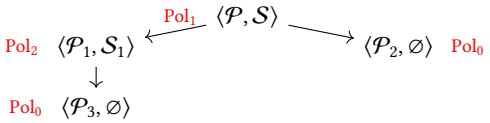


Figure 3: Example of a solved proof tree

Definition 3.14 (Processor). An *(ADP) processor* Proc is a function $\mathrm{Proc}(\langle\mathcal{P},\mathcal{S}\rangle) = (c, \{\langle\mathcal{P}_1,\mathcal{S}_1\rangle, \ldots, \langle\mathcal{P}_n,\mathcal{S}_n\rangle\})$ mapping an ADP problem $\langle\mathcal{P},\mathcal{S}\rangle$ to a complexity $c \in \mathfrak{C}$ and a set of ADP problems.

The intuition for Proc is that in addition to the complexities of $\langle\mathcal{P}_i,\mathcal{S}_i\rangle$, the complexity c is also used to obtain an upper bound on the complexity of $\langle\mathcal{P},\mathcal{S}\rangle$. During the analysis with our ADP framework, we construct a *proof tree* that contains all subproblems and complexities resulting from the application of processors.

Definition 3.15 (Proof Tree). A *proof tree* is a labeled, finite tree $(V, E, L_\mathcal{A}, L_C)$ with a labeling $L_\mathcal{A}$ that maps each node to an ADP problem and a second labeling L_C that maps each node to a complexity from $\mathfrak{C}$. Each edge represents an application of a processor, i.e., if $vE = \{w_1, \ldots, w_n\}$, then $\mathrm{Proc}(L_\mathcal{A}(v)) = (L_C(v), \{L_\mathcal{A}(w_1), \ldots, L_\mathcal{A}(w_n)\})$ for some processor Proc, and we require that the complexity of all leaves v is $L_C(v) = \mathrm{Pol}_0$ if the corresponding ADP problem $L_\mathcal{A}(v)$ is solved, and $L_C(v) = \omega$ otherwise. We call a proof tree *solved* if all ADP problems in its leaves are solved.

Fig. 3 shows an example of a solved proof tree where the complexities given by the labeling L_C are depicted next to the nodes. So here, a processor with $\mathrm{Proc}(\langle\mathcal{P},\mathcal{S}\rangle) = (\mathrm{Pol}_1, \{\langle\mathcal{P}_1,\mathcal{S}_1\rangle, \langle\mathcal{P}_2,\emptyset\rangle\})$ was used for the step from the root to its two children.

To ensure that the maximum of all complexities in a proof tree is an upper bound on the complexity of the ADP problem at the root, we require that proof trees are *well formed*, i.e., that for every node v with $L_\mathcal{A}(v) = \langle\mathcal{P},\mathcal{S}\rangle$, the expected complexity of $\langle\mathcal{P},\mathcal{S}\rangle$ is bounded by the L_C-labels of the subtree starting at v and the L_C-labels on the path from the root to v. However, since no processor has been applied on the leaves of the proof tree (yet), for leaves we use the actual expected complexity instead of the L_C-label. Moreover, for well-formed proof trees we require that the ADPs from $\mathcal{P}\backslash\mathcal{S}$ have already been taken into account in the path from the root to v. This will be exploited, e.g., in the *knowledge propagation processor* of Sect. 4.4.

Definition 3.16 ($\oplus$, Well-Formed Proof Tree). Let $\oplus$ be the maximum operator on complexities, i.e., for $c, d \in \mathfrak{C}$, let $c \oplus d = d$ if $c \sqsubseteq d$ and $c \oplus d = c$ otherwise (so, e.g., $\mathrm{Pol}_2 \oplus \mathrm{Pol}_1 = \mathrm{Pol}_2$).

A proof tree $(V, E, L_\mathcal{A}, L_C)$ *well formed* if for every node v with $L_\mathcal{A}(v) = \langle\mathcal{P},\mathcal{S}\rangle$ and path $v_1, \ldots, v_k = v$ from the root v_1 to v, we have

$$\iota_{\langle\mathcal{P},\mathcal{S}\rangle} \sqsubseteq L_C(v_1) \oplus \cdots \oplus L_C(v_{k-1}) \oplus \max\{L'_C(w) \mid (v, w) \in E^*\}$$
$$\iota_{\langle\mathcal{P},\mathcal{P}\backslash\mathcal{S}\rangle} \sqsubseteq L_C(v_1) \oplus \cdots \oplus L_C(v_{k-1})$$

Here, E^* is the reflexive-transitive closure of the edge relation, i.e., $(v, w) \in E^*$ if v reaches w in the proof tree. Moreover, let $L'_C(v) = L_C(v)$ for inner nodes v and $L'_C(v) = \iota_{L_\mathcal{A}(v)}$ for leaves v.

The following corollary shows that for well-formed proof trees, the complexity of the ADP problem at the root is indeed bounded by the maximum of all complexities at the nodes.

Corollary 3.17 (Complexity Bound from Well-Formed Proof Tree). *Let $\mathfrak{P} = (V, E, L_\mathcal{A}, L_C)$ be a well-formed proof tree with*

$L_{\mathcal{A}}(v_1) = \langle \mathcal{P}, \mathcal{S} \rangle$ *for the root* v_1 *of* $\mathfrak{P}$*. Then,*

$$\iota_{\langle \mathcal{P}, \mathcal{S} \rangle} \sqsubseteq \max\{L_C(v) \mid v \in V\}.$$

Now we can define when a processor is *sound*.

Definition 3.18 (Soundness of Proc.). A processor $\text{Proc}(\langle \mathcal{P}, \mathcal{S} \rangle) = (c, \{\langle \mathcal{P}_1, \mathcal{S}_1 \rangle, \ldots, \langle \mathcal{P}_n, \mathcal{S}_n \rangle\})$ is *sound* if for all well-formed proof trees $(V, E, L_{\mathcal{A}}, L_C)$ and all nodes $v \in V$, we have: if $L_{\mathcal{A}}(v) = \langle \mathcal{P}, \mathcal{S} \rangle$ and $v_1, \ldots, v_k = v$ is the path from the root node v_1 to v, then

$$\iota_{\langle \mathcal{P}, \mathcal{S} \rangle} \sqsubseteq L_C(v_1) \oplus \ldots \oplus L_C(v_{k-1}) \oplus c \oplus \iota_{\langle \mathcal{P}_1, \mathcal{S}_1 \rangle} \oplus \ldots \oplus \iota_{\langle \mathcal{P}_n, \mathcal{S}_n \rangle} \quad (7)$$

$$\iota_{\langle \mathcal{P}_i, \mathcal{P}_i \setminus \mathcal{S}_i \rangle} \sqsubseteq L_C(v_1) \oplus \ldots \oplus L_C(v_{k-1}) \oplus c \qquad \text{for all } 1 \leq i \leq n \quad (8)$$

So (7) requires that the complexity of the considered ADPs $\mathcal{S}$ must be bounded by the maximum of all "previous" complexities $L_C(v_1), \ldots, L_C(v_{k-1})$, the newly derived complexity c, and the complexity of the remaining ADP problems $\langle \mathcal{P}_1, \mathcal{S}_1 \rangle, \ldots, \langle \mathcal{P}_n, \mathcal{S}_n \rangle$. Moreover, (8) ensures that in the remaining ADP problems $\langle \mathcal{P}_i, \mathcal{S}_i \rangle$, the complexity of the "non-considered" ADPs $\mathcal{P}_i \setminus \mathcal{S}_i$ is bounded by the maximum of all previous complexities and the newly derived complexity c. This ensures that well-formedness of proof trees is preserved when extending them by applying sound processors.

Lemma 3.19 (Sound Processors Preserve Well-Formedness). *Let* $\mathfrak{P} = (V, E, L_{\mathcal{A}}, L_C)$ *be a proof tree with a leaf* v *where* $L_{\mathcal{A}}(v)$ *is not solved, and let* Proc *be a sound processor such that* $\text{Proc}(L_{\mathcal{A}}(v)) = (c, \{\langle \mathcal{P}_1, \mathcal{S}_1 \rangle, \ldots, \langle \mathcal{P}_n, \mathcal{S}_n \rangle\})$*. Let* $\mathfrak{P}'$ *result from* $\mathfrak{P}$ *by adding fresh nodes* $w_1, \ldots, w_n$ *and edges* $(v, w_1), \ldots, (v, w_n)$*, where the labeling is extended such that* $L_{\mathcal{A}}(w_i) = \langle \mathcal{P}_i, \mathcal{S}_i \rangle$ *for all* $1 \leq i \leq n$ *and* $L_C(v) = c$*. Then* $\mathfrak{P}'$ *is also well formed.*

To determine an upper bound on the expected runtime complexity $\iota_{\mathcal{R}}$ of a PTRS $\mathcal{R}$, our ADP framework starts with the canonical ADP problem $\langle \mathcal{A}(\mathcal{R}), \mathcal{A}(\mathcal{R}) \rangle$ and applies sound processors repeatedly until all problems are solved. Then by Thm. 3.13, Cor. 3.17, and Lemma 3.19, the expected runtime complexity $\iota_{\mathcal{R}}$ is bounded by the maximum of all complexities occurring in the corresponding proof tree.

Corollary 3.20 (Soundness of the ADP Framework for Expected Runtime Complexity). *Let* $\mathcal{R}$ *be a PTRS and* $\mathfrak{P} = (V, E, L_{\mathcal{A}}, L_C)$ *be a well-formed solved proof tree where* $L_{\mathcal{A}}(v_1) = \langle \mathcal{A}(\mathcal{R}), \mathcal{A}(\mathcal{R}) \rangle$ *for the root* v_1 *of* $\mathfrak{P}$*. Then we have*

$$\iota_{\mathcal{R}} \sqsubseteq \max\{L_C(v) \mid v \in V\}.$$

Remark 3.21. While our framework is inspired by the DT framework of [54] for complexity analysis of non-probabilistic TRSs, our adaption to PTRSs differs from [54] in several aspects. Apart from using ADPs instead of dependency tuples (which results in a "tight" chain criterion instead of an over-approximation), we use proof trees (instead of just proof chains, which allows us to use processors that return several subproblems), and we introduced the novel concept of *well-formed* proof trees and require that sound processors preserve well-formedness. This will allow us to define a *knowledge propagation processor* in Sect. 4.4 which takes the knowledge provided by well-formed proof trees into account. In contrast to [54], we obtain such a processor without extending our ADP problems by an additional component $\mathcal{K}$ that contains those dependency tuples which were already taken into account in the proof up to now, since in our setting we would always have $\mathcal{K} = \mathcal{P} \setminus \mathcal{S}$.

4 ADP Processors

In this section, we adapt the main processors of the DP and the DT framework [21, 54] in order to analyze expected runtime complexity of probabilistic TRSs. Throughout the section, we illustrate our processors with the PTRS $\mathcal{R}_1$ from Ex. 2.5. The resulting solved proof tree for the initial ADP problem $\langle \mathcal{A}(\mathcal{R}_1), \mathcal{A}(\mathcal{R}_1) \rangle$ that we construct during this section is depicted in Fig. 6.

4.1 Usable Rules Processor

We start with a processor that considers the *usable rules*. Usable rules over-approximate the set of those rules that can be used to evaluate the arguments of annotated function symbols if their variables are instantiated by normal forms (as required in innermost evaluations). Essentially, the usable rules of a term t consist of all rules for the defined symbols f occurring in t and all rules that are usable for the terms in the right-hand sides of f-rules.

Definition 4.1 (Usable Rules). For every $f \in \Sigma^\sharp$ and set of ADPs $\mathcal{P}$, let $\text{Rules}_{\mathcal{P}}(f) = \{\ell \to \mu^{\text{true}} \in \mathcal{P} \mid \text{root}(\ell) = f\}$. Moreover, for every $t \in \mathcal{T}^\sharp$, the *usable rules* $\mathcal{U}_{\mathcal{P}}(t)$ of t w.r.t. $\mathcal{P}$ are defined as:

$$\begin{aligned} \mathcal{U}_{\mathcal{P}}(t) &= \varnothing, && \text{if } t \in \mathcal{V} \text{ or } \mathcal{P} = \varnothing \\ \mathcal{U}_{\mathcal{P}}(f(t_1, \ldots, t_k)) &= \text{Rules}_{\mathcal{P}}(f) \cup \textstyle\bigcup_{1 \leq j \leq k} \mathcal{U}_{\mathcal{P}'}(t_j) \\ &\quad \cup \textstyle\bigcup_{\ell \to \mu^{\text{true}} \in \text{Rules}_{\mathcal{P}}(f),\, r \in \text{Supp}(\mu)} \mathcal{U}_{\mathcal{P}'}(\text{b}(r)) \end{aligned}$$

where $\mathcal{P}' = \mathcal{P} \setminus \text{Rules}_{\mathcal{P}}(f)$. The usable rules of $\mathcal{P}$ are

$$\mathcal{U}(\mathcal{P}) = \textstyle\bigcup_{\ell \to \mu^m \in \mathcal{P},\, r \in \text{Supp}(\mu),\, t \trianglelefteq_\sharp r} \mathcal{U}_{\mathcal{P}}(t^\sharp).$$

Similar to the usable rules processor for AST in [36], our usable rules processor sets the flag of all non-usable rules in $\mathcal{P}$ to false to indicate that they cannot be used to evaluate arguments of annotated functions that are rewritten afterwards. The rules in $\mathcal{P}$'s subset $\mathcal{S}$ are changed analogously (since the purpose of $\mathcal{S}$ is only to indicate which ADPs must still be counted for complexity).

Theorem 4.2 (Usable Rules Pr.). *For an ADP problem* $\langle \mathcal{P}, \mathcal{S} \rangle$*, let*

$$\begin{aligned} \mathcal{P}' &= \mathcal{U}(\mathcal{P}) \cup \{\ell \to \mu^{\text{false}} \mid \ell \to \mu^m \in \mathcal{P} \setminus \mathcal{U}(\mathcal{P})\}, \\ \mathcal{S}' &= (\mathcal{S} \cap \mathcal{U}(\mathcal{P})) \cup \{\ell \to \mu^{\text{false}} \mid \ell \to \mu^m \in \mathcal{S} \setminus \mathcal{U}(\mathcal{P})\}. \end{aligned}$$

Then, $\text{Proc}_{\mathsf{UR}}(\langle \mathcal{P}, \mathcal{S} \rangle) = (\text{Pol}_0, \{\langle \mathcal{P}', \mathcal{S}' \rangle\})$ *is sound.*

Example 4.3 (Usable Rules Processor). Consider the ADP problem $\langle \mathcal{A}(\mathcal{R}_1), \mathcal{A}(\mathcal{R}_1) \rangle$ from Ex. 3.4. There is only one term r in the right-hand sides with an annotated subterm $t \trianglelefteq_\sharp r$ where $t^\sharp$ has a defined symbol below the annotated root, viz. $t^\sharp = \text{Q}(\text{geo}(x), y, y)$. Thus, only the geo-ADP (2) is usable and we can set the flag of all other ADPs to false. Hence, we get $\text{Proc}_{\mathsf{UR}}(\langle \mathcal{A}(\mathcal{R}_1), \mathcal{A}(\mathcal{R}_1) \rangle) = (\text{Pol}_0, \{\langle \mathcal{P}_1, \mathcal{P}_1 \rangle\})$ where $\mathcal{P}_1 = \{(9) - (13)\}$ with

$$\text{start}(x, y) \to \{1 : \text{Q}(\text{Geo}(x), y, y)\}^{\text{false}} \quad (9)$$

$$\text{geo}(x) \to \{1/2 : \text{Geo}(\text{s}(x)),\ 1/2 : x\}^{\text{true}} \quad (10)$$

$$\text{q}(\text{s}(x), \text{s}(y), z) \to \{1 : \text{Q}(x, y, z)\}^{\text{false}} \quad (11)$$

$$\text{q}(x, 0, \text{s}(z)) \to \{1 : \text{s}(\text{Q}(x, \text{s}(z), \text{s}(z)))\}^{\text{false}} \quad (12)$$

$$\text{q}(0, \text{s}(y), \text{s}(z)) \to \{1 : 0\}^{\text{false}} \quad (13)$$

Having fewer rules with the flag true has advantages, e.g., for the *dependency graph* and the *reduction pair processor* in Sect. 4.2 and 4.3.

Example 4.4 (Basic Start Terms). The restriction to basic start terms is not only required to infer polynomial upper bounds from CPIs (see Sect. 2.1), but it is also essential for the soundness of the

usable rules processor. To see this, let $\mathcal{R}_3$ contain all rules of $\mathcal{R}_2$ from Ex. 2.5 except start $\to \{1 : \mathsf{f}(\mathsf{geo}(0))\}$. If we do not require basic start terms, then we can start an evaluation with the term $\mathsf{f}(\mathsf{geo}(0))$, i.e., then $\mathcal{R}_3$ is not SAST. The canonical ADPs $\mathcal{A}(\mathcal{R}_3)$ are the same as for $\mathcal{R}_2$ in Ex. 3.4 just without the start-ADP. Thus, $\mathcal{A}(\mathcal{R}_3)$ has no usable rules and $\mathsf{Proc}_{\mathsf{UR}}$ sets the flag of all ADPs to false. When starting with a term like $\mathsf{F}(\mathsf{Geo}(0))$, then one application of the geo-ADP now removes the annotations of the F-symbols above it, as the geo-ADP now has the flag false. So for the resulting ADP problem, only Geo is annotated in chain trees and thus, they all have finite expected derivation length. Hence, we would now falsely infer that $\mathcal{R}_3$ is SAST w.r.t. arbitrary start terms.

4.2 Dependency Graph Processor

The *dependency graph* is a control flow graph that indicates which function calls can occur after each other. This does not depend on the probabilities, and we can consider each function call on its own. Hence, we can use the ordinary dependency graph of the corresponding (non-probabilistic) dependency pairs. To also detect the predecessors of ADPs $\ell \to \mu$ without annotations, we add a dependency pair $\ell^\sharp \to \bot$ for a fresh symbol $\bot$ in that case.

Definition 4.5 (Non-Probabilistic Variant, Dependency Pairs). For a set of ADPs $\mathcal{P}$, let $\mathsf{np}(\mathcal{P}) = \{\ell \to \flat(r_j) \mid \ell \to \{p_1 : r_1, \dots, p_k : r_k\}^{\mathsf{true}} \in \mathcal{P}, 1 \le j \le k\}$ denote its *non-probabilistic rule variant*. So $\mathsf{np}(\mathcal{P})$ is an ordinary TRS which only considers the ADPs with the flag true. For any ADP $\alpha = \ell \to \{p_1 : r_1, \dots, p_k : r_k\}^m$, let $\mathsf{dp}(\alpha) = \{\ell^\sharp \to t^\sharp \mid 1 \le j \le k, t \trianglelefteq_\sharp r_j\}$. Moreover, let $\mathsf{dp}^\bot(\alpha) = \{\ell^\sharp \to \bot\}$ if $\mathsf{dp}(\alpha) = \emptyset$, and $\mathsf{dp}^\bot(\alpha) = \mathsf{dp}(\alpha)$, otherwise. For $\mathcal{P}$, let $\mathsf{dp}(\mathcal{P}) = \bigcup_{\alpha \in \mathcal{P}} \mathsf{dp}^\bot(\alpha)$ denote its *non-probabilistic DP variant*, which is a set containing dependency pairs as in [3, 21] and rules of the form $\ell^\sharp \to \bot$.

So each dependency pair from $\mathsf{dp}(\alpha)$ corresponds to a single annotation on the right-hand side of the ADP α. In the dependency graph, the edges indicate whether one DP can follow another when the instantiated arguments are evaluated with $\mathsf{np}(\mathcal{P})$.

Definition 4.6 (Dependency Graph). The *$\mathcal{P}$-dependency graph* has the nodes $\mathsf{dp}(\mathcal{P})$ and there is an edge from $\ell_1^\sharp \to t_1^\sharp$ to $\ell_2^\sharp \to \dots$ if there are substitutions σ_1, σ_2 such that $t_1^\sharp\sigma_1 \xrightarrow{\mathsf{i}}{}^*_{\mathsf{np}(\mathcal{P})} \ell_2^\sharp\sigma_2$ and both $\ell_1^\sharp\sigma_1$ and $\ell_2^\sharp\sigma_2$ are in argument normal form, i.e., $\ell_1^\sharp\sigma_1, \ell_2^\sharp\sigma_2 \in \mathsf{ANF}_{\mathcal{P}}$.

While the dependency graph is not computable in general, several techniques have been developed to compute over-approximations of the graph automatically, e.g., [3, 21, 26].

Example 4.7 (Dependency Graph). We continue with $\langle \mathcal{P}_1, \mathcal{P}_1 \rangle$ from Ex. 4.3, where $\mathcal{P}_1 = \{(9) - (13)\}$. We have

$$\begin{aligned}
\mathsf{dp}(\mathcal{P}_1) = \{\ \mathsf{Start}(x,y) &\to \mathsf{Q}(\mathsf{geo}(x), y, y), && (14)\\
\mathsf{Start}(x,y) &\to \mathsf{Geo}(x), && (15)\\
\mathsf{Geo}(x) &\to \mathsf{Geo}(\mathsf{s}(x)), && (16)\\
\mathsf{Q}(\mathsf{s}(x), \mathsf{s}(y), z) &\to \mathsf{Q}(x, y, z), && (17)\\
\mathsf{Q}(x, 0, \mathsf{s}(z)) &\to \mathsf{Q}(x, \mathsf{s}(z), \mathsf{s}(z)), && (18)\\
\mathsf{Q}(0, \mathsf{s}(y), \mathsf{s}(z)) &\to \bot\ \}. && (19)
\end{aligned}$$

The $\mathcal{P}_1$-dependency graph is depicted in Fig. 4.

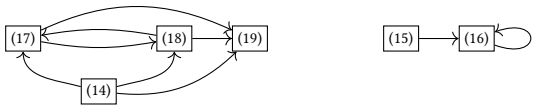


Figure 4: $\mathcal{P}_1$-dependency graph

The idea of the *dependency graph processor* for termination analysis is to analyze each strongly connected component (SCC)[4] of the dependency graph separately. However (already in the non-probabilistic setting, e.g., [54]), this is not possible when analyzing complexity. There are examples where all SCCs have linear complexity but the full system has quadratic complexity, or where all individual SCCs are SAST but the full system is not (see Ex. 4.11).

The problem is that when considering an SCC individually, then we lose the information how often and with which instantiations of the variables this SCC is "called". For that reason, we now present a novel dependency graph processor which regards each SCC *together with its "prefix"*, i.e., together with all nodes of the dependency graph that can reach the SCC. As usual, we say that a node *reaches* an SCC if there is a path from the node to the SCC in the dependency graph (where the path has length ≥ 0, i.e., each node also reaches itself). However, prefixes which are independent from each other can be regarded separately, i.e., we only regard SCC-prefixes $\mathcal{J}$ where for all nodes $\alpha, \beta \in \mathcal{J}$, α reaches β or β reaches α.

Definition 4.8 (SCC-Prefix). Let $\mathcal{P}$ be a set of ADPs. Then $\mathcal{J}$ is an *SCC-prefix* of the $\mathcal{P}$-dependency graph if there exists an SCC $\mathcal{G} \subseteq \mathcal{J}$ where $\mathcal{J} \subseteq \mathsf{dp}(\mathcal{P})$ is a maximal set such that all DPs of $\mathcal{J}$ reach $\mathcal{G}$ and for all $\alpha, \beta \in \mathcal{J}$, α reaches β or β reaches α.

For example, the $\mathcal{P}_1$-dependency graph of Fig. 4 has two SCCs $\{(16)\}$ and $\{(17), (18)\}$, and two SCC-prefixes $\mathcal{J}_1 = \{(15), (16)\}$ and $\mathcal{J}_2 = \{(14), (17), (18)\}$. In this example, $\mathcal{J}_1$ and $\mathcal{J}_2$ represent two completely independent parts of the dependency graph.

The dependency graph processor now handles each SCC-prefix $\mathcal{J}$ separately. To consider only the effects of the DPs $\mathcal{J}$ in the ADPs of $\mathcal{P}$, we replace every ADP α by the variant $\alpha|_{\mathcal{J}}$ where only those symbols are annotated that correspond to the DPs from $\mathcal{J}$. For example, for $\mathcal{J}_1 = \{(15), (16)\}$ and the ADP

$$\alpha = \qquad \mathsf{start}(x,y) \to \{1 : \mathsf{Q}(\mathsf{Geo}(x), y, y)\}^{\mathsf{false}}, \qquad (9)$$

in $\alpha|_{\mathcal{J}_1}$ we only annotate Geo but not Q in the right-hand side, due to the DP (15) which results from its subterm $\mathsf{Geo}(x)$, i.e., $\alpha|_{\mathcal{J}_1} = \mathsf{start}(x,y) \to \{1 : \mathsf{q}(\mathsf{Geo}(x), y, y)\}^{\mathsf{false}}$.

Theorem 4.9 (Dependency Graph Proc.). *Let $\langle \mathcal{P}, \mathcal{S} \rangle$ be an ADP problem and let $\mathcal{J}$ be an SCC-prefix of the $\mathcal{P}$-dependency graph. For any ADP $\alpha = \ell \to \{p_1 : r_1, \dots, p_k : r_k\}^m \in \mathcal{P}$ let $\alpha|_{\mathcal{J}} = \ell \to \{p_1 : \sharp_{\Phi_1}(r_1), \dots, p_k : \sharp_{\Phi_k}(r_k)\}^m$ where for $1 \le j \le k$, we have $\pi \in \Phi_j$ iff there exists an $\ell^\sharp \to t^\sharp \in \mathcal{J}$ such that $t \trianglelefteq_\sharp^\pi r_j$. Similarly, let $\mathcal{P}|_{\mathcal{J}} = \{\alpha|_{\mathcal{J}} \mid \alpha \in \mathcal{P}\}$ and $\mathcal{S}|_{\mathcal{J}} = \{\alpha|_{\mathcal{J}} \mid \alpha \in \mathcal{S}\}$.*[5]

Then $\mathsf{Proc}_{\mathsf{DG}}(\langle \mathcal{P}, \mathcal{S} \rangle) = (\mathsf{Pol}_0, \{\langle \mathcal{P}|_{\mathcal{J}}, \mathcal{S}|_{\mathcal{J}} \rangle \mid \mathcal{J}$ is an SCC-prefix of the $\mathcal{P}$-dependency graph $\})$ is sound.

[4] A set $\mathcal{G}$ of DPs is an *SCC* if it is a maximal cycle, i.e., a maximal set where for any α, α' in $\mathcal{G}$ there is a non-empty path from α to α' only traversing nodes from $\mathcal{G}$.

[5] Note that if a DP from $\mathcal{J}$ could originate from several subterms $r_j|_\pi$ in several ADPs α, then we annotate all their positions in $\mathcal{P}|_{\mathcal{J}}$. A slightly more powerful variant of the processor could be obtained by storing for every DP of $\mathcal{J}$ from which subterm of which ADP it originates, and only annotating this position. We did not do this in Thm. 4.9 to ease the presentation.

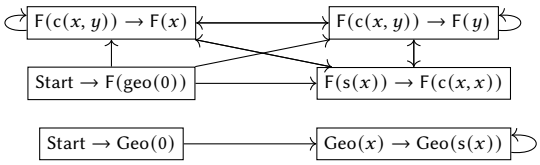


Figure 5: $\mathcal{A}(\mathcal{R}_2)$-dependency graph

Example 4.10 (Dependency Graph Proc.). Due to the two SCC-prefixes $\mathcal{J}_1 = \{(15),(16)\}$ and $\mathcal{J}_2 = \{(14),(17),(18)\}$, the dependency graph processor transforms the ADP problem $\langle \mathcal{P}_1, \mathcal{P}_1 \rangle$ from Ex. 4.3 into $\langle \mathcal{P}_1|_{\mathcal{J}_1}, \mathcal{P}_1|_{\mathcal{J}_1} \rangle$ (corresponding to the Geo-SCC-Prefix) and $\langle \mathcal{P}_1|_{\mathcal{J}_2}, \mathcal{P}_1|_{\mathcal{J}_2} \rangle$ (corresponding to the Q-SCC-Prefix) with

$$\mathcal{P}_1|_{\mathcal{J}_1}: \quad \mathsf{start}(x,y) \to \{1 : \mathsf{q}(\mathsf{Geo}(x), y, y)\}^{\mathsf{false}} \tag{20}$$
$$\mathsf{geo}(x) \to \{1/2 : \mathsf{Geo}(\mathsf{s}(x)),\ 1/2 : x\}^{\mathsf{true}} \tag{21}$$
$$\mathsf{q}(\mathsf{s}(x),\mathsf{s}(y),z) \to \{1 : \mathsf{q}(x,y,z)\}^{\mathsf{false}} \tag{22}$$
$$\mathsf{q}(x,0,\mathsf{s}(z)) \to \{1 : \mathsf{s}(\mathsf{q}(x,\mathsf{s}(z),\mathsf{s}(z)))\}^{\mathsf{false}} \tag{23}$$
$$\mathsf{q}(0,\mathsf{s}(y),\mathsf{s}(z)) \to \{1 : 0\}^{\mathsf{false}} \tag{24}$$

$$\mathcal{P}_1|_{\mathcal{J}_2}: \quad \mathsf{start}(x,y) \to \{1 : \mathsf{Q}(\mathsf{geo}(x), y, y)\}^{\mathsf{false}} \tag{25}$$
$$\mathsf{geo}(x) \to \{1/2 : \mathsf{geo}(\mathsf{s}(x)),\ 1/2 : x\}^{\mathsf{true}} \tag{26}$$
$$\mathsf{q}(\mathsf{s}(x),\mathsf{s}(y),z) \to \{1 : \mathsf{Q}(x,y,z)\}^{\mathsf{false}} \tag{27}$$
$$\mathsf{q}(x,0,\mathsf{s}(z)) \to \{1 : \mathsf{s}(\mathsf{Q}(x,\mathsf{s}(z),\mathsf{s}(z)))\}^{\mathsf{false}} \tag{28}$$
$$\mathsf{q}(0,\mathsf{s}(y),\mathsf{s}(z)) \to \{1 : 0\}^{\mathsf{false}} \tag{29}$$

Our novel dependency graph processor subsumes several previous processors from the literature, like the "leaf removal processor" of [54]. Leaves of the dependency graph like (19) are not part of any SCC. Hence, they are never contained in SCC-prefixes and thus, the annotations that only correspond to such leaves are always removed. For a similar reason, $\mathrm{Proc}_{\mathsf{DG}}$ from Thm. 4.9 subsumes the "rhs simplification processor" of [54] and the related "usable terms processor" of [36], both of which share the same underlying idea.

Ex. 4.11 shows that only considering SCCs without prefixes would be unsound for analyzing complexity and proving SAST.

Example 4.11 (Combining Non-Connected SCCs may Increase Complexity). Recall $\mathcal{R}_2$ from Ex. 2.5 which is not SAST and $\mathcal{A}(\mathcal{R}_2)$ from Ex. 3.4. The $\mathcal{A}(\mathcal{R}_2)$-dependency graph is depicted in Fig. 5. When only considering the SCCs $\mathcal{G}_{\mathsf{Geo}}$ (containing the Geo-DP) and $\mathcal{G}_{\mathsf{F}}$ (containing the three F-DPs), then we could falsely "prove" SAST since $\iota_{\langle \mathcal{A}(\mathcal{R}_2)|_{\mathcal{G}_{\mathsf{Geo}}}, \mathcal{A}(\mathcal{R}_2)|_{\mathcal{G}_{\mathsf{Geo}}} \rangle} = \mathrm{Pol}_0$ and $\iota_{\langle \mathcal{A}(\mathcal{R}_2)|_{\mathcal{G}_{\mathsf{F}}}, \mathcal{A}(\mathcal{R}_2)|_{\mathcal{G}_{\mathsf{F}}} \rangle} = \mathrm{Exp}$. The problem is that for the F-SCC $\mathcal{G}_{\mathsf{F}}$, one also has to consider the DP $\mathsf{Start} \to \mathsf{F}(\mathsf{geo}(0))$ which determines with which instantiations of the variables the F-SCC is called (i.e., it ensures that the geo-rules become usable). Indeed, we have $\iota(\langle \mathcal{A}(\mathcal{R}_2), \mathcal{A}(\mathcal{R}_2) \rangle) = \omega$.

4.3 Reduction Pair Processor

Now we lift the direct application of polynomial interpretations explained in Sect. 2.1 to an ADP processor for complexity analysis of PTRSs, which allows us to apply polynomial interpretations $\mathcal{I}$ in a modular way. As in the classical DP approach [3, 21], here it suffices if $\mathcal{I}$ is *weakly* monotonic, i.e., if $x \geq y$ implies $\mathcal{I}_f(\ldots, x, \ldots) \geq \mathcal{I}_f(\ldots, y, \ldots)$ for all $f \in \Sigma$ and $x, y \in \mathbb{N}$. Moreover, as in [7, 36], to ensure "weak monotonicity w.r.t. expected values" we restrict ourselves to interpretations with multilinear polynomials, where all monomials have the form $c \cdot x_1^{e_1} \cdot \ldots \cdot x_k^{e_k}$ with $c \in \mathbb{N}$ and $e_1, \ldots, e_k \in \{0, 1\}$.

The *reduction pair processor* imposes three requirements on $\mathcal{I}$:
(1) All rules with the flag true must be weakly decreasing in expectation when removing all annotations. Due to weak monotonicity, this ensures that evaluating the arguments of a function call (i.e., applying the rule in a context) also decreases weakly in expectation.
(2) All ADPs must be weakly decreasing when comparing the annotated left-hand side $\mathcal{I}(\ell^\sharp)$ with the expected value of the annotated subterms of the right-hand side $\{p_1 : r_1, \ldots, p_k : r_k\}$. To measure the value of a term r_j, here we consider all its subterms $t \trianglelefteq_\sharp r_j$ at annotated positions and sum up the polynomial interpretations of all such $t^\sharp$, i.e., we consider $\mathcal{I}^\sharp_\Sigma(r_j) = \sum_{t \trianglelefteq_\sharp r_j} \mathcal{I}(t^\sharp)$. Regarding this sum instead of the interpretation $\mathcal{I}(r_j)$ of the whole term r_j is the reason why only need *weak* monotonicity.
(3) Finally, the processor removes all strictly decreasing ADPs from the component $\mathcal{S}$ of the ADP problem. However, the ADPs are still kept in $\mathcal{P}$, because they may still be used in reductions. Moreover, if $\mathcal{I}$ is a CPI, then the processor infers a polynomial bound corresponding to the degrees of the polynomials used for annotated symbols. Otherwise, it only infers an exponential bound (which is still useful when analyzing SAST).

Theorem 4.12 (Reduction Pair Proc.). *Let $\mathcal{I} : \Sigma^\sharp \to \mathbb{N}(\mathcal{V})$ be a weakly monotonic, multilinear polynomial interpretation. Let $\langle \mathcal{P}, \mathcal{S} \rangle$ be an ADP problem where $\mathcal{P} = \mathcal{P}_\geq \uplus \mathcal{P}_>$ and $\mathcal{P}_> \cap \mathcal{S} \neq \emptyset$ such that:*
(1) For every $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\}^{\mathsf{true}} \in \mathcal{P}$: $\mathcal{I}(\ell) \geq \sum_{1 \leq j \leq k} p_j \cdot \mathcal{I}(\mathfrak{b}(r_j))$
(2) For every $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\}^m \in \mathcal{P}_\geq$: $\mathcal{I}(\ell^\sharp) \geq \sum_{1 \leq j \leq k} p_j \cdot \mathcal{I}^\sharp_\Sigma(r_j)$
(3) For every $\ell \to \{p_1 : r_1, \ldots, p_k : r_k\}^m \in \mathcal{P}_>$: $\mathcal{I}(\ell^\sharp) > \sum_{1 \leq j \leq k} p_j \cdot \mathcal{I}^\sharp_\Sigma(r_j)$,
where $\mathcal{I}^\sharp_\Sigma(r_j) = \sum_{t \trianglelefteq_\sharp r_j} \mathcal{I}(t^\sharp)$.

Then $\mathrm{Proc}_{\mathsf{RP}}(\langle \mathcal{P}, \mathcal{S} \rangle) = (c, \langle \mathcal{P}, \mathcal{S} \setminus \mathcal{P}_> \rangle)$ is sound, where the complexity $c \in \mathfrak{C}$ is determined as follows: If $\mathcal{I}$ is a CPI and for all annotated symbols $f^\sharp \in \mathcal{D}^\sharp$, the polynomial $I_{f^\sharp}$ has at most degree a, then $c = \mathrm{Pol}_a$. If $\mathcal{I}$ is not a CPI, then $c = \mathrm{Exp}$ if all constructors are interpreted by linear polynomials, and otherwise $c = 2\text{-Exp}$.

In contrast to the reduction pair processor for proving AST from [36], our processor cannot remove any annotation from $\mathcal{P}$. The reason is that the ADPs from $\mathcal{P}_>$ may still be needed to reach all annotated terms that are relevant for the complexity of $\mathcal{P}_\geq$. (This problem already occurs when analyzing complexity in the non-probabilistic setting, see, e.g., [54, Ex. 25].) But we can remove strictly decreasing ADPs from $\mathcal{S}$ and, therefore, do not have to count them anymore for the complexity. The complexity of the removed ADPs is accounted for by $c \in \{\mathrm{Pol}_a, \mathrm{Exp}, 2\text{-Exp}\}$.

Example 4.13 (Reduction Pair Processor). Consider $\langle \mathcal{P}_1|_{\mathcal{J}_1}, \mathcal{P}_1|_{\mathcal{J}_1} \rangle$ from Ex. 4.10, and a polynomial interpretation $\mathcal{I}$ with $\mathcal{I}_{\mathsf{Start}}(x,y) = 2$, $\mathcal{I}_{\mathsf{Geo}}(x) = \mathcal{I}_{\mathsf{Q}} = 1$, and $\mathcal{I}_{\mathsf{s}}(x) = \mathcal{I}_{\mathsf{geo}}(x) = x + 1$. Then $\mathsf{geo}(x) \to \{1/2 : \mathsf{Geo}(\mathsf{s}(x)),\ 1/2 : x\}^{\mathsf{true}}$ (21) is weakly decreasing in expectation when disregarding annotations, since $\mathcal{I}(\mathsf{geo}(x)) = x + 1 = 1/2 \cdot \mathcal{I}(\mathsf{geo}(\mathsf{s}(x))) + 1/2 \cdot \mathcal{I}(x)$. Moreover, when regarding annotations, then all ADPs are strictly decreasing: for (21) we have $\mathcal{I}(\mathsf{Geo}(x)) = 1 > 1/2 = 1/2 \cdot \mathcal{I}(\mathsf{Geo}(\mathsf{s}(x)))$, for $\mathsf{start}(x,y) \to \{1 : \mathsf{q}(\mathsf{Geo}(x), y, y)\}^{\mathsf{false}}$ (20) we have $\mathcal{I}(\mathsf{Start}(x,y)) = 2 > 1 = \mathcal{I}(\mathsf{Geo}(x))$, and for (22)-(24) we have $\mathcal{I}(\mathsf{Q}(\ldots)) = 1 > 0$ (as their right-hand sides do not contain annotations). Since $\mathcal{I}$ is a CPI which interprets all annotated symbols as constants, we obtain $\mathrm{Proc}_{\mathsf{RP}}(\langle \mathcal{P}_1|_{\mathcal{J}_1}, \mathcal{P}_1|_{\mathcal{J}_1} \rangle) = (\mathrm{Pol}_0, \langle \mathcal{P}_1|_{\mathcal{J}_1}, \varnothing \rangle)$, containing a solved ADP problem.

For the other ADP problem $\langle \mathcal{P}_1|_{\mathcal{J}_2}, \mathcal{P}_1|_{\mathcal{J}_2}\rangle$ from Ex. 4.10, we use a polynomial interpretation with $\mathcal{I}_{\mathsf{Start}}(x,y) = x+3$, $\mathcal{I}_{\mathsf{Geo}}(x) = 1$, and $\mathcal{I}_{\mathsf{Q}}(x,y,z) = \mathcal{I}_{\mathsf{s}}(x) = \mathcal{I}_{\mathsf{geo}}(x) = x+1$. Then (26) is again weakly decreasing when disregarding annotations. When regarding the annotations, then the ADP (28) is weakly decreasing (since $\mathcal{I}(\mathsf{Q}(x,0,\mathsf{s}(z))) = x+1 = \mathcal{I}(\mathsf{Q}(x,\mathsf{s}(z),\mathsf{s}(z)))$), and all other ADPs are strictly decreasing. Since $\mathcal{I}$ is a CPI where Start and Q are interpreted as linear polynomials, we get $\mathsf{Proc}_{\mathsf{RP}}(\langle \mathcal{P}_1|_{\mathcal{J}_2}, \mathcal{P}_1|_{\mathcal{J}_2}\rangle) = (\mathsf{Pol}_1, \langle \mathcal{P}_1|_{\mathcal{J}_2}, \{(28)\}\rangle)$. However, there is no polynomial interpretation which orients (28) strictly and the other ADPs weakly. Thus, we need another processor to solve the remaining problem.

4.4 Knowledge Propagation Processor

The dependency graph can not only be used to decompose an ADP problem $\langle \mathcal{P}, \mathcal{S}\rangle$ according to the SCC-prefixes via the dependency graph processor, but it can also be used to remove an ADP α from $\mathcal{S}$ if all "*predecessors*" of α have already been taken into account. More precisely, let $\mathrm{Pre}(\alpha) \subseteq \mathcal{P}$ contain all ADPs that can "generate" a redex for a step with α at an annotated position, i.e., $\mathrm{Pre}(\alpha)$ consists of all ADPs $\beta \in \mathcal{P}$ such that there is an edge from some DP in $\mathrm{dp}^{\perp}(\beta)$ to some DP in $\mathrm{dp}^{\perp}(\alpha)$ in the $\mathcal{P}$-dependency graph. Note that $\mathrm{dp}^{\perp}(\alpha) \neq \varnothing$ for all ADPs α. If d is the maximal number of annotated symbols in any term on a right-hand side of an ADP from $\mathcal{P}$, then in any $\mathcal{P}$-chain tree $\mathfrak{T}$, the probabilities of α-steps can be over-approximated as follows. Except for the very first step, every (**at**)- or (**af**)-step with α is preceded by a step with some ADP β from $\mathrm{Pre}(\alpha)$. Every term in β's right-hand side can trigger at most d α-steps. If the β-step had probability p, then adding all probabilities for these α-steps yields at most $d \cdot p$. Since the very first step of the tree might also be an α-step, one obtains

$$\begin{aligned}\mathrm{edl}_{\langle \mathcal{P},\{\alpha\}\rangle}(\mathfrak{T}) &\leq 1 + \textstyle\sum_{v \in V^{\mathfrak{T}} \setminus \mathrm{Leaf}^{\mathfrak{T}},\ \mathcal{P}(v) \in \mathrm{Pre}(\alpha) \times \{(\mathbf{at}),(\mathbf{af})\}} d \cdot p_v \\ &= 1 + d \cdot \mathrm{edl}_{\langle \mathcal{P},\mathrm{Pre}(\alpha)\rangle}(\mathfrak{T}).\end{aligned}$$

This in turn implies $\mathrm{erc}_{\langle \mathcal{P},\{\alpha\}\rangle}(n) \leq 1 + d \cdot \mathrm{erc}_{\langle \mathcal{P},\mathrm{Pre}(\alpha)\rangle}(n)$ for all $n \in \mathbb{N}$, and thus, $\iota_{\langle \mathcal{P},\{\alpha\}\rangle} \sqsubseteq \iota_{\langle \mathcal{P},\mathrm{Pre}(\alpha)\rangle}$.

Hence, if $\alpha \in \mathcal{S}$ and $\mathrm{Pre}(\alpha) \cap \mathcal{S} = \varnothing$ (i.e., $\mathrm{Pre}(\alpha) \subseteq \mathcal{P} \setminus \mathcal{S}$), then in any well-formed proof tree with a node v where $L_{\mathcal{A}}(v) = \langle \mathcal{P}, \mathcal{S}\rangle$, the ADPs from $\mathrm{Pre}(\alpha)$ have already been taken into account in the path $v_1, \ldots, v_k = v$ from the root node v_1 to v, i.e.,

$$\iota_{\langle \mathcal{P},\{\alpha\}\rangle} \sqsubseteq \iota_{\langle \mathcal{P},\mathrm{Pre}(\alpha)\rangle} \sqsubseteq \iota_{\langle \mathcal{P},\mathcal{P}\setminus\mathcal{S}\rangle} \sqsubseteq L_C(v_1) \oplus \cdots \oplus L_C(v_{k-1}).$$

As the proof tree already contains knowledge about $\mathrm{Pre}(\alpha)$'s complexity, the *knowledge propagation processor* removes α from $\mathcal{S}$.

Theorem 4.14 (Knowledge Propagation Proc.). *Let $\langle \mathcal{P}, \mathcal{S}\rangle$ be an ADP problem, let $\alpha \in \mathcal{S}$ and $\mathrm{Pre}(\alpha) \cap \mathcal{S} = \varnothing$, where $\mathrm{Pre}(\alpha)$ consists of all ADPs $\beta \in \mathcal{P}$ such that there is an edge from some DP in $\mathrm{dp}^{\perp}(\beta)$ to some DP in $\mathrm{dp}^{\perp}(\alpha)$ in the $\mathcal{P}$-dependency graph. Then the following processor is sound:*

$$\mathsf{Proc}_{\mathsf{KP}}(\langle \mathcal{P}, \mathcal{S}\rangle) = (\mathsf{Pol}_0, \langle \mathcal{P}, \mathcal{S} \setminus \{\alpha\}\rangle)$$

Example 4.15 (Knowledge Propagation Processor). We still have to solve the ADP problem $\langle \mathcal{P}_1|_{\mathcal{J}_2}, \{(28)\}\rangle$ from Ex. 4.13. We have $\mathrm{dp}^{\perp}((28)) = \{(18)\}$ and the only DPs with edges to (18) in the dependency graph of Fig. 4 are (14) and (17), where $(14) \in \mathrm{dp}^{\perp}((25))$ and $(17) \in \mathrm{dp}^{\perp}((27))$. Thus, $\mathrm{Pre}((28)) = \{(25),(27)\}$, i.e., in particular $(28) \notin \mathrm{Pre}((28))$. Hence, we can apply the knowledge propagation processor and obtain $\mathsf{Proc}_{\mathsf{KP}}(\langle \mathcal{P}_1|_{\mathcal{J}_2}, \{(28)\}\rangle) = (\mathsf{Pol}_0, \langle \mathcal{P}_1|_{\mathcal{J}_2}, \varnothing\rangle)$, again containing a solved ADP problem.

The solved proof tree is shown in Fig. 6. Thus, we inferred that $\mathcal{R}_1$ is SAST and its complexity is at most linear, i.e., $\iota_{\mathcal{R}_1} \sqsubseteq \mathsf{Pol}_1$.

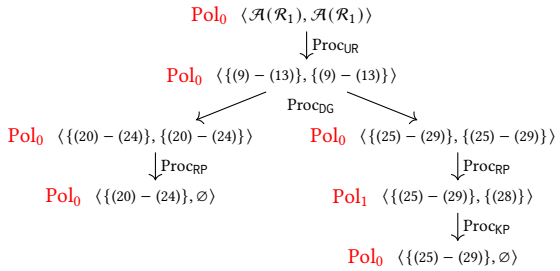


Figure 6: Solved proof tree for $\mathcal{R}_1$

4.5 Probability Removal Processor

Our framework may also yield ADP (sub)problems with non-probabilistic structure, i.e., where every ADP has the form $\ell \to \{1 : r\}^m$. Then, the *probability removal processor* can switch to ordinary (non-probabilistic) DT problems for complexity analysis from [54].

These DT problems have four components $(\mathcal{P}, \mathcal{S}, \mathcal{K}, \mathcal{R})$: a set of dependency tuples $\mathcal{P}$, the subset $\mathcal{S} \subseteq \mathcal{P}$ that is counted for complexity, a subset $\mathcal{K}$ whose complexity has already been taken into account (see Remark 3.21), and a set of rewrite rules $\mathcal{R}$.

Theorem 4.16 (Probability Removal Processor). *Let $\langle \mathcal{P}, \mathcal{S}\rangle$ be an ADP problem where every ADP in $\mathcal{P}$ has the form $\ell \to \{1 : r\}^m$. Let $\mathrm{dt}(\ell \to \{1 : r\}^m) = \ell^\sharp \to [t_1^\sharp, \ldots, t_n^\sharp]$ if $\{t \mid t \trianglelefteq_\sharp r\} = \{t_1, \ldots, t_n\}$, and let $\mathrm{dt}(\mathcal{P}) = \{\mathrm{dt}(\alpha) \mid \alpha \in \mathcal{P}\}$. Then the expected runtime complexity of $\langle \mathcal{P}, \mathcal{S}\rangle$ is equal to the runtime complexity of the non-probabilistic DT problem $\beta = (\mathrm{dt}(\mathcal{P}), \mathrm{dt}(\mathcal{S}), \mathrm{dt}(\mathcal{P} \setminus \mathcal{S}), \mathrm{np}(\mathcal{P}))$. So the processor $\mathsf{Proc}_{\mathsf{PR}}(\langle \mathcal{P}, \mathcal{S}\rangle) = (c, \varnothing)$ is sound if the DT framework returns c as an upper bound on the runtime complexity of β.*

When proving AST as in [36], one should move to the non-probabilistic DP framework for termination whenever possible, because then one can analyze function calls in right-hand sides of rules separately. In contrast, in the non-probabilistic DT framework for complexity analysis, one also has to consider all function calls from a right-hand side simultaneously. However, the switch to the non-probabilistic setting is still advantageous, since, e.g., the reduction pair processor of the non-probabilistic DT framework allows more orderings than just multilinear polynomial interpretations.

4.6 Further Processors

In the non-probabilistic setting, there are several further processors used to derive upper complexity bounds. For example, the DP and DT frameworks of [3, 21, 54] also provide processors that use narrowing, rewriting, or instantiations to *transform* DP/DT problems to simplify the task of proving termination or complexity. To show how to adapt such transformational processors for our novel ADP framework, we present a corresponding adapted *narrowing processor* in [38].

5 Evaluation

We implemented our new DP framework for upper bounds on the expected innermost runtime complexity in our termination prover AProVE [22]. To this end, AProVE first creates the canonical ADPs and then applies processors according to the following strategy:

First, we try to apply the dependency graph processor Proc_{DG}, the usable rules processor Proc_{UR}, the knowledge propagation processor Proc_{KP}, and the probability removal processor Proc_{PR} in this order. The advantage of these processors is that they do not rely on searching (i.e., they are very fast) and they simplify the ADP problem whenever they are applicable. If none of these processors can be applied anymore, then we search for CPIs for the reduction pair processor Proc_{RP} in order to derive polynomial complexity bounds for certain ADPs (otherwise, we try to apply Proc_{RP} with a non-CPI polynomial interpretation to derive an exponential bound). As soon as one of the processors is applied successfully, we re-start the strategy, since other processors might be applicable again on the simplified subproblems. Moreover, before the first application of the reduction pair processor, we use our adaption of the narrowing processor. Since it does not always help in inferring an upper bound and often increases the number of ADPs, we use this processor only once on a fixed number of terms.

For every PTRS, the user can indicate whether one wants to analyze termination or complexity, consider arbitrary or only basic start terms, and whether one wants to analyze innermost or full rewriting (with an arbitrary rewrite strategy). Since our novel DP framework only works for innermost rewriting and basic start terms, if the user asks for complexity analysis or SAST of full rewriting, we check whether the PTRS belongs to a known class where, e.g., upper bounds on the expected innermost runtime complexity are upper bounds w.r.t. an arbitrary rewrite strategy as well. Such properties were studied in [34, 35]. If one wants to consider arbitrary instead of basic start terms, we perform the transformation of [20] (adapted to PTRSs, see [34, 35]) in order to move from derivational to runtime complexity, i.e., the PTRS $\mathcal{R}$ is transformed into a new PTRS $\mathcal{R}'$ such that the complexity of $\mathcal{R}'$ on basic start terms is a bound on the complexity of $\mathcal{R}$ on all start terms.

For our evaluation, we used the benchmark set of all 128 PTRSs from the *Termination Problem Data Base* [56], i.e., the benchmarks considered for the annual *Termination and Complexity Competition* [23], containing 128 typical probabilistic programs, including examples with complicated probabilistic structure and probabilistic algorithms on lists and trees. Note that this set also contains many examples that are AST, but not SAST. Therefore, we extended the collection by 10 additional examples that are interesting for expected complexity analysis (including all examples from our paper).

To evaluate our novel framework for proving SAST, we compare AProVE with its previous version (called "POLO" in Table 2) whose only way to prove SAST was to search for a monotonic, multilinear polynomial interpretation such that all rules of the PTRS are strictly decreasing [7], and with the tool NaTT [58] that implements polynomial and matrix interpretations to prove SAST. Thus, POLO and NaTT neither consider a specific rewrite strategy nor start terms. As shown by the experiments on all 138 PTRSs in Table 2, our novel ADP framework increases the power of proving SAST significantly. Here, the numbers indicate for how many PTRSs the respective tool proved SAST. Note that as usual for benchmarks in termination or complexity analysis, the termination behavior or runtime complexity of the benchmarks is in general not known, i.e., the best quantitative performance measure for a tool is the comparison to other tools.

Strategy	Start Terms	POLO	NaTT	AProVE
Full	Arbitrary	30	33	35
Full	Basic	30	33	44
Innermost	Arbitrary	30	33	54
Innermost	Basic	30	33	62

Table 2: Number of SAST proofs found by each tool

Strategy	Start Terms	Pol_0	Pol_1	Pol_2	Exp	2-Exp	ω
Full	Arbitrary	2	21	0	11	1	103
Full	Basic	15	25	1	3	0	94
Innermost	Arbitrary	2	47	0	5	0	84
Innermost	Basic	25	35	0	2	0	76

Table 3: Runtime complexities proven by AProVE

Table 3 shows the upper bounds inferred by AProVE. So AProVE obtains numerous constant and/or linear bounds, even for full rewriting and/or arbitrary start terms. Note that in contrast to the non-probabilistic setting, a PTRS with expected constant runtime is not necessarily trivial as it can have evaluations of unbounded (and even infinite) length. However, the transformation of [20] to move from arbitrary to basic start terms may add rules with linear runtime to the PTRS. This explains the low number of constant upper bounds for arbitrary start terms. Due to the restriction to multilinear polynomial interpretations, Pol_a for $a > 1$ can currently only be inferred from multilinear, but non-linear interpretations like $\mathcal{I}_{f^\sharp}(x, y) = x \cdot y$. In the future, we intend to extend our implementation to also use, e.g., matrix orderings [17] in order to improve the inference of polynomial bounds of higher degrees.

For more details on our experiments, the collection of examples, and for instructions on how to run our implementation in AProVE via its *web interface* or locally, we refer to:

https://aprove-developers.github.io/PTRSExpectedRuntime/

6 Conclusion

In this paper, we presented the first DP framework to infer upper bounds on the expected innermost runtime complexity of PTRSs automatically. Our implementation in AProVE is the first tool for automatic complexity analysis of PTRSs and it improves substantially over previous tools to analyze SAST of PTRSs.

There are several directions for future work, e.g., by extending the reduction pair processor of Thm. 4.12 to other orderings, by adapting further transformational processors to our new ADP framework, and by developing variants of our framework that are directly applicable for full instead of innermost rewriting and/or for arbitrary instead of basic start terms.

Acknowledgments

This work is supported by the DFG Research Training Group 2236 UnRAVeL.

References

[1] Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. 2021. Learning Probabilistic Termination Proofs. In *Proc. CAV '21 (LNCS 12760)*. 3–26. doi:10.1007/978-3-030-81688-9_1

[2] Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. 2018. Lexicographic Ranking Supermartingales: An Efficient Approach to Termination of Probabilistic Programs. *Proc. ACM Program. Lang.* 2, POPL (2018). doi:10.1145/3158122

[3] Thomas Arts and Jürgen Giesl. 2000. Termination of Term Rewriting Using Dependency Pairs. *Theor. Comput. Sc.* 236, 1-2 (2000), 133–178. doi:10.1016/S0304-3975(99)00207-8

[4] Martin Avanzini and Georg Moser. 2016. A Combination Framework for Complexity. *Inf. Comput.* 248 (2016), 22–55. doi:10.1016/j.ic.2015.12.007

[5] Martin Avanzini, Georg Moser, and Michael Schaper. 2016. TcT: Tyrolean Complexity Tool. In *Proc. TACAS '16 (LNCS 9636)*. 407–423. doi:10.1007/978-3-662-49674-9_24

[6] Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. 2019. Type-Based Complexity Analysis of Probabilistic Functional Programs. In *Proc. LICS '19*. doi:10.1109/LICS.2019.8785725

[7] Martin Avanzini, Ugo Dal Lago, and Akihisa Yamada. 2020. On Probabilistic Term Rewriting. *Sci. Comput. Program.* 185 (2020). doi:10.1016/j.scico.2019.102338

[8] Martin Avanzini, Georg Moser, and Michael Schaper. 2020. A Modular Cost Analysis for Probabilistic Programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). doi:10.1145/3428240

[9] Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press. doi:10.1017/CBO9781139172752

[10] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Lena Verscht. 2023. A Calculus for Amortized Expected Runtimes. *Proc. ACM Program. Lang.* 7, POPL (2023), 1957–1986. doi:10.1145/3571260

[11] Raven Beutner and Luke Ong. 2021. On Probabilistic Termination of Functional Programs with Continuous Distributions. In *Proc. PLDI '21*. 1312–1326. doi:10.1145/3453483.3454111

[12] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. 2001. Algorithms with Polynomial Interpretation Termination Proof. *Journal of Functional Programming* 11, 1 (2001), 33–53. doi:10.1017/S0956796800003877

[13] Olivier Bournez and Claude Kirchner. 2002. Probabilistic Rewrite Strategies. Applications to ELAN. In *Proc. RTA '02 (LNCS 2378)*. 252–266. doi:10.1007/3-540-45610-4_18

[14] Olivier Bournez and Florent Garnier. 2005. Proving Positive Almost-Sure Termination. In *Proc. RTA '05 (LNCS 3467)*. 323–337. doi:10.1007/978-3-540-32033-3_24

[15] Krishnendu Chatterjee, Hongfei Fu, and Petr Novotný. 2020. Termination Analysis of Probabilistic Programs with Martingales. In *Foundations of Probabilistic Programming*, Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). Cambridge University Press, 221–258. doi:10.1017/9781108770750.008

[16] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, Jiri Zárevúcky, and Dorde Zikelic. 2023. On Lexicographic Proof Rules for Probabilistic Termination. *Formal Aspects Comput.* 35, 2 (2023). doi:10.1145/3585391

[17] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. 2008. Matrix Interpretations for Proving Termination of Term Rewriting. *J. Autom. Reason.* 40, 2-3 (2008), 195–220. doi:10.1007/S10817-007-9087-9

[18] Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Proc. POPL '15*. 489–501. doi:10.1145/2676726.2677001

[19] Hongfei Fu and Krishnendu Chatterjee. 2019. Termination of Nondeterministic Probabilistic Programs. In *Proc. VMCAI '19 (LNCS 11388)*. 468–490. doi:10.1007/978-3-030-11245-5_22

[20] Carsten Fuhs. 2019. Transforming Derivational Complexity of Term Rewriting to Runtime Complexity. In *Proc. FroCoS '19 (LNCS 11715)*. 348–364. doi:10.1007/978-3-030-29007-8_20

[21] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. 2006. Mechanizing and Improving Dependency Pairs. *J. Autom. Reason.* 37, 3 (2006), 155–203. doi:10.1007/s10817-006-9057-7

[22] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. *J. Autom. Reason.* 58, 1 (2017), 3–31. doi:10.1007/s10817-016-9388-y

[23] Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. 2019. The Termination and Complexity Competition. In *Proc. TACAS '19 (LNCS 11429)*. 156–166. doi:10.1007/978-3-030-17502-3_10 Website of *TermComp*: https://termination-portal.org/wiki/Termination_Competition.

[24] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *Proc. FOSE '14*. 167–181. doi:10.1145/2593882.2593900

[25] Raúl Gutiérrez and Salvador Lucas. 2020. MU-TERM: Verify Termination Properties Automatically (System Description). In *Proc. IJCAR '20 (LNCS 12167)*. 436–447. doi:10.1007/978-3-030-51054-1_28

[26] Nao Hirokawa and Aart Middeldorp. 2005. Automating the Dependency Pair Method. *Inf. Comput.* 199, 1-2 (2005), 172–199. doi:10.1016/j.ic.2004.10.004

[27] Nao Hirokawa and Georg Moser. 2008. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. IJCAR '08 (LNCS 5195)*. 364–379. doi:10.1007/978-3-540-71070-7_32

[28] Dieter Hofbauer and Clemens Lautemann. 1989. Termination Proofs and the Length of Derivations (Preliminary Version). In *Proc. RTA '89 (LNCS 355)*. 167–177. doi:10.1007/3-540-51081-8_107

[29] Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. 2019. Modular Verification for Almost-Sure Termination of Probabilistic Programs. *Proc. ACM Program. Lang.* 3, OOPSLA (2019). doi:10.1145/3360555

[30] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2018. Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms. *J. ACM* 65, 5 (2018). doi:10.1145/3208102

[31] Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2020. Expected Runtime Analyis by Program Verification. In *Foundations of Probabilistic Programming*, Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). Cambridge University Press, 185–220. doi:10.1017/9781108770750.007

[32] Jan-Christoph Kassing and Jürgen Giesl. 2023. Proving Almost-Sure Innermost Termination of Probabilistic Term Rewriting Using Dependency Pairs. In *Proc. CADE '23 (LNCS 14132)*. 344–364. doi:10.1007/978-3-031-38499-8_20

[33] Jan-Christoph Kassing, Grigory Vartanyan, and Jürgen Giesl. 2024. A Dependency Pair Framework for Relative Termination of Term Rewriting. In *Proc. IJCAR '24 (LNCS 14740)*. 360–380. doi:10.1007/978-3-031-63501-4_19

[34] Jan-Christoph Kassing, Florian Frohn, and Jürgen Giesl. 2024. From Innermost to Full Almost-Sure Termination of Probabilistic Term Rewriting. In *Proc. FoSSaCS '24 (LNCS 14575)*. 206–228. doi:10.1007/978-3-031-57231-9_10

[35] Jan-Christoph Kassing and Jürgen Giesl. 2024. From Innermost to Full Probabilistic Term Rewriting: Almost-Sure Termination, Complexity, and Modularity. *CoRR* abs/2409.17714 (2024). doi:10.48550/arXiv.2409.17714 Long version of [34].

[36] Jan-Christoph Kassing, Stefan Dollase, and Jürgen Giesl. 2024. A Complete Dependency Pair Framework for Almost-Sure Innermost Termination of Probabilistic Term Rewriting. In *Proc. FLOPS '24 (LNCS 14659)*. 62–80. doi:10.1007/978-981-97-2300-3_4

[37] Jan-Christoph Kassing and Jürgen Giesl. 2024. Annotated Dependency Pairs for Full Almost-Sure Termination of Probabilistic Term Rewriting. In *Principles of Verification (LNCS 15260)*. 339–366. doi:10.1007/978-3-031-75783-9_14

[38] Jan-Christoph Kassing, Leon Spitzer, and Jürgen Giesl. 2025. Dependency Pairs for Expected Innermost Runtime Complexity and Strong Almost-Sure Termination of Probabilistic Term Rewriting. *CoRR* abs/2507.12918 (2025). doi:10.48550/arXiv.2507.12918

[39] Andrew Kenyon-Roberts and C.-H. Luke Ong. 2021. Supermartingales, Ranking Functions and Probabilistic Lambda Calculus. In *Proc. LICS '21*. doi:10.1109/LICS52264.2021.9470550

[40] Naoki Kobayashi, Ugo Dal Lago, and Charles Grellois. 2020. On the Termination Problem for Probabilistic Higher-Order Recursive Programs. *Log. Methods Comput. Sci.* 16, 4 (2020). doi:10.23638/LMCS-16(4:2)2020

[41] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. 2009. Tyrolean Termination Tool 2. In *Proc. RTA '09 (LNCS 5595)*. 295–304. doi:10.1007/978-3-642-02348-4_21

[42] Ugo Dal Lago and Charles Grellois. 2019. Probabilistic Termination by Monadic Affine Sized Typing. *ACM Trans. Program. Lang. Syst.* 41, 2 (2019). doi:10.1145/3293605

[43] Ugo Dal Lago, Claudia Faggian, and Simona Ronchi Della Rocca. 2021. Intersection Types and (Positive) Almost-Sure Termination. *Proc. ACM Program. Lang.* 5, POPL (2021). doi:10.1145/3434313

[44] Ugo Dal Lago and Luca Padovani. 2024. On the Almost-Sure Termination of Binary Sessions. In *Proc. PPDP '24*. 1–12. doi:10.1145/3678232.3678239

[45] Dallas S. Lankford. 1979. *On Proving Term Rewriting Systems are Noetherian.* Memo MTP-3, Math. Dept. Louisiana Technical University, Ruston, LA. http://www.ens-lyon.fr/LIP/REWRITING/TERMINATION/Lankford_Poly_Term.pdf

[46] Lorenz Leutgeb, Georg Moser, and Florian Zuleger. 2022. Automated Expected Amortised Cost Analysis of Probabilistic Data Structures. In *Proc. CAV '22 (LNCS 13372)*. 70–91. doi:10.1007/978-3-031-13188-2_4

[47] Nils Lommen, Éléanore Meyer, and Jürgen Giesl. 2024. Control-Flow Refinement for Complexity Analysis of Probabilistic Programs in KoAT (Short Paper). In *Proc. IJCAR '24 (LNCS 14739)*. 233–243. doi:10.1007/978-3-031-63498-7_14

[48] Rupak Majumdar and V. R. Sathiyanarayana. 2024. Positive Almost-Sure Termination: Complexity and Proof Rules. *Proc. ACM Program. Lang.* 8, POPL (2024), 1089–1117. doi:10.1145/3632879

[49] Rupak Majumdar and V. R. Sathiyanarayana. 2025. Sound and Complete Proof Rules for Probabilistic Termination. *Proc. ACM Program. Lang.* 9, POPL (2025), 1871–1902. doi:10.1145/3704899

[50] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. 2018. A New Proof Rule for Almost-Sure Termination. *Proc. ACM Program. Lang.* 2, POPL (2018). doi:10.1145/3158121

[51] Fabian Meyer, Marcel Hark, and Jürgen Giesl. 2021. Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes. In *Proc. TACAS '21 (LNCS 12651)*. 250–269. doi:10.1007/978-3-030-72016-2_14

[52] Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. 2021. Automated Termination Analysis of Polynomial Probabilistic Programs. In *Proc. ESOP '21 (LNCS 12648)*. 491–518. doi:10.1007/978-3-030-72019-3_18

[53] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Proc. PLDI '18*. 496–512. doi:10.1145/3192366.3192394

[54] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. 2013. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *J. Autom. Reason.* 51 (2013), 27–56. doi:10.1007/s10817-013-9277-6

[55] Vineet Rajani, Gilles Barthe, and Deepak Garg. 2024. A Modal Type Theory of Expected Cost in Higher-Order Probabilistic Programs. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 389–414. doi:10.1145/3689725

[56] TPDB. 2025. Termination Problem Data Base. https://github.com/TermCOMP/TPDB-ARI

[57] Di Wang, David M. Kahn, and Jan Hoffmann. 2020. Raising Expectations: Automating Expected Cost Analysis with Types. *Proc. ACM Program. Lang.* 4, ICFP (2020), 31 pages. doi:10.1145/3408992

[58] Akihisa Yamada, Keiichirou Kusakari, and Toshiki Sakabe. 2014. Nagoya Termination Tool. In *Proc. RTA-TLCA '14 (LNCS 8560)*. 466–475. doi:10.1007/978-3-319-08918-8_32

Realisability and Complementability of Multiparty Session Types

Cinzia Di Giusto
Etienne Lozes
Pascal Urso
cinzia.di-giusto@univ-cotedazur.fr
etienne.lozes@univ-cotedazur.fr
pascal.urso@univ-cotedazur.fr
Université Côte d'Azur, CNRS, I3S
France

Abstract

Multiparty session types (MPST) are a type-based approach for specifying message-passing distributed systems. They rely on the notion of global type, specifying the global behaviour, and local types, which are the projections of the global behaviour onto each local participant. An essential property of global types is realisability, i.e., whether the composition of the local behaviours conforms to those specified by the global type. We explore how realisability of MPST relates to their complementability, i.e., whether there exists a global type that describes the complementary behaviour of the original global type. First, we show that if a global type is realisable with p2p communications, then it is realisable with synchronous communications. Second, we show that if a global type is realisable in the synchronous model, then it is complementable, in the sense that there exists a global type that describes the complementary behaviour of the original global type. Third, we give an algorithm to decide whether a complementable global type, given with an explicit complement, is realisable in p2p. The algorithm is PSPACE in the size of the global type and its complement. As a side contribution, we propose a complementation construction for global types with sender-driven choice with a linear blowup in the size of the global type.

CCS Concepts: • **Theory of computation** → *Regular languages*; **Process calculi**; **Type theory**; **Program analysis**.

Keywords: Multiparty session types, deadlock-freedom, synchronizability, message-sequence-charts

ACM Reference Format:
Cinzia Di Giusto, Etienne Lozes, and Pascal Urso . 2025. Realisability and Complementability of Multiparty Session Types. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The design of communication protocols for concurrent and distributed systems is a central concern in programming languages research. Modern systems – from microservices to IoT networks – rely on structured sequences of message exchanges between multiple parties. In the family of behavioural types [20], multiparty session types (MPST) were introduced by Honda et al. [14, 15, 22] and have emerged as a powerful methodology to specify and verify such structured interactions. An MPST defines a global type describing the communication protocol as a whole, which can be projected into local types for each participant.

This type-based approach enables static verification of communication safety (no unexpected messages) and liveness properties (no deadlocks) by ensuring each party's code conforms to its local type. In essence, MPSTs allow protocol designers to treat communication patterns as first-class abstractions that yield a rigorous framework for building correct distributed applications.

In this work, we focus on the realisability problem, i.e., whether the local types projected from a global type can be recomposed into a concrete system that conforms to the intended global protocol. In classical MPST theory, protocols are often formulated under the sender-driven choice assumption: only one designated participant (the sender) decides between alternative message flow branches for all parties. While this assumption simplifies theoretical analysis, it proves too restrictive for many real-world protocols. We lift this assumption and build a theory for deterministic, unrestricted choice.

Local types are often defined as systems of communicating finite state machines [6], following the works of Villard [21] and Denielou and Yoshida [9]: to every global type G, one can associate a system of communicating finite state machines (CFSMs) $\mathrm{proj}(G)$ whose machines are the projections of G onto each participant. In some recent works [19], global types may also be seen as a special case of Higher-order Message Sequence Charts [2] (HMSCs), an automaton model that accepts languages of message sequence charts (MSCs). Adopting such a point of view, every global type G defines a language of MSCs $\mathcal{L}^{\exists}_{\mathsf{msc}}(G)$ which provides a semantics of G. In this work, we study two notions on MPSTs coming from these connections with HMSCs and automata theory: deadlock-free realisability, and complementability.

Deadlock-free realisability, introduced by Alur et al. [2] for HMSCs, is central to the goal of *correct-by-construction* protocol design underpinning the MPST approach. Deadlock-free realisability ensures that $\mathrm{proj}(G)$ does not suffer from communication mismatches, deadlocks, and behaviours that are not part of $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$. In the literature on MPSTs, this property has appeared under various names, including *implementability* and *projectability*, with some slight variations in the formal definition of this notion.

The second notion, complementability, asks whether a global type G admits a complementary one $\overline{G}$ that gives all those behaviours that are not described by G, i.e., for all synchronous-like MSCs M,

$$M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\overline{G}) \quad \Leftrightarrow \quad M \notin \mathcal{L}^{\exists}_{\mathsf{msc}}(G).$$

We also study whether some complementation procedure exists that explicitly constructs $\overline{G}$ from G, and with which complexity. These problems are popular in automata theory, e.g., it is well know that pushdown automata are not complementable in general, but their restriction to visibly pushdown automata [3] is complementable, Büchi automata are complementable [7], and there has been a long quest for efficient complementation procedures and optimisations for subclasses of Büchi automata.

Contributions. In this work, we approach the realisability question for two fundamental communication models for implementing MPST protocols: the peer-to-peer (p2p) model, and the synchronous one. In the p2p model, processes communicate via point-to-point message passing over FIFO channels.In the synchronous model, communications are based on rendez-vous (handshakes where send and receive happen simultaneously), representing a globally synchronised message-passing semantics. These two models are widely considered as canonical extremes in analysing distributed protocols.

In this work, we also investigate for the first time the question of the complementability of global types. Our main contribution is to show that realisable global types form a subclass of the complementable ones. More in detail, we establish the following results:

- Every global type realisable in the synchronous communication model is complementable (Theorem 4.5), and the complementation is effective (based on projection and NFA complementation), with a doubly exponential blowup in the size of the global type.
- Realisability is decidable for complementable global types provided an explicit complement is also given as input, both in the synchronous communication model (Theorem 5.1) and in the peer-to-peer model (Theorem 5.4). The latter result is non-trivial and based on recent results on RSC systems [12].
- We also show that if a global type is realisable in the peer-to-peer model, then it is also realisable in the synchronous one (Theorem 5.2), which is not obvious, as deadlock-freedom is not a property that is monotonic in the communication model (a more permissive communication model may both hide some deadlocks and trigger new ones).
- Finally, we show that not all global types are complementable (Theorem 4.2) and we give procedures to find the complement of certain classes: global types with at most three participants are complementable (Corollary 4.4), as well as global types with sender-driven choices (Theorem 4.13). For the latter, we provide a complementation procedure that is linear in the size of the global type.

Outline. The paper is organised as follows: Section 2 introduces the necessary background on executions, Message Sequence Charts, communication models and communicating finite state machines. MPST and realisability are introduced in Section 3. Then Section 4 discusses the issue of complementability, discussing an example of a non-complementable global type as well as several complementation procedures. Finally, Section 5 establishes the decidability of realisability for complementable global types. Section 6 concludes with some final remarks and discusses related works. Omitted proofs and can be found in [10].

2 Preliminaries

We assume a finite set of *processes* $\mathbb{P} = \{p, q, \ldots\}$ and a finite set of messages (labels) $\mathbb{M} = \{m, \ldots\}$. We consider two kinds of actions: *send actions* that are of the form $!m^{p\to q}$ and are executed by process p sending message m to q; *receive actions* that are of the form $?m^{p\to q}$ and are executed by q receiving m from p. We write Act for the finite set $\mathbb{P}\times\mathbb{P}\times\{!,?\}\times\mathbb{M}$ of all actions, and Act_p for the set of actions that can be executed by p (i.e., $!m^{p\to q}$ or $?m^{q\to p}$). We omit processes when they are clear from the context and simply write $!m$ or $?m$ for a send or receive action, respectively.

An *event* η of a sequence of actions $w \in \mathrm{Act}^*$, is an index i in $\{1, \ldots, \mathsf{length}(w)\}$; i is a send (resp. receive) event of w

if $w[i]$ is a send (resp. receive) action. We write $\mathsf{events}_S(w)$ (resp. $\mathsf{events}_R(w)$) for the set of send (resp. receive) events of w and $\mathsf{events}(w) = \mathsf{events}_S(w) \cup \mathsf{events}_R(w)$ for the set of all events of w. When all events are labeled with distinct actions, we identify an event with its action.

Executions. An execution is a well defined sequence of actions $e \in \mathsf{Act}^*$, where a receive action is always preceded by a unique corresponding send action.

Definition 2.1 (Execution)**.** An *execution* over $\mathbb{P}$ and $\mathbb{M}$ is a sequence of actions $e \in \mathsf{Act}^*$ where an injective function from receive events to send events $\mathsf{src}_e : \mathsf{events}_R(w) \to \mathsf{events}_S(w)$ is such that for each receive event i labeled with $?m^{p\to q}$, $\mathsf{src}_e(i)$ is labeled with $!m^{p\to q}$, and $\mathsf{src}_e(i) < i$.

For a set of executions $\mathcal{E}$, we write $\mathsf{Prefixes}(\mathcal{E})$ for the set of all prefixes of the executions in $\mathcal{E}$. We say that an execution e_2 is a *completion* of an execution e_1 if e_1 is a prefix of e_2. A *concatenation* $e_1 \cdot w$ of an execution e_1 and a sequence of actions w is the execution $e_2 = e_1 \cdot w_2$ where e_2 is a completion of e_1 (note that w is not an execution, since it may contain receive events which sources are in e_1). The *projection* $\mathsf{proj}_p(e)$ of an execution e on a process p is the subsequence of actions in Act_p. A send event s is *matched* if there is a receive event r such that $s = \mathsf{src}(r)$. An execution e is *orphan-free* if src is surjective over the send events of e, i.e., all send events are matched.

Communication Models. In this paper, we consider two communication models: peer-to-peer (p2p) and synchronous (synch). However, as commented in the conclusions, this work is part of a bigger more general project and a large amount of the results of the paper can be extended to "any" communication model, or only require few assumptions. In this perspective, we introduce here a general definition of a communication model and discuss specific results regarding how it can be extended to the general case.

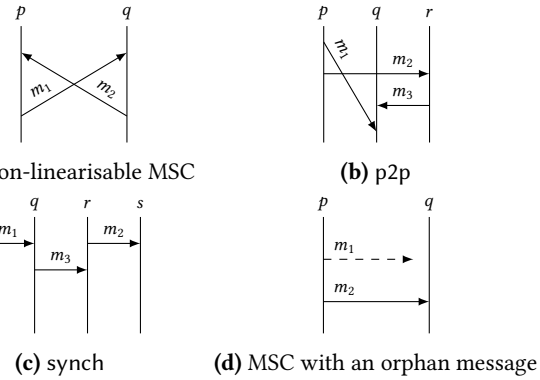
Definition 2.2 (Communication model)**.** A *communication model* com is a set $\mathcal{E}_{\mathsf{com}}$ of executions.

In the synchronous communication model synch, message exchanges can be thought as rendezvous synchronisations. In other words, an execution e belongs to $\mathcal{E}_{\mathsf{synch}}$ if all send events are immediately followed by their corresponding receive events.

Definition 2.3 (synch)**.** An execution $e = (w, \mathsf{src}) \in \mathcal{E}_{\mathsf{synch}}$ if for all send event $s \in \mathsf{events}_S(e)$, $s+1$ is a receive event of e and $\mathsf{src}(s+1) = s$.

In the peer-to-peer communication model p2p, messages sent by a process p to q transit over a FIFO channel that is specific to the pair (p, q): if p sends first m_1 then m_2 to q, then m_2 cannot overtake m_1 in the FIFO channel. In particular:

- if m_1 is not received, then m_2 is not received either;
- if both are received, then m_1 is received before m_2.



(a) non-linearisable MSC **(b)** p2p

(c) synch **(d)** MSC with an orphan message

Figure 1. Examples of MSCs. The sender of a message is at the origin of the arrow and the receiver at the destination. Unmatched send events are depicted with dashed arrows.

Definition 2.4 (p2p)**.** $\mathcal{E}_{\mathsf{p2p}}$ is the set of executions e such that for any two send events $s_1 = !m_1^{p\to q}$ and $s_2 = !m_2^{p\to q}$ in $\mathsf{events}_S(e)$, with $s_1 < s_2$, one of the two holds:

- s_2 is unmatched, or
- there exist r_1, r_2 such that $r_1 < r_2$, $\mathsf{src}(r_1) = s_1$, and $\mathsf{src}(r_2) = s_2$.

Note that, $\mathcal{E}_{\mathsf{synch}} \subset \mathcal{E}_{\mathsf{p2p}}$. Moreover, if e is an execution in p2p, then src_e is defined as follows: the source of the i-th receive event of q from p is the i-th send event of p to q. If e is an execution in synch, then src_e is defined as follows: for all receive event i, $\mathsf{src}_e(i) = i - 1$.

Message Sequence Charts. While executions correspond to a total order view of the events occurring in a system, message sequence charts (MSC) adopt a distributed and partial order view on the events. For a tuple $M = (w_p)_{p\in\mathbb{P}}$, each $w_p \in \mathsf{Act}_p^*$ is a sequence of actions representing the ones executed by process p according to some total, locally observable, order. We write $\mathsf{events}(M)$ for the set $\{(p, i) \mid p \in \mathbb{P} \text{ and } 0 \le i < \mathsf{length}(w_p)\}$. The label $\mathsf{action}(\eta)$ of the event $\eta = (p, i)$ is the action $w_p[i]$. The event η is a send (resp. receive) event if it is labeled with a send (resp. receive) action. We write $\mathsf{events}_S(M)$ (resp. $\mathsf{events}_R(M)$) for the set of send (resp. receive) events of M; we also write $\mathsf{msg}(\eta)$ for the message sent or received at event η, and $\mathsf{proc}(\eta)$ for the process executing η. Finally, we write $\eta_1 \prec_{\mathsf{proc}} \eta_2$ if there is a process p and $i < j$ such that $\eta_1 = (p, i)$ and $\eta_2 = (p, j)$.

Definition 2.5 (MSC)**.** An MSC over $\mathbb{P}$ and $\mathbb{M}$ is a tuple $M = \big((w_p)_{p\in\mathbb{P}}, \mathsf{src}\big)$ where

1. for each process p, $w_p \in \mathsf{Act}_p^*$ is a finite sequence of actions;
2. $\mathsf{src} : \mathsf{events}_R(M) \to \mathsf{events}_S(M)$ is an injective function from receive events to send events such that for all receive event η labeled with $?m^{p\to q}$, $\mathsf{src}(\eta)$ is labeled with $!m^{p\to q}$.

For an execution e, $\mathsf{msc}(e)$ is the MSC $\big((w_p)_{p\in\mathbb{P}}, \mathsf{src}\big)$ where w_p is the subsequence of e restricted to the actions of p, and src is the lifting of src_e to the events of $(w_p)_{p\in\mathbb{P}}$.

Example 2.6. MSC M in Fig. 1a is an MSC over $\mathbb{P} = \{p, q\}$ and $\mathbb{M} = \{m_1, m_2\}$ with $M = ((w_p, w_q), \mathsf{src})$, $w_p = ?m_2!m_1$, $w_q = ?m_1!m_2$, $\mathsf{src}((p,0)) = (q,1)$, and $\mathsf{src}((q,0)) = (p,1)$. Note that there is no execution e such that $M = \mathsf{msc}(e)$ as all receptions precede the corresponding sends. On the other hand, the MSC in Fig. 1b is $\mathsf{msc}(e)$ for the execution $e = (!m_1!m_2?m_2!m_3?m_3?m_1, \mathsf{src})$ where $\mathsf{src}(3) = 2$, $\mathsf{src}(5) = 2$ and $\mathsf{src}(6) = 4$. This is the only execution that induces this MSC, but in general there might exist several executions inducing the same MSC.

For a set of processes $\mathbb{P}$, an MSC $M = \big((w_p)_{p\in\mathbb{P}}, \mathsf{src}\big)$ is the *prefix* of another MSC $M' = \big((w'_p)_{p\in\mathbb{P}}, \mathsf{src}'\big)$, in short $M \leq_{\mathsf{pref}} M'$, if for all $p \in \mathbb{P}$, w_p is a prefix of w'_p and $\mathsf{src}'(e) = \mathsf{src}(e)$ for all receive events e of M. The *concatenation* of MSCs M_1 and M_2 is the MSC $M_1 \cdot M_2$ obtained by gluing "vertically" M_1 before M_2: formally, if $M_1 = ((w^1_p)_{p\in\mathbb{P}}, \mathsf{src}_1)$ and $M_2 = ((w^2_p)_{p\in\mathbb{P}}, \mathsf{src}_2)$, then $M_1 \cdot M_2 = ((w_p)_{p\in\mathbb{P}}, \mathsf{src})$ where for all p, $w_p = w^1_p \cdot w^2_p$, and src is defined by $\mathsf{src}(e) = \mathsf{src}_i(e)$ for all receive events e of M_i $(i = 1, 2)$.

Happens-before relation and linearisations. In a given MSC M, an event η happens before η', if (i) η and η' are events of a same process p and happen in that order on the time line of p, or (ii) η is send event matched by η', or (iii) a sequence of such situations defines a path from η to η'.

Definition 2.7 (Happens-before relation)**.** Let M be an MSC. The happens-before relation over M is the binary relation $\prec_M$ defined as the least transitive relation over $\mathsf{events}(M)$ such that:

- for all p, i, j, if $i < j$, then $(p, i) \prec_M (p, j)$, and
- for all receive events η, $\mathsf{src}(\eta) \prec_M \eta$.

Definition 2.8 (Linearisation)**.** A *linearisation* of an MSC M is a total order $\ll$ on $\mathsf{events}(M)$ that refines $\prec_M$: for all events η, η', if $\eta \prec_M \eta'$, then $\eta \ll \eta'$.

We write $\mathsf{lin}(M)$ for the set of all linearisations of M. We often identify a linearisation with the execution it induces.

Example 2.9. Let M be the MSC in Fig. 1c. Then $!m_1$ happens before $?m_1$, which happens before $!m_3$, and both $!m_3$ and $!m_2$ happen before $?m_3$. Moreover, $\prec_M$ is a partial order, and $!m_1!m_2?m_1!m_3?m_3?m_2 \in \mathsf{lin}(M_c)$. On the other hand, consider the MSC M in in Fig. 1a; then $\prec_{M'}$ is not a partial order, because $?m_2 \prec_{M'} !m_1 \prec_{M'} ?m_1 \prec_{M'} !m_2 \prec_{M'} ?m_2$, therefore $\mathsf{lin}(M') = \emptyset$.

Given an MSC M, we write $\mathsf{lin}_{\mathsf{com}}(M)$ to denote $\mathsf{lin}(M) \cap \mathcal{E}_{\mathsf{com}}$; the executions of $\mathsf{lin}_{\mathsf{com}}(M)$ are called the linearisations of M in the communication model com.

Definition 2.10 (com-linearisable MSC)**.** An MSC M is *linearisable* in a communication model com if $\mathsf{lin}_{\mathsf{com}}(M) \neq \emptyset$. We write $\mathcal{M}_{\mathsf{com}}$ for the set of all MSCs linearisable in com.

Example 2.11. The MSC M_b in Fig. 1b is linearisable in p2p, with $\mathsf{lin}_{\mathsf{p2p}}(M_b) = \{!m_1!m_2?m_2!m_3?m_3?m_1\}$. However, M_b is not linearisable in synch. Finally the MSC M_c in Fig. 1c is linearisable in synch with $\mathsf{lin}_{\mathsf{synch}}(M_c) =$

$$\{\ !m_1?m_1!m_2?m_2!m_3?m_3\ ,\ !m_2?m_2!m_1?m_1!m_3?m_3\ \}$$

Finally, we introduce a property that will be helpful in the next paragraph for giving an alternative characterisation of deadlock-freedom of a system of communicating finite state machines.

Definition 2.12 (Causally-closed communication model)**.** A communication model com is *causally-closed* if for all MSCs M, $\mathsf{lin}_{\mathsf{com}}(M) \neq \emptyset$ implies that $\mathsf{lin}_{\mathsf{com}}(M) = \mathsf{lin}(M)$.

Observe that not all communication models are causally closed. It is the case for p2p, but it is immediate to see that the property is not valid for synch. Take for instance MSC M in Fig. 1c, its linearisation $!m_1!m_2?m_1?m_2!m_3?m_3$ does not belong to $\mathsf{lin}_{\mathsf{synch}}(M_c)$. To show that p2p is causally closed, notice that $<$ can be replaced with $\preceq_M$ in Definition 2.4.

Lemma 2.13. *Let e be an execution and $M = \mathsf{msc}(e)$. Then e is an execution of* p2p *if and only if for any two send events $s_1 = !m_1^{p\to q}$ and $s_2 = !m_2^{p\to q}$ in $\mathsf{events}_S(e)$, with $s_1 \preceq_M s_2$, one of the two holds:*

- *s_2 is unmatched, or*
- *there exist r_1, r_2 such that $r_1 \preceq_M r_2$, $\mathsf{src}(r_1) = s_1$, and $\mathsf{src}(r_2) = s_2$.*

Proof. Assume that e is a p2p execution. Let $s_1 = !m_1^{p\to q}$ and $s_2 = !m_2^{p\to q}$ be two send events in $\mathsf{events}_S(e)$ such that $s_1 \preceq_M s_2$. Then $s_1 < s_2$ in e, because $<$ is a linearisation of $\preceq_M$. By definition of p2p, s_2 is unmatched or there exists r_1, r_2 such that $r_1 < r_2$, $\mathsf{src}(r_1) = s_1$, and $\mathsf{src}(r_2) = s_2$. In the second case, $r_1 < r_2$ implies that $r_1 \preceq_M r_2$, because both r_1 and r_2 occur on the same process q. Conversely, assume that e and M satisfy the above reformulation of the definition of p2p. Let $s_1 = !m_1^{p\to q}$ and $s_2 = !m_2^{p\to q}$ be two send events in $\mathsf{events}_S(e)$ such that $s_1 < s_2$. Then $s_1 \preceq_M s_2$ because s_1 and s_2 occur on the same process p. By the reformulation of the definition of p2p, s_2 is unmatched or there exists r_1, r_2 such that $r_1 \preceq_M r_2$, $\mathsf{src}(r_1) = s_1$, and $\mathsf{src}(r_2) = s_2$. In the second case, $r_1 \preceq_M r_2$ implies that $r_1 < r_2$ because $<$ is a linearisation of $\preceq_M$. □

Lemma 2.14. *$\mathcal{E}_{\mathsf{p2p}}$ is causally-closed.*

Proof. Let $M \in \mathcal{M}_{\mathsf{p2p}}$ and e be a linearisation of M. We want to show that $e \in \mathcal{E}_{\mathsf{p2p}}$. By definition of $\mathcal{M}_{\mathsf{p2p}}$, there is an execution $e' \in \mathcal{E}_{\mathsf{p2p}}$ such that $\mathsf{msc}(e') = M$. By Lemma 2.13, e is also a p2p execution. □

Communicating finite state machines. We assume standard notations for automata, words and languages. As usual, a non-deterministic finite state automaton (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, l_0, F)$ where Q is a set of control states, Σ is an alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the transition relation, l_0 is the initial control state, and $F \subseteq Q$ is the set of accepting states. The language $\mathcal{L}_{\mathsf{words}}(\mathcal{A})$ of an NFA $\mathcal{A}$ and the notion of deterministic finite state automaton (DFA) or ε transitions are defined as usual. We write $\widehat{\mathcal{A}}$ for the automaton obtained from $\mathcal{A}$ by setting $F = Q$. We start by recalling the definition of communicating finite state machines [6].

Definition 2.15 (CFSM)**.** A communicating finite state machine (CFSM) is an NFA with ε-transitions $\mathcal{A}$ over the alphabet Act. A system of CFSMs is a tuple $\mathcal{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$.

Given a system of CFSMs $\mathcal{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$, we write $\widehat{\mathcal{S}}$ for the system of CFSMs $\widehat{\mathcal{S}} = (\widehat{\mathcal{A}_p})_{p\in\mathbb{P}}$ where all states are accepting, i.e., $F_p = Q_p$.

Definition 2.16 (Executions of a CFSMs in com)**.** Given a system $\mathcal{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ of CFSMs, and a communication model com, $\mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S})$ is the set of *executions* $e \in \mathcal{E}_{\mathsf{com}}$ such that for all p, $\mathsf{proj}_p(e)$ is in $\mathcal{L}_{\mathsf{words}}(\mathcal{A}_p)$.

Remark 2.17. Let com be a communication model, $\mathcal{S}$ a system of CFSMs, and $e, e' \in \mathcal{E}_{\mathsf{com}}$ such that $\mathsf{msc}(e) = \mathsf{msc}(e')$, then $e \in \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S})$ if and only if $e' \in \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S})$. This follows from the fact that $\mathsf{proj}_p(e) = \mathsf{proj}_p(e')$ for all p.

We write $\mathcal{L}^{\mathsf{com}}_{\mathsf{msc}}(\mathcal{S})$ for the set $\{\mathsf{msc}(e) \mid e \in \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S})\}$.

A system is orphan-free if, whenever all machines have reached an accepting state, no message remains in transit, i.e., no message is sent but not received.

Definition 2.18 (Orphan-free)**.** A system of CFSMs $\mathcal{S}$ is *orphan-free* for a communication model com if for all $e \in \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S})$, e is orphan-free.

All synchronous executions are orphan-free by definition.

A system is deadlock-free if, any *partial* execution can be extended/completed to an accepting execution.

Definition 2.19 (Deadlock-free)**.** A system of CFSMs $\mathcal{S}$ is *deadlock-free* in a communication model com if for all $e \in \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\widehat{\mathcal{S}})$, there is an execution $e' \in \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S})$ such that $e \leq_{\mathsf{pref}} e'$.

Remark 2.20. A system of CFSMs $\mathcal{S}$ is *deadlock-free* for a communication model com if and only if

$$\mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\widehat{\mathcal{S}}) \subseteq \mathsf{Prefixes}(\mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S}))$$

The following result shows that, for either p2p or synch communication models, the deadlock-freedom property of a system of CFSMs can be expressed as a property on the MSCs of the system.

Proposition 2.21 (Deadlock-freedom as an MSC property)**.** *Assume* com *is* p2p *(respectively,* com *is* synch*). Then a system of CFSMs $\mathcal{S}$ is deadlock-free for* com *if and only if*

$$\mathcal{L}^{\mathsf{com}}_{\mathsf{msc}}(\widehat{\mathcal{S}}) \subseteq \mathsf{Prefixes}(\mathcal{L}^{\mathsf{com}}_{\mathsf{msc}}(\mathcal{S})).$$

3 Global Types

In this section, we define global types. We deviate from the standard definition to allow for more liberal behaviours. In our setting, global types are automata that describe a language of MSCs. An *arrow* is a triple $(p, q, m) \in \mathbb{P} \times \mathbb{P} \times \mathbb{M}$ with $p \neq q$; we often write $p \xrightarrow{m} q$ instead of (p, q, m), and write Arr to denote the finite set of arrows.

Definition 3.1 (Global Type)**.** A global type G is a deterministic finite state automaton over the alphabet Arr.

The projection of a global type G on a process p is the CFSM G_p obtained by replacing each arrow $q \xrightarrow{m} r$ of a transitions of G by the corresponding action of p (either $!m^{p\to r}$ if $p = q$, or $?m^{q\to p}$ if $p = r$, or ε otherwise).

Definition 3.2 (Projected system of CFSMs)**.** The projected system of CFSMs proj(G) associated to a global type G is the tuple $(\mathrm{G}_p)_{p\in\mathbb{P}}$.

Conversely, for every system, we can associate a global type computing its synchronous product. Let $\mathcal{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of CFSMs, where $\mathcal{A}_p = (L_p, \mathrm{Act}_p, \delta_p, l_{0,p}, F_p)$ is the CFSM associated to process p. The *synchronous product* of $\mathcal{S}$ is the NFA $\mathrm{preprod}(\mathcal{S}) = (L, \mathrm{Arr}, \delta, l_0, F)$, where $L = \Pi_{p\in\mathbb{P}} L_p$, $l_0 = (l_{0,p})_{p\in\mathbb{P}}$, $F = \Pi_{p\in\mathbb{P}} F_p$, and δ is the transition relation defined by $(\vec{l}, p \xrightarrow{m} q, \vec{l'}) \in \delta$ if $(l_p, !m^{p\to q}, l'_p) \in \delta_p$, $(l_q, ?m^{p\to q}, l'_q) \in \delta_q$, and $l'_r = l_r$ for all $r \notin \{p, q\}$.

Note that, in general, $\mathrm{preprod}(\mathcal{S})$ is not a global type, as it may be non-deterministic. However, one can compute its determinisation $\mathrm{prod}(\mathrm{G}) \stackrel{\mathrm{def}}{=} \det(\mathrm{preprod}(\mathcal{S}))$ by standard powerset construction obtaining a global type.

We write $\mathsf{exec}(w)$ to denote the execution obtained by replacing each arrow $p \xrightarrow{m} q$ of w with $!m^{p\to q} \cdot ?m^{p\to q}$. We write $\mathsf{msc}(w)$ to denote the MSC $\mathsf{msc}(\mathsf{exec}(w))$.

A global type defines a language of MSCs in two different ways, one existential and one universal. Let $\mathcal{L}_{\mathsf{words}}(\mathrm{G})$ be the set of sequences of arrows w accepted by G. Note that for $w \in \mathrm{Arr}^*$, the function $w \mapsto \mathsf{msc}(w)$ with $\mathsf{msc}(w) \in \mathcal{M}_{\mathsf{synch}}$ is not injective, as two arrows with disjoint pairs of processes commute. We write $w_1 \sim w_2$ if $\mathsf{msc}(w_1) = \mathsf{msc}(w_2)$, and $[w]$ for the equivalence class of w wrt $\sim$. The existential MSC language $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$ of a global type G is the set of MSCs that admit at least one representation as a sequence of arrows in $\mathcal{L}_{\mathsf{words}}(\mathrm{G})$, and the universal MSC language $\mathcal{L}^{\forall}_{\mathsf{msc}}(\mathrm{G})$ of a global type G is the set of MSCs whose representations as a sequence of arrows are all in $\mathcal{L}_{\mathsf{words}}(\mathrm{G})$:

$$\begin{aligned}\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}) &\stackrel{\mathrm{def}}{=} \{\mathsf{msc}(w) \mid w \in \mathcal{L}_{\mathsf{words}}(\mathrm{G})\}\\ \mathcal{L}^{\forall}_{\mathsf{msc}}(\mathrm{G}) &\stackrel{\mathrm{def}}{=} \{\mathsf{msc}(w) \mid [w] \subseteq \mathcal{L}_{\mathsf{words}}(\mathrm{G})\}.\end{aligned}$$

Definition 3.3 (Commutation-closed)**.** A global type G is *commutation-closed* if $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}) = \mathcal{L}^{\forall}_{\mathsf{msc}}(\mathrm{G})$.

In that case, we write $\mathcal{L}^{\exists/\forall}_{\mathsf{msc}}(\mathrm{G})$ for the common language. As an example, any global type with $\#\mathbb{P} \leq 3$ is commutation-closed, as any two arrows share at least one process and therefore do not commute.

Proposition 3.4. *For all global type* G*,*

$$\mathcal{L}^{\forall}_{\mathsf{msc}}(\mathrm{G}) \subseteq \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}) \subseteq \mathcal{L}^{\mathsf{synch}}_{\mathsf{msc}}(\mathsf{proj}(\mathrm{G})).$$

Proof. (Sketch) The first inclusion is immediate from the definitions. For the second inclusion, let $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$, we show that $M \in \mathcal{L}^{\mathsf{synch}}_{\mathsf{msc}}(\mathsf{proj}(\mathrm{G}))$. By definition of $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$, there is a word $w \in \mathcal{L}_{\mathsf{words}}(\mathrm{G})$ such that $\mathsf{msc}(w) = M$. Let ρ be an accepting run of G for w. For every $p \in \mathbb{P}$, let ρ_p denote the run of G_p (the machine of p in $\mathsf{proj}(\mathrm{G})$) obtained by projecting ρ: note that we kept ε-transitions in G_p, see Definition 3.2, so ρ_p is obviously defined. Then ρ_p is an accepting run of G_p, therefore $M \in \mathcal{L}^{\mathsf{synch}}_{\mathsf{msc}}(\mathrm{G})$ (by Definition 2.16). □

Lemma 3.5. *Let* G_1 *and* G_2 *be two global types, with* G_2 *commutation-closed. Then*

$$\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}_1 \otimes \mathrm{G}_2) = \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}_1) \cap \mathcal{L}^{\exists/\forall}_{\mathsf{msc}}(\mathrm{G}_2).$$

Proposition 3.6. *For all system* $\mathcal{S}$ *of communicating finite state machines,* $\mathsf{prod}(\mathcal{S})$ *is commutation-closed and*

$$\mathcal{L}^{\mathsf{synch}}_{\mathsf{msc}}(\mathcal{S}) = \mathcal{L}^{\exists/\forall}_{\mathsf{msc}}(\mathsf{prod}(\mathcal{S})).$$

Proof. (Sketch) The runs of the synchronous executions

$$!m_1?m_1 \ldots !m_k?m_k$$

of $\mathcal{S}$ are in one-to-one correspondence with the runs of the words $m_1 \ldots m_k$ of $\mathsf{preprod}(\mathcal{S})$. □

When a global type is implemented in a concrete system, its behaviour depends on the chosen communication model.

Definition 3.7 (Global Type Language)**.** Let G be a global type and com a communication model. The language of G in com is $\mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathrm{G}) \stackrel{\mathsf{def}}{=} \bigcup\{\mathsf{lin}_{\mathsf{com}}(M) \mid M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})\}$.

Realisability. We have finally collected all the definitions needed to introduce our notion of realisability of global types that is parametric in a given communication model.

Definition 3.8 (Deadlock-free realisability)**.** A global type G is *deadlock-free realisable*[1] in the communication model com if the following two conditions hold:

1. $\mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathsf{proj}(\mathrm{G})) = \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathrm{G})$
2. $\mathsf{proj}(\mathrm{G})$ is deadlock-free in com.

[1]We sometimes say simply *realisable* instead of *deadlock-free realisable.*

Condition 1 of Definition 3.8 corresponds to *global type conformance*: the executions of the projected system do not deviate from the ones specified by the global type.

Deadlock-free realisability is equivalent to the notion of *safe realisability* of [2] when com is p2p or synch. This is not the case for other communication models. Our definition better captures the fact that a key factor for realisability is deadlock-freedom which is strongly dependent on the communication model being causally-closed. The following proposition is easily proved.

Proposition 3.9 (Global type conformance as an MSC property)**.** *Let* $\mathcal{E}_{\mathsf{com}} \supseteq \mathcal{E}_{\mathsf{synch}}$*, Condition 1 of Definition 3.8 is equivalent to* $\mathcal{L}^{\mathsf{com}}_{\mathsf{msc}}(\mathsf{proj}(\mathrm{G})) \subseteq \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$.

4 Complementability

In this section, we introduce the notion of complementability for global types, then we present some complementation procedures for some notable subclasses of global types, namely commutation-closed global types, synch-realisable global-types, and global types with sender-driven choice. We start with the definition of complementable global type:

Definition 4.1 (Complement of a Global Type)**.** A global type $\overline{\mathrm{G}}$ is a complement of G if $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}) = \mathcal{M}_{\mathsf{synch}} \setminus \mathcal{L}^{\exists}_{\mathsf{msc}}(\overline{\mathrm{G}})$. We say that G is *complementable* if it admits at least one complement.

Theorem 4.2. *Not all global types are complementable.*

Proof. Let $\mathbb{P} = \{p, q, r, s\}$, $\mathbb{M} = \{m_1, m_2, m_3\}$, and $\mathsf{Arr} = \{p \xrightarrow{m_1} q, r \xrightarrow{m_2} s, p \xrightarrow{m_3} q\}$. Consider a global type G_0 such that $\mathcal{L}_{\mathsf{words}}(\mathrm{G}_0) = (m_1 + m_2)^*(m_2 + m_3)^*$ and take the minimal DFA recognising this language. Note that G_0 is commutation-closed and the MSCs of $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}_0)$ are of the form depicted in Fig. 3.

Now consider the global type G depicted in Fig. 2. We claim that for all natural numbers k_1, k_2, k_3, the following two are equivalent:

- $\mathsf{msc}(m_1^{k_1} m_2^{k_2} m_3^{k_3}) \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$;
- $k_1 > k_2$ or $k_2 > k_3$.

The claim follows by analysis on the paths leading to accepting states in G, as hinted by Fig. 2. For instance, the language of the words accepted by G with state q_8 as the last state is of the form $m_1^+(m_2m_1)^*m_3^+$, therefore lead to MSCs with $k_1 > k_2$.

We show that G is not complementable. By contradiction, suppose that G is complementable and G′ is a complement. Let $\mathrm{G}'' = \mathrm{G}' \otimes \mathrm{G}_0$; by Lemma 3.5, and the fact that G_0 is commutation-closed,

$$\begin{aligned} \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}'') &= \mathcal{L}^{\exists/\forall}_{\mathsf{msc}}(\mathrm{G}_0) \cap \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}') \\ &= \{\mathsf{msc}(m_1^{k_1} m_2^{k_2} m_3^{k_3}) \mid k_1 \leq k_2 \leq k_3\}. \end{aligned}$$

Now, let $\mathcal{A}$ denote the NFA obtained from G′′ after replacing each m_2 transition with an ε transition. Then $\mathcal{L}_{\mathsf{words}}(\mathcal{A}) =$

$\{m_1^{k_1} m_3^{k_3} \mid k_1 \leq k_3\}$, which is not a regular language, hence the contradiction. □

4.1 Complementation By Duality

We write dual(G) for the dual DFA of a global type G, where accepting states and non-accepting ones are swapped (possibly completing first G with a sink state). It follows from the definition of $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$ and $\mathcal{L}^{\forall}_{\mathsf{msc}}(\mathrm{G})$ that

$$\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{dual}(\mathrm{G})) = \mathcal{M}_{\mathsf{synch}} \setminus \mathcal{L}^{\forall}_{\mathsf{msc}}(\mathrm{G}).$$

In general dual(G) is not a complement of G; still duality can be used to obtain a complement of G in a few cases. The first and most obvious case is the following.

Proposition 4.3. *If* G *is commutation-closed, then* dual(G) *is a complement of* G.

As observed before, all global types with less than three participants are commutation closed, hence from Lemma 4.3 we have the following immediate result.

Corollary 4.4. *If* $|\mathbb{P}| \leq 3$, *then* dual(G) *is a complement of* G.

Now, even if a global type G is not commutation-closed, it may be "enlarged" as the commutation-closed global type prod(proj(G)), a process sometimes called Cartesian abstraction [4]. This commutation-closed global type can later be complemented by duality to an under-approximation of a complement of G. When G is deadlock-free realisable in synch, this under-approximation is exact.

Theorem 4.5. *If a global type* G *is deadlock-free realisable in* synch, *then* dual(prod(proj(G))) *is a complement of* G.

Proof. Let G be a global type that is deadlock-free realisable in synch. By Condition (CC) of Definition 3.8,

$$\mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(\mathsf{proj}(\mathrm{G})) = \mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(\mathrm{G}).$$

If two sets of executions are equal, the corresponding sets of their MSCs are also equal, and thus

$$\mathcal{L}^{\mathsf{synch}}_{\mathsf{msc}}(\mathsf{proj}(\mathrm{G})) = \mathcal{L}^{\mathsf{synch}}_{\mathsf{msc}}(\mathrm{G}).$$

By Proposition 3.6, the synchronous product prod(proj(G)) is a commutation-closed global type whose runs are exactly the synchronous executions of the projected system, so

$$\mathcal{L}^{\exists/\forall}_{\mathsf{msc}}(\mathsf{prod}(\mathsf{proj}(\mathrm{G}))) = \mathcal{L}^{\mathsf{synch}}_{\mathsf{msc}}(\mathsf{proj}(\mathrm{G})) = \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}).$$

Finally, by Proposition 4.3, dual(prod(proj(G))) is a complement of prod(proj(G)), thus of G. □

Remark 4.6. In terms of complexity, complementation by duality is linear in the number of states of the global type G (possibly after adding a sink state). However, prod(proj(G)) involves in the worst case a doubly exponential blowup. This may be mitigated if we can avoid to explicitly compute prod(proj(G)).

4.2 Complementation by Renunciation for Sender-Driven Choices

In this section, we introduce another complementation procedure for global types with sender-driven choices. This construction is also linear. Stutz et al. introduced this assumption on global types in order to prove that p2p-implementability (a notion close to realisability) is decidable [16].

Let G be a global type with sender-driven choice. For a state s of G, let $\mathsf{sender}_{\mathrm{G}}(s)$ be the process p driving the choices in s, i.e., all outgoing transitions of s are labeled with arrows of the form $p \xrightarrow{m} q$ for some q, m; note that $\mathsf{sender}_{\mathrm{G}}(s)$ is undefined if s has no outgoing transitions.

Definition 4.7 (Complementation of a sender-driven global type)**.** Let G be a global type with sender-driven choices. Let S be the set of states of G. renun(G) is the global type with set of states $S \cup \{\overline{s} \mid s \in S\} \cup \{s_{\mathsf{acc}}, s_{\mathsf{rej}}\}$ defined as follows:

- the initial state of renun(G) is the initial state of G;
- a state s of G is accepting in renun(G) if and only if s is non-accepting in G;
- for all state s of G, $\overline{s}$ is accepting;
- s_{acc} is accepting, s_{rej} is non-accepting;
- for any two states s, s' of G, $(s, p \xrightarrow{m} q, s')$ is a transition in renun(G) if and only if it is a transition in G;
- for any state s of G, $(s, p \xrightarrow{m} q, \overline{s})$ (resp. $(\overline{s}, p \xrightarrow{m} q, \overline{s})$) is a transition in renun(G) if and only if
  - $p \xrightarrow{m} q$ is not labeling an outgoing transition of s,
  - $\mathsf{sender}_{\mathrm{G}}(s)$ is defined, and
  - $\mathsf{sender}_{\mathrm{G}}(s) \notin \{p, q\}$;
- for any state s of G, $(s, p \xrightarrow{m} q, s_{\mathsf{acc}})$ (resp. $(\overline{s}, p \xrightarrow{m} q', s_{\mathsf{acc}})$) is a transition in renun(G) if and only if
  - $p \xrightarrow{m} q$ is not labeling an outgoing transition of s in G, and
  - $\mathsf{sender}_{\mathrm{G}}(s)$ is either undefined or in $\{p, q\}$,
- for any state s of G, $(\overline{s}, p \xrightarrow{m} q, s_{\mathsf{rej}})$ is a transition of renun(G) if and only if $p \xrightarrow{m} q$ is labeling an outgoing transition of s in G;
- $(s_{\mathsf{acc}}, p \xrightarrow{m} q, s_{\mathsf{acc}})$ is a transition in renun(G) for all p, q, m;
- $(s_{\mathsf{rej}}, p \xrightarrow{m} q, s_{\mathsf{rej}})$ is a transition in renun(G) for all p, q, m.

Example 4.8. Fig. 4 depicts a sender-driven global type and the complement computed according to Definition 4.7.

It is easy to verify that renun(G) is deterministic, hence a global type. Note that renun(G) is also complete. Moreover, the number of states of renun(G) is linear in the number of states of G. Finally, observe that using Boolean expressions to label transitions, the size of renun(G) can also be kept linear in the size of G.

Intuitively, renun(G) describes a MSC M that starts with a prefix that is in $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$ up to a certain state s, at which

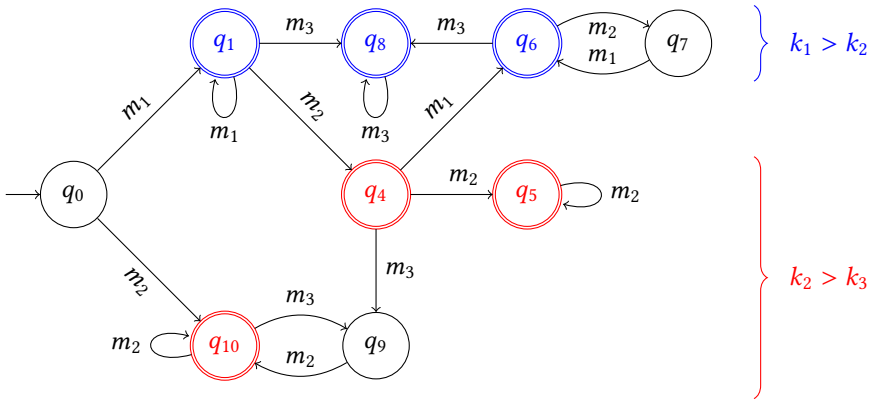


Figure 2. A non-complementable global type

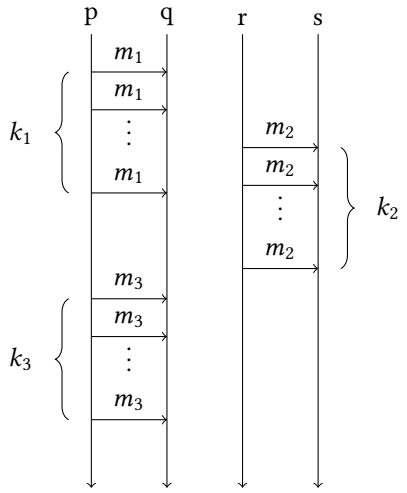


Figure 3. The shape of the MSCs in $\mathcal{L}^{\exists/\forall}_{\mathsf{msc}}(\mathrm{G}_0)$ for the global type G_0.

point a "renunciation" of the sender-driven choices of G at s occurs. Once renunciating to the choices proposed in a state s, we move to state $\overline{s}$, and remain there either until the end of the MSC, or until an arrow $p \xrightarrow{m} q$ with $\mathsf{sender}_{\mathrm{G}}(s) \in \{p, q\}$ is encountered, in which case renunciation is definitive and we move to the accepting state s_{acc}. The subtle part to be understood is that, as long as such an arrow is not encountered, the renunciation is not definitive, and an arrow $p \xrightarrow{m} q$ proposed in the choices at s could still "cancel" the renunciation; for this reason, we also have the transition $(\overline{s}, p \xrightarrow{m} q, s_{\mathsf{rej}})$.

Example 4.9. Let G, $\mathsf{renun}(\mathrm{G}), M_1, M_2$ be the global types and MSCs depicted in Fig. 4. It is easy to verify that $M_2 \notin \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{renun}(\mathrm{G}))$, because the only sequence of arrows $w_2 = p \xrightarrow{a_2} q'$ such that $M_2 = \mathsf{msc}(w_2)$ is not in $\mathcal{L}_{\mathsf{words}}(\mathsf{renun}(\mathrm{G}))$. It can also be checked that $M_1 \notin \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{renun}(\mathrm{G}))$, because the only two sequences of arrows $w_{1,1} = p \xrightarrow{a_1} q \cdot r \xrightarrow{b} r'$ and $w_{1,2} = r \xrightarrow{b} r' \cdot p \xrightarrow{a_1} q$ such that $\mathsf{msc}(w_{1,1}) = \mathsf{msc}(w_{1,2}) = M_1$ are not in $\mathcal{L}_{\mathsf{words}}(\mathsf{renun}(\mathrm{G}))$ (note that for $w_{1,2}$, the run of the automaton ends in s_{rej}). The harder part is to show that all other MSCs are in $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{renun}(\mathrm{G}))$. Consider for instance the MSC $M_3 = \mathsf{msc}(p \xrightarrow{a_1} q \cdot p \xrightarrow{a_3} q \cdot r \xrightarrow{b} r')$. Then M_3 is accepted by $\mathsf{renun}(\mathrm{G})$ because the run of $\mathsf{renun}(\mathrm{G})$ on $w_{3,1} = p \xrightarrow{a_1} q \cdot p \xrightarrow{a_3} q \cdot r \xrightarrow{b} r'$ ends in s_{acc}. However, the other sequence of arrows $w_{3,2} = r \xrightarrow{b} r' \cdot p \xrightarrow{a_1} q \cdot p \xrightarrow{a_3} q$ such that $\mathsf{msc}(w_{3,2}) = M_3$ is not in $\mathcal{L}_{\mathsf{words}}(\mathsf{renun}(\mathrm{G}))$, because its run ends in s_{rej}. More generally, $\mathsf{renun}(\mathrm{G})$ is *not* commutation-closed, and for any MSC $M \notin \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$, the sequence of arrows w such that $M = \mathsf{msc}(w)$ and $w \in \mathcal{L}_{\mathsf{words}}(\mathsf{renun}(\mathrm{G}))$ should be carefully constructed. To do so, we pick the sequence of arrows with the longest prefix in $\mathsf{Prefixes}(\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}))$, and we "renunciate" to a sender-driven choice as late as possible.

For a sequence of arrows $w \in \mathsf{Arr}^*$, let $\mathsf{state}_{\mathrm{G},s}(w)$ denote the state of G reached after reading w starting from state s (recall that G is deterministic); note that $\mathsf{state}_{\mathrm{G},s}(w)$ may be undefined.

Intuitively, a state $\overline{s}$ of $\overline{\mathrm{G}}$ is reached after a word w when $\mathsf{msc}(w) \notin \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$. The difficult point is that we are not in position to accept any sequence of arrows that start with w: it might be the case that $\mathsf{msc}(w \cdot w')$ is in $\mathcal{L}^{\exists}_{\mathsf{msc}}(\overline{\mathrm{G}})$ for some sequence of arrows w'; still, if this is the case, the first arrow that involves p in w' would then necessarily be one of the arrows labelling an outgoing transition of s in G. For this reason, $\overline{\mathrm{G}}$ conservatively rejects $w \cdot w'$ when this is the case.

The reason why every $M \notin \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$ belongs to $\mathcal{L}^{\exists}_{\mathsf{msc}}(\overline{\mathrm{G}})$, is that *at least one* sequence of arrows w such that $\mathsf{msc}(w) = M$ is in $\mathcal{L}_{\mathsf{words}}(\overline{\mathrm{G}})$. More precisely, any sequence of arrows w such that the run of w on $\overline{\mathrm{G}}$ remains as long as possible in the states of G will be accepted.

In order to formalise this intuition, we define the notion of residuation.

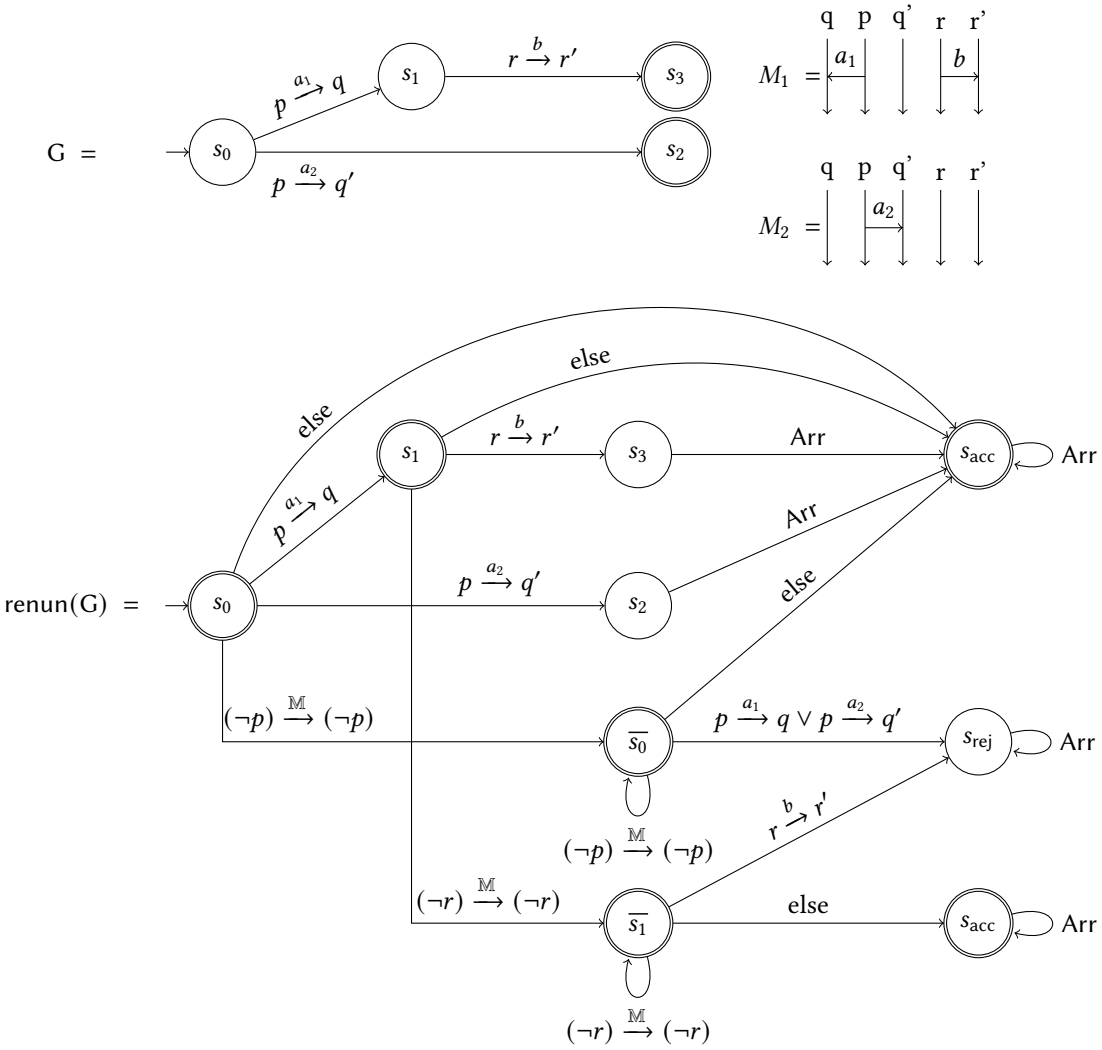


Figure 4. A sender-driven global type G, with $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}) = \{M_1, M_2\}$, and its complement renun(G) (for better readability, the state s_{acc} has been duplicated, the states $\overline{s_3}$ and $\overline{s_4}$, which are not reachable, have been omitted, and some transition labels have been summarised by Boolean expressions).

Definition 4.10 (Residuation)**.** Let G be a global type with sender-driven choice. The residuation $\mathsf{resid}^s_{\mathrm{G}}(M)$ of a MSC M from a state s of G is the pair (w, M_r) defined by induction on the number of events of M as follows.

1. if there are p, q, m, s_1, M_1 such that $M = \mathsf{msc}(p \xrightarrow{m} q) \cdot M_1$, and $(s, p \xrightarrow{m} q, s_1)$ is a transition of G, then $\mathsf{resid}^s_{\mathrm{G}}(M)$ is the pair $(p \xrightarrow{m} q \cdot w_1, M_r)$, where
$$\mathsf{resid}^{s_1}_{\mathrm{G}}(M_1) = (w_1, M_r);$$
2. otherwise, $\mathsf{resid}^s_{\mathrm{G}}(M) = (\varepsilon, M)$

The residuation of M with respect to G is $\mathsf{resid}^{s_0}_{\mathrm{G}}(M)$, where s_0 is the initial state of G.

Note that in condition 1 of Definition 4.10, there is at most one tuple (p, q, m, s_1, M_1) satisfying the condition, due to the sender-driven choice, which ensures that $\mathsf{resid}^s_{\mathrm{G}}(M)$ is uniquely defined.

Example 4.11. Consider G in Fig. 4, and $M_3 = \mathsf{msc}(r \xrightarrow{b} r' \cdot p \xrightarrow{a_1} q \cdot p \xrightarrow{a_3} q)$ as in Example 4.8. Then $\mathsf{resid}_{\mathrm{G}}(M) = (p \xrightarrow{a_1} q, M_r)$ where $M_r = \mathsf{msc}(p \xrightarrow{a_3} q \cdot r \xrightarrow{b} r')$.

When $\mathsf{resid}_{\mathrm{G}}(M) = (w, M_r)$, we call M_r the *residual* of M with respect to G, and w its *co-residual*. As we will later show, a sequence of arrows w such that $\mathsf{msc}(w) \notin \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$ is in $\mathcal{L}_{\mathsf{words}}(\overline{\mathrm{G}})$ if and only if the co-residual of $\mathsf{msc}(w)$ is a prefix of w.

Lemma 4.12. *Assume* $\mathsf{resid}_{\mathrm{G}}(M) = (w, M_r)$. *Then* $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$ *if and only if* M_r *is the empty MSC and* $w \in \mathcal{L}_{\mathsf{words}}(\mathrm{G})$.

Proof. For a state s of G, let $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}, s)$ (resp. $\mathcal{L}_{\mathsf{words}}(\mathrm{G}, s)$) denote the MSC language (resp. the language of arrows) accepted by G with s as initial state. Assume $\mathsf{resid}^s_{\mathrm{G}}(M) = (w, M_r)$. By induction on the number of events of M, we show that for all state s of G, $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}, s)$ if and only if M_r is the empty MSC and $w \in \mathcal{L}_{\mathsf{words}}(\mathrm{G}, s)$.

- Assume there are $p, q, m, s_1, M_1, w_1, M_r$ such that $M = \mathsf{msc}(p \xrightarrow{m} q) \cdot M_1$, and $(s, p \xrightarrow{m} q, s_1)$ is a transition of G, $\mathsf{resid}^s_{\mathrm{G}}(M) = (p \xrightarrow{m} q \cdot w_1, M_r)$, and $\mathsf{resid}^{s_1}_{\mathrm{G}}(M_1) = (w_1, M_r)$. By definition if $M_1 \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}, s_1)$, then $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}, s)$. By sender-driven choice hypothesis, conversely, if $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}, s)$, then $M_1 \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}, s_1)$. By induction hypothesis M_r is the empty MSC and $w_1 \in \mathcal{L}_{\mathsf{words}}(\mathrm{G}, s_1)$ if and only $M_1 \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}, s_1)$. As just observed, the later is equivalent to $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}, s)$; finaly, the former is equivalent to the fact that M_r is the empty MSC and $w \in \mathcal{L}_{\mathsf{words}}(\mathrm{G}, s)$ (because G is deterministic), which ends the proof.
- Assume now that Condition 1 of Definition 4.10 does not hold, and $\mathsf{resid}^s_{\mathrm{G}}(M) = (\epsilon, M)$. If M is the empty MSC and $\varepsilon \in \mathcal{L}_{\mathsf{words}}(\mathrm{G}, s)$, then $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}, s)$. Conversely, if $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}, s)$ and M is not empty then Condition 1 of Definition 4.10 should hold, and the contradiction. Finaly, if $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}, s)$, M is empty, but $\varepsilon \notin \mathcal{L}_{\mathsf{words}}(\mathrm{G}, s)$, we also get a contradiction.

□

Theorem 4.13. *Assume* G *is a global type with sender-driven choices, and let* $\mathsf{renun}(\mathrm{G})$ *denote the global type defined as in Def 4.7. Then* $\mathsf{renun}(\mathrm{G})$ *is a complement of* G.

Proof. Let G be a global type with sender-driven choice, and $\overline{\mathrm{G}}$ its complement as defined in Definition 4.7. Let $w \in \mathsf{Arr}^*$ be fixed. Let M, w_1, M_r, s, s_1, w_2 be such that $M = \mathsf{msc}(w)$, $\mathsf{resid}_{\mathrm{G}}(M) = (w_1, M_r)$, $\mathsf{state}_{\overline{\mathrm{G}}}(w) = s$, $\mathsf{state}_{\overline{\mathrm{G}}}(w_1) = s_1$, and w_2 is the longest prefix of w such that $\mathsf{state}_{\mathrm{G}}(w_2)$ is defined. The following properties are show by induction on the length of w.

1. If $w_1 \neq w_2$, then $s = s_{\mathsf{rej}}$;
2. if $w_1 = w_2$ and M_r is the empty MSC, then $s = s_1 = \mathsf{state}_{\mathrm{G}}(w)$ is a state of G;
3. if $w_1 = w_2$, M_r is not empty, and $\mathsf{sender}_{\mathrm{G}}(s_1)$ is undefined, then $s = s_{\mathsf{acc}}$;
4. if $w_1 = w_2$, M_r is not the empty MSC, and $\mathsf{sender}_{\mathrm{G}}(s_1)$ is undefined, then $s = s_{\mathsf{acc}}$;
5. if $w_1 = w_2$, M_r is not the empty MSC, $\mathsf{sender}_{\mathrm{G}}(s_1) = p$, and M_r contains no p events, then $s = \overline{s_1}$;
6. if $w_1 = w_2$, M_r is not the empty MSC, $\mathsf{sender}_{\mathrm{G}}(s_1) = p$, and the first p event of M_r is $?m^{q \to p}$, then $s = s_{\mathsf{acc}}$;
7. if $w_1 = w_2$, M_r is not the empty MSC, $\mathsf{sender}_{\mathrm{G}}(s_1) = p$, the first p event of M_r is $!m^{p \to q}$, and $p \xrightarrow{m} q$ does not label an outgoing transition of s_1 in G, then $s = s_{\mathsf{acc}}$.
8. if $w_1 = w_2$, M_r is not the empty MSC, $\mathsf{sender}_{\mathrm{G}}(s_1) = p$, the first p event of M_r is $!m^{p \to q}$, $p \xrightarrow{m} q$ labels an outgoing transition of s_1 in G, and $?m^{p \to q}$ is not the first event of q, then $s = s_{\mathsf{acc}}$.

From the definition of accepting states of $\overline{\mathrm{G}}$ and Lemma 4.12, it follows that w is accepted by $\overline{\mathrm{G}}$ if and only if $w_1 = w_2$ and $M \notin \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$.

In particular, if $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\overline{\mathrm{G}})$, then $M \notin \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$. Conversely, let $M \notin \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G})$ be fixed. There is a linearization of M of the form $w = w_1 \cdot w_r$ such that $\mathsf{resid}_{\mathrm{G}}(M) = (w_1, \mathsf{msc}(w_r))$. From the above properties, $\mathsf{state}_{\overline{\mathrm{G}}}(w)$ is accepting in $\overline{\mathrm{G}}$, therefore $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\overline{\mathrm{G}})$, which ends the proof. □

5 Decidability of Realisability

In this section, we show that the realisability problem for complementable global types, given with an explicit complement, is decidable in both the synchronous and the p2p model. We first show the result for the synchronous model, which is simpler, and then extend it to the p2p model using a characterisation of realisability in the p2p model.

5.1 Realisability in the synchronous model

Theorem 5.1. *Given a global type* G *and its complement* G′*, it is decidable in PSPACE complexity whether* G *is deadlock-free realisable in* synch.

Proof. (Sketch) Condition (CC) of Definition 3.8 is equivalent to

$$\mathcal{L}^{\exists/\forall}_{\mathsf{msc}}(\mathsf{prod}(\mathsf{proj}(\mathrm{G}))) \cap \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathrm{G}') = \emptyset.$$

By Lemma 3.5, this reduces to checking the emptiness of $\mathcal{L}_{\mathsf{words}}(\mathsf{prod}(\mathsf{proj}(\mathrm{G})) \otimes \mathrm{G}')$. Recall that

$$\mathsf{prod}(\mathcal{S}) = \mathsf{det}(\mathsf{preprod}(\mathcal{S}))$$

where $\mathsf{preprod}(\mathcal{S})$ denotes the synchronous product of the communicating machines keeping all ε-transitions and nondeterminism. It follows that Condition (CC) of Definition 3.8 is equivalent to checking the emptiness of the language of the NFA $\mathsf{preprod}(\mathsf{proj}(\mathrm{G})) \otimes \mathrm{G}'$. The non-emptiness of the language of an NFA can be checked in non-deterministic logarithmic space in the size of the NFA, which yields a PSPACE algorithm in that case, as the exponential size NFA $\mathsf{preprod}(\mathsf{proj}(\mathrm{G}))$ can be lazily constructed while guessing a path to an accepting state.

Assuming that Condition (CC) of Definition 3.8 holds, the second condition of Definition 3.8 is equivalent to whether all control states of $\mathsf{preprod}(\mathsf{proj}(\mathrm{G}))$ that are reachable from the initial state can reach a final state, which again, for the same reason, can be checked in PSPACE. □

5.2 Realisability in the p2p model

We now consider the realisability problem in the p2p model. Note that relaxing the communication model may both remove some deadlock situations, but also introduce some new

ones (because new configurations may be reachable). So realisability in the p2p model and realisability in the synch could be expected to be two different properties, none of which implies the other. Due to the the fact that global types enforce a form of synchronous-like behaviour, however, it turns out that the every global type that is realisable in the p2p model is also realisable in the synch model.

Theorem 5.2. *If* G *is deadlock-free realisable in* p2p*, then* G *is deadlock-free realisable in* synch.

We now show a "converse" of Theorem 5.2, which precisely states which additional conditions are needed to ensure that a global type that is deadlock-free realisable in the synch model is also deadlock-free realisable in the p2p model.

Theorem 5.3 (Reduction to synch-implementability)**.** *A global type* G *is deadlock-free realisable in* p2p *if and only if the following four conditions hold:*

1. $\mathcal{L}^{\text{p2p}}_{\text{msc}}(\text{proj}(\text{G})) \subseteq \text{Prefixes}(\mathcal{M}_{\text{synch}})$
2. $\text{proj}(\text{G})$ *is orphan-free in* p2p*,*
3. $\mathcal{L}^{\text{p2p}}_{\text{msc}}(\text{proj}(\widehat{\text{G}})) \subseteq \text{Prefixes}(\mathcal{L}^{\text{p2p}}_{\text{msc}}(\text{proj}(\text{G})))$*, and*
4. G *is deadlock-free realisable in* synch.

A system $\mathfrak{S}$ with $\mathcal{L}^{\text{p2p}}_{\text{msc}}(\mathfrak{S}) \subseteq \text{Prefixes}(\mathcal{M}_{\text{synch}})$ is called *realisable with synchronous communications* (RSC) in [12]. Germerie [12] showed that the RSC property is decidable and that whether a RSC system may reach a regular set of configurations is also decidable. As a consequence, conditions 1 and 2 of Theorem 5.3 are decidable. Assuming that $\text{proj}(\text{G})$ is RSC, $\mathcal{L}^{\text{p2p}}_{\text{msc}}(\text{proj}(\widehat{\text{G}}))$ can be coded as an effective regular set of executions, as well as $\text{Prefixes}(\mathcal{L}^{\text{p2p}}_{\text{msc}}(\text{proj}(\text{G})))$, and Condition 3 of Theorem 5.3 reduces to checking the inclusion of two NFAs, which can be checked in PSPACE using a lazy construction of the NFAs.

From these results we can conclude that it is decidable whether a given complementable global type, together with its complement, is deadlock-free realisable in the p2p model.

Theorem 5.4. *Given a complementable global type* G *and a complement* $\overline{\text{G}}$*, it is decidable whether* G *is deadlock-free realisable in* p2p.

6 Concluding remarks

In this paper, we investigated the realisability and complementability of MPSTs across two canonical communication models: synchronous and asynchronous (p2p). We reduce the task of checking realisability in the complex asynchronous setting to the simpler synchronous case, as long as the initial system satisfies key properties such as deadlock-freedom, absence of orphan messages, and a form of "synchronisability" (RSC property [12]). Said differently, we showed that global types that are realisable in the p2p model form a recursive subclass of the ones realisable in synch. We also showed that this latter class itself is a subclass of the class of global types that are equivalent (in terms of MSC semantics) to a

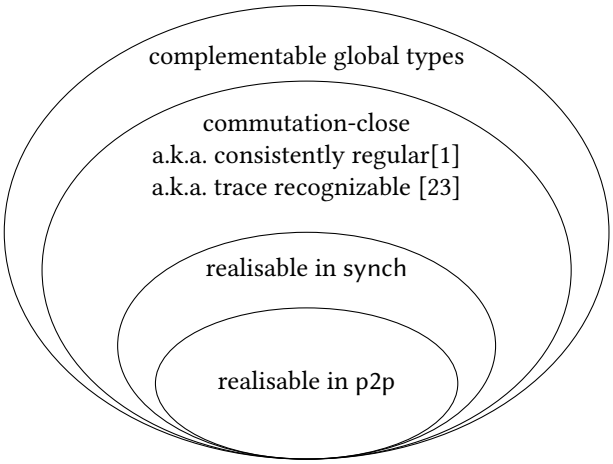


Figure 5. Hierarchy of semantic properties of global types

commutation-closed global type, and these form a subclass of the complementable ones (see Figure 5). Among others, it suggests that any framework for enforcing by construction the realisabilty of global types "hides" a complementation procedure for the global types embraced by the framework.

We explored the complementability of global types, looking for complementation procedures. We first observed that the complementation of a global type is not always possible, but we proposed several complementation procedures for various subclasses of global types, among which the one of sender-driven global types. For each of these subclasses, we derive a decision procedure for realisability in synch with PSPACE complexity, and for realisability in p2p with at most EXPSPACE complexity. Figure 6 summarises these results.

We believe that our results not only improve theoretical clarity but also are amenable to generalisation. This paper represents a first step towards a parametric framework for realisability checking, with potential applicability across diverse communication models. Throughout the paper, we have made a deliberate effort to explicitly state the hypotheses (related to the communication model) under which our theorems hold. A particularly critical assumption is that the communication models in question must be causally closed (see Definition 2.12). Based on this, we conjecture that our results can be extended to models such as bag and causally ordered systems, both of which satisfy causal closure. In contrast, models like mailbox or those based on bounded FIFO channels lack this property and therefore require specialised analysis. We aim to explore these cases further and, ultimately, to develop a more comprehensive framework in future work.

Apart from this generalisation, we aim to extend the framework to cover more expressive classes of global types, including those that go beyond synchronous or (quasi-synchronous) behaviour. We also plan to revisit results on the realisability of MSCs, possibly linking criteria such as send-validity

Class of global types	**Complementation procedure**	**Size of complement**	synch-**realisability**
$\lvert\mathbb{P}\rvert \leq 3$	$G \mapsto \mathsf{dual}(G)$	linear	PSPACE
commutation-closed	$G \mapsto \mathsf{dual}(G)$	linear	PSPACE
com-realisable	$G \mapsto \mathsf{dual}(\mathsf{prod}(\mathsf{proj}(G)))$	doubly exponential	constant time
sender-driven choices	$G \mapsto \mathsf{renun}(G)$	linear	PSPACE

Figure 6. Summary of complementation procedures for global types

and receive-validity [16, 19] to our synchronous realisability conditions. This opens the door to formally investigating whether global types that are realisable in the synchronous model satisfy send-validity, or if this is a weaker or stronger condition—this remains an open and promising question.

Related works. We conclude with a brief excursus on related work. Naturally, the most closely related lines of research are those concerning MPST. Comprehensive surveys on the topic are available in [8, 22]. It is noteworthy that in standard MPST literature, realisability—often referred to as implementability—is typically addressed through syntactic restrictions on global types. In most cases, a global type is implementable if a projection function exists. These syntactic constraints, along with the existence of the projection, imply that the global type is sender-driven, thus placing it within one of the complementable classes discussed in this paper. In contrast, a recent paper by Scalas and Yoshida [18] offers a novel perspective. Their approach emphasizes session fidelity and deliberately abandons syntactic restrictions on session types. Instead, they adopt semantic criteria—validated through model checking—to establish implementability, marking a significant shift from the conventional syntactic paradigm or our approach. Although, the comparison is difficult as the underlying models are substantially different, it is worth mentioning [5] that discusses projectability in the context of synchronous MPST.

The connection between communicating automata and behavioural types were first explored by Villard [21] for the binary case and Deniélou and Yoshida in [9] for MPSTs and later exploited by the authors in [16, 19]. Although our automata-theoretic approach of session types is not new, it is perphaps not so well established, and most of the papers on MPST define them syntactically. We took a fundamentally semantic view of session types, considering them as the subclass of higher order message sequence charts that specify only protocols that are realisable with synchronous communications. This point of view is highly debatable, but helps to clarify the relation between MPST and HMSCs.

In [16, 19], the authors revisited the theory of MPST projections through the lenses of realisability of MSCs, and proposed two criteria, called *send validity* and *receive validity*, that yield the first sound and complete characterisation of MPST implementable in the context of the p2p communication model, provided the global type is sender-driven. In particular, the second criteria, receive validity, is strongly justified by the choice of the communication model and is unsound for other communication models beyond p2p, hence not generalisable. As already commented above our approach provides a semantic criterion for implementability that leverages a reduction to the synchronous case, and it identifies a wider decidable subclass of global types (effectively complementable protocols) that covers the class of systems considered by Stutz et al. as a special case.

On a more foundational level, global types can be seen as a special form of higher-order message sequence charts (HMSCs, see [2]) with synchronous communications. Thus, the question of when a global specification is realisable by a distributed system is related to the classical MSC realisability problem studied by Alur et al. In particular, the implementability problem was formally introduced by Alur et al. [2] with early undecidability results, later simplified by Lohrey [17]. Our results refine these boundaries by showing that undecidability persists with four or more participants. Guanciale and Tuosto [13] extended these results on pomsets, we plan to build on their work to generalise to other communication models.

Mazurkiewicz traces [11] are words over partially commutative alphabets. Synchronous MSCs (but not asynchronous ones) can be seen as Mazurkiewicz traces over the alphabet Arr of arrows. Consistent regular trace languages [1] are the ones recognized by commutation-closed finite automata. Zielonka's asynchronous automata [23] are a model of a distributed implementation of a regular trace language. It differs from realizability of MPSTs and HMSCs in several ways: first, processes in this model synchronise by rendezvous (that may involve more than two processes), second processes may exchange additionnal information during rendezvous (not just the message label specified by the choreography), and third, the choreography is commutation-closed. As a consequence, all regular trace languages are implementable, but the "projection" operation is highly non-trivial in this setting. Although quite far from MPSTs at first sight, some old works on Mazurkiewicz traces could possibly shed some light on a few questions that arise from our work, and which, to the best of our knowledge, remain open, like the decidability of the complementability of a global type, or concise complementation procedures for non commutation-closed global types beyond sender-driven choices.

References

[1] IJsbrand Jan Aalbersberg and Grzegorz Rozenberg. 1988. Theory of traces. *Theoretical Computer Science* 60, 1 (1988), 1–82. doi:10.1016/0304-3975(88)90051-5

[2] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. 2005. Realizability and verification of MSC graphs. *Theor. Comput. Sci.* 331, 1 (2005), 97–114. doi:10.1016/J.TCS.2004.09.034

[3] Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing* (Chicago, IL, USA) *(STOC '04)*. Association for Computing Machinery, New York, NY, USA, 202–211. doi:10.1145/1007352.1007390

[4] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. 2001. Boolean and Cartesian Abstraction for Model Checking C Programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*. Springer-Verlag, Berlin, Heidelberg, 268–283.

[5] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 2024. Un-projectable Global Types for Multiparty Sessions. In *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming* (Milano, Italy) *(PPDP '24)*. Association for Computing Machinery, New York, NY, USA, Article 15, 13 pages. doi:10.1145/3678232.3678245

[6] Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (apr 1983), 323–342. doi:10.1145/322374.322380

[7] J.R. Buechi. 1962. On a decision method in restricted second-order arithmetic. *Proc. Internat. Congr. Logic, Methodology and Philosophy of Science* 1 (1962), 1–12.

[8] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. *A Gentle Introduction to Multiparty Asynchronous Session Types*. Springer International Publishing, Cham, 146–178. doi:10.1007/978-3-319-18941-3_4

[9] Pierre-Malo Deniélou and Nobuko Yoshida. 2012. Multiparty Session Types Meet Communicating Automata. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 194–213. doi:10.1007/978-3-642-28869-2_10

[10] Cinzia Di Giusto, Étienne Lozes, and Pascal Urso. 2025. *Realisability and complementability of multiparty session types*. Technical Report. Université Côte d'Azur. https://hal.science/hal-05166255

[11] Volker Diekert and Grzegorz Rozenberg (Eds.). 1995. *The Book of Traces*. World Scientific. doi:10.1142/2563

[12] Loic Germerie Guizouarn. 2023. *Communicating automata and quasi-synchronous communications*. Ph. D. Dissertation. Université Côte d'Azur.

[13] Roberto Guanciale and Emilio Tuosto. 2019. Realisability of pomsets. *J. Log. Algebraic Methods Program.* 108 (2019), 69–89. doi:10.1016/J.JLAMP.2019.06.003

[14] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 273–284. doi:10.1145/1328438.1328472

[15] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1, Article 3 (April 2016), 36 pages. doi:10.1145/2873052

[16] Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. 2023. Complete Multiparty Session Type Projection with Automata. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 13966)*, Constantin Enea and Akash Lal (Eds.). Springer, 350–373. doi:10.1007/978-3-031-37709-9_17

[17] Markus Lohrey. 2003. Realizability of high-level message sequence charts: closing the gaps. *Theor. Comput. Sci.* 309, 1-3 (2003), 529–554. doi:10.1016/J.TCS.2003.08.002

[18] Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* 3, POPL (2019), 30:1–30:29. doi:10.1145/3290343

[19] Felix Stutz. 2023. Asynchronous Multiparty Session Type Implementability is Decidable - Lessons Learned from Message Sequence Charts. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:31. doi:10.4230/LIPICS.ECOOP.2023.32

[20] Vasco Thudichum Vasconcelos and Kohei Honda. 1993. Principal Typing Schemes in a Polyadic pi-Calculus. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*. 524–538. doi:10.1007/3-540-57208-2_36

[21] Jules Villard. 2011. *Heaps and Hops*. Ph. D. Dissertation. École Normale Supérieure de Cachan.

[22] Nobuko Yoshida and Lorenzo Gheri. 2020. A Very Gentle Introduction to Multiparty Session Types. In *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 11969)*, Dang Van Hung and Meenakshi D'Souza (Eds.). Springer, 73–93. doi:10.1007/978-3-030-36987-3_5

[23] Wieslaw Zielonka. 1989. Safe Executions of Recognizable Trace Languages by Asynchronous Automata. In *Logic at Botik '89, Symposium on Logical Foundations of Computer Science, Pereslav-Zalessky, USSR, July 3-8, 1989, Proceedings (Lecture Notes in Computer Science, Vol. 363)*, Albert R. Meyer and Michael A. Taitslin (Eds.). Springer, 278–289. doi:10.1007/3-540-51237-3_22

Transporting Theorems about Typeability in LF Across Schematically Defined Contexts

Chase Johnson
joh13266@umn.edu
University of Minnesota
Minneapolis, MN, USA

Gopalan Nadathur
ngopalan@umn.edu
University of Minnesota
Minneapolis, MN, USA

Abstract

The dependently-typed lambda calculus LF is often used as a vehicle for formalizing rule-based descriptions of object systems. Proving properties of object systems encoded in this fashion requires reasoning about formulas over LF typing judgements. An important characteristic of LF is that it supports a higher-order abstract syntax representation of binding structure. When such an encoding is used, the typing judgements include contexts that assign types to bound variables and formulas must therefore allow for quantification over contexts. The possible instantiations of such quantifiers are usually governed by schematic descriptions that must be also be made explicit for effectiveness in reasoning. In practical reasoning tasks, it is often necessary to transport theorems involving universal quantification over contexts satisfying one schematic description to those satisfying another description. We provide here a logical justification for this ability. Towards this end, we utilize the logic $\mathcal{L}_{LF}$, which has previously been designed for formalizing properties of LF specifications. We develop a transportation proof rule and show it to be sound relative to the semantics of $\mathcal{L}_{LF}$. Key to this proof rule is a notion of context schema subsumption that uses the subordination relation between types as a means for determining the equivalence of contexts relative to individual LF typing judgements. We discuss the incorporation of this rule into the Adelfa proof assistant and its use in actual reasoning examples.

CCS Concepts

• **Theory of computation** → **Type theory**; **Logic and verification**; **Proof theory**; *Automated reasoning*.

Keywords

dependently typed lambda calculi, formalizing properties of typing judgements, equivalence of typing contexts, type subordination

ACM Reference Format:
Chase Johnson and Gopalan Nadathur. 2025. Transporting Theorems about Typeability in LF Across Schematically Defined Contexts. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The Edinburgh Logical Framework, also known as LF or the $\lambda\Pi$-calculus, has been used as a vehicle for formalizing rule-based relational specifications. When used in this mode, the dependent types of LF provide a natural means for encoding relations, and terms correspond to derivations for such relations. Typing judgements then validate such derivations.

An important feature of LF as a formalization tool is its support for what is often referred to as *higher-order abstract syntax* [11]. In this style of representation, binding in the constructs of an object-language are encoded using abstraction in the meta-language, which, in this case, is LF. When such a representation is used, the specification of relations that involve recursion over the structure of object language expressions give rise naturally to typing judgements in LF with non-empty typing contexts. As a concrete example, consider the size relation between a lambda term and a natural number. Under a higher-order abstract syntax encoding in LF, this relation for an abstraction term will be based on one corresponding to its body but in a typing context that binds the "free" variable in the body and that simultaneously assigns it the unit size.[1] An important point to note is that the typing contexts that manifest themselves in typing judgements that arise from such specifications satisfy a regular pattern. Thus, in the example considered, they comprise a sequence of a pair of bindings, each of which identifies a term variable and associates a size with it.

The broad concern in this paper is with reasoning about specifications that are encoded in LF in the manner described. A problem that arises in this context is that of lifting a property that has been established relative to a typing context that exhibits one kind of regularity to a situation where it possesses a different structure. To illustrate this phenomenon, suppose that we are interested in showing that every lambda term has an associated size. Observing that the size of an application term would be obtained by the addition of the sizes of the two subterms, we see that proving the property of interest relies crucially on the existence of a natural number that is the sum of two given natural numbers. Now, this is a property that is of general interest with regard to natural numbers and is likely to have been established in a library that formalizes them. Since the structure of natural numbers is devoid of binding notions, the most sensible setting for proving the property would be that where the typing context is empty. However, in the example under consideration, we need to use the property in a situation where the context may be non-empty but has the structure determined by the definition of the size relation.

[1] We will consider this example in more detail in Section 2.1, where this intuitive picture will become more precise.

It is this kind of lifting of theorems concerning LF specifications that we refer to as "transportation," and it is this problem that we study in this paper. An important part of building such a capability is understanding when one context can be replaced by another without impacting the validity of a formula. A basis for doing so is provided by the "dependence relation" between types that was introduced by Roberto Virga [18] and that is now commonly referred to as the *subordination* relation [8]. The negation of this relation allows us to determine when an object of a particular type may not influence the construction of an object of another type and, thereby, when a variable of the first type may be dropped from a context without impacting the validity of the typing judgement. However, this idea directly justifies the replacement of one context by another only in *individual* typing judgements. To be applicable in our situation, this analysis must be extended to the setting of properties that relate varied typing judgements. An additional complexity is that we must consider not just the replacement of single contexts but, rather, the swapping of *families of contexts* that are determined by the schematic descriptions governing the quantification of a context variable. Our primary endeavor in this work is to describe and formally justify a mechanism that addresses the two mentioned issues and that thereby provides a sound basis for theorem transportation.[2]

To consider the problem in a mathematically precise fashion, we cast it within a logic for reasoning about LF specifications. The particular logic we use for this purpose is called $\mathcal{L}_{LF}$ [10]. The atomic formulas in this logic correspond to typing judgements in LF and they are interpreted by the typing rules of the calculus. Properties about typing judgements can be expressed using a collection of logical connectives and quantifiers. The quantifiers in the logic range over both LF terms and contexts. The meaningful description of properties that involve quantification over contexts requires the domain of such quantifiers to be properly controlled. As we have observed in our example, a clue to what the relevant domain for particular such quantifiers may be is provided by the structure of the typing judgements involved. Adapting the idea of regular worlds from the Twelf system [13, 15], $\mathcal{L}_{LF}$ introduces the notion of a *context schema* as a type for context variables to capture this kind of structure. The question that we are interested in can then be formalized as the following: Can we describe circumstances under which the validity of a formula that involves a universal quantification over a context variable governed by one context schema implies the validity of the same formula but with the context variable quantification being governed by a different context schema? We answer this question by developing and utilizing the notion of *context schema subsumption*.

The logic $\mathcal{L}_{LF}$, which is initially described via a classical, substitution-based semantics, has been complemented by a proof system [9] that has been implemented in the Adelfa proof assistant [16]. The notion of *context schema subsumption* is brought to fruition in this setting through the enunciation of a sound proof rule to support theorem transportation. We discuss how this proof rule can be incorporated into the Adelfa system and we illustrate its use in this form through an example. More details concerning this example and a few others, including complete proof developments, can be found at the URL https://adelfa-prover.org/schema-subsumption.

The rest of this paper is structured as follows. The next two sections provide an overview of the aspects of LF and $\mathcal{L}_{LF}$ that are needed in subsequent discussions. Section 4 then develops the idea of context schema subsumption and establishes its properties. Section 5 uses this idea to describe the transportation proof rule. Section 6 discusses the integration of the proof rule into Adelfa and provides some examples of its use. We conclude the paper with a consideration of related work.

2 The Edinburgh Logical Framework

This section has a twofold purpose. First, it introduces the particular version of LF that will be used in this paper. Second it identifies the relation of subordination between types that provides the basis for pruning an LF context down to the part that is relevant to a typing judgement.

2.1 LF Syntax and Formation Judgements

Standard presentations of LF permit lambda terms that are not in normal form; *e.g.*, see [7]. Such a presentation simplifies the treatment of substitution, but it complicates arguments concerning adequacy and derivability of typing judgements that arise when the calculus is used to represent object systems. In light of this, we use a version known as *canonical LF* [8, 19] that only admits terms that are in β-normal form and where well-typing additionally requires that they be in η-long form. We will refer to this version simply as LF.

Kinds	K	::=	$\text{Type} \mid \Pi x{:}A.\, K$
Canonical Type Families	A, B	::=	$P \mid \Pi x{:}A.\, B$
Atomic Type Families	P	::=	$a \mid P\ M$
Canonical Terms	M, N	::=	$R \mid \lambda x.\, M$
Atomic Terms	R	::=	$c \mid x \mid R\ M$
Signatures	Σ	::=	$\cdot \mid \Sigma, c : A \mid \Sigma, a : K$
Contexts	G	::=	$\cdot \mid G, x : A$

Figure 1: The Syntax of LF Expressions

Expressions in LF belong to three categories: kinds, types, and terms. Kinds index types and types index terms. The syntax of these expressions is presented in Figure 1. The symbols c and a respectively denote term and type constants and x denotes a term variable. Terms and types are divided into canonical forms and atomic forms. The expressions include two binding forms: $\Pi x{:}A.\, B$ or $\Pi x{:}A.\, K$, which binds the variable x of type A in the type B or kind K, and $\lambda x.\, M$, which binds x in the term M.

An important operation on LF expressions that has a bearing even on typing judgements is that of substitution. Since the term syntax does not permit β-redexes, this operation must normalize

[2] A solution to these issues that uses subordination more or less directly has been described in a situation where LF is used also to encode metatheorems about LF specifications [8]. This solution relies on the particular, limited form for stating metatheorems in the underlying approach. Our goal here is to address the problem in a more general setting. We discuss this matter again in Section 7.

$$
\begin{array}{rl}
\texttt{nat}: & \text{Type} \\
\texttt{z}: & \texttt{nat} \\
\texttt{s}: & \texttt{nat} \to \texttt{nat}
\end{array}
\qquad
\begin{array}{rl}
\texttt{plus}: & \texttt{nat} \to \texttt{nat} \to \texttt{nat} \to \text{Type} \\
\texttt{plus-z}: & \Pi N{:}\texttt{nat}.\, \texttt{plus z } N\ N \\
\texttt{plus-s}: & \Pi N_1{:}\texttt{nat}.\, \Pi N_2{:}\texttt{nat}.\, \Pi N_3{:}\texttt{nat}. \\
& \Pi D{:}\texttt{plus } N_1\ N_2\ N_3. \\
& \texttt{plus } (\texttt{s } N_1)\ N_2\ (\texttt{s } N_3)
\end{array}
$$

$$
\begin{array}{rl}
\texttt{tm}: & \text{Type} \\
\texttt{app}: & \texttt{tm} \to \texttt{tm} \to \texttt{tm} \\
\texttt{lam}: & (\texttt{tm} \to \texttt{tm}) \to \texttt{tm}
\end{array}
\qquad
\begin{array}{rl}
\texttt{size}: & \texttt{tm} \to \texttt{nat} \to \text{Type} \\
\texttt{size-app}: & \Pi M_1{:}\texttt{tm}.\, \Pi M_2{:}\texttt{tm}.\, \Pi N_1{:}\texttt{nat}.\, \Pi N_2{:}\texttt{nat}.\, \Pi N_3{:}\texttt{nat}. \\
& \Pi D{:}\texttt{plus } N_1\ N_2\ N_3.\, \texttt{size } (\texttt{app } M_1\ M_2)\ (\texttt{s } N_3) \\
\texttt{size-lam}: & \Pi M{:}\texttt{tm} \to \texttt{tm}.\, \Pi N{:}\texttt{nat}. \\
& \Pi D{:}\big(\Pi x{:}\texttt{tm}.\, \texttt{size } x\ (\texttt{s z}) \to \texttt{size } (M\ x)\ N\big).\, \texttt{size } (\texttt{lam } M)\ (\texttt{s } N)
\end{array}
$$

Figure 2: Encoding Sizes of Untyped Lambda Terms

terms as it replaces variables. Care must therefore be exercised to ensure it is a terminating operation. Towards this end, substitutions are indexed by arity types that, as we will see in the next section, can be used to characterize the functional structure of expressions. Formally, these are simple types constructed from the atomic type o using the function type constructor $\to$. A substitution θ is then a finite set of the form $\{\langle x_1, M_1, \alpha_1\rangle, \ldots, \langle x_n, M_n, \alpha_n\rangle\}$, where, for $1 \leq i \leq n$, x_i is a distinct variable, M_i is a canonical term that is intended to replace the variable, and α_i is an arity type that governs the replacement. The application of θ to an expression E that is a kind, type, or term corresponds to the replacement of the free occurrences of the variables x_i by the terms M_i and a subsequent normalization mediated by the arity type α_i; we refer the reader to [9] for the details of this operation. Its application is not always guaranteed to be successful. However, in the situations that we use it here, it will always yield a result, which we will denote by $E[\![\theta]\!]$.[3]

There are seven typing judgements in (canonical) LF: $\vdash \Sigma$ sig that ensures that the constants declared in a signature are distinct and their type or kind classifiers are well-formed; $\vdash_\Sigma \Gamma$ ctx that ensures that the variables declared in a context are distinct and their type classifiers are well-formed in the preceding declarations and well-formed signature Σ; $\Gamma \vdash_\Sigma K$ kind that determines that a kind K is well-formed with respect to a well-formed signature and context pair; $\Gamma \vdash_\Sigma A$ type and $\Gamma \vdash_\Sigma P \Rightarrow K$ that check, respectively, the formation of a canonical and atomic type relative to a well-formed signature, context and, in the latter case, kind; and $\Gamma \vdash_\Sigma M \Leftarrow A$ and $\Gamma \vdash_\Sigma R \Rightarrow A$ that ensure, respectively, that a canonical and atomic term are well-formed with respect to a well-formed signature, context and canonical type. We refer the reader to [8] or [10] for the rules defining these judgements. We limit ourselves to two remarks here. First, the formation rules for canonical types and terms, *i.e.*, the rules for $\Gamma \vdash_\Sigma P \Rightarrow K$ and $\Gamma \vdash_\Sigma R \Rightarrow A$ will require the substitution of the argument term in an LF expression. This substitution will be indexed by the arity type that is obtained from the one determined for the argument term through an "erasure" operation denoted by $(\cdot)^-$ and defined as follows: $(P)^- = o$ and $(\Pi x{:}A_1.\, A_2)^- = (A_1)^- \to (A_2)^-$. Second, all the judgements forms other than $\vdash \Sigma$ sig are parameterized by a signature that remains unchanged in the course of their derivation. In the rest of this paper we will assume a fixed signature that has been verified to be well-formed at the outset.

In the typical use of LF, an object system is specified through a signature. One set of declarations in the signature identify types and constructors for representing the objects of interest in the system. Further declarations identify (dependent) types that encode relations over these objects. Finally, these types are used to identify constants that encode rules for deriving the corresponding relations. Figure 2, which presents a set of declarations that encode the natural numbers and then uses this to describe the size relation relative to the terms in the untyped lambda calculus, exemplifies this paradigm. The encoding of natural numbers and the rules for the addition relation over such numbers, encoded here by the type constructor `plus`, follows the expected lines. The representation of lambda terms—in particular, the representation of abstractions—makes use of the higher-order abstract syntax approach. Thus, an object language term of the form $\lambda x.\, M$ will be represented by the LF expression $(\texttt{lam } (\lambda x.\, \overline{M}))$, where $\overline{M}$ is a representation of M and the latter abstraction is one in the meta-language, *i.e.*, in LF. The encoding of the rule that specifies the size relation for abstractions in the object language is also interesting to note. This encoding is realized through the LF constant `size-lam`. Using it to construct a term that has the type $(\texttt{size } (\texttt{lam } M)\ N)$ relative to an LF typing context Γ will require us to construct a term that has the type $(\texttt{size } (M\ x)\ N')$ relative to a context that enhances Γ with the bindings $x : \texttt{tm}, y : \texttt{size } x\ (\texttt{s } z)$; if this construction succeeds and if N can be matched with $(\texttt{s } N')$, then the overall construction will succeed. The derivation of the size relation for a closed lambda term represented by M would correspond under this encoding to showing the inhabitation of the type $(\texttt{size } M\ N)$ for some N relative to the empty typing context. It is easy to see now that the typing contexts that arise in the course of this task will all comprise of repetitions of "blocks" of bindings of the form $x : \texttt{tm}, y : \texttt{size } x\ (\texttt{s } z)$. It is this kind of regularity in the shape of contexts that will need to be captured in a logic for reasoning about LF specifications.

2.2 Subordination and Context Redundancies

Not all the associations contained in a context may be relevant to a typing judgement. An analytical approach to determining which bindings are irrelevant and hence may be pruned from the context would be useful in determining context equivalence: if

[3] The notion of substitution we use here is that developed in Section 2.1.2 of [9], which generalizes the hereditary substitution of [8] to the situation where multiple variables are replaced at the same time.

two contexts can be pruned to the same "core," then the validity of a typing judgement is invariant under a replacement of one by the other. The subordination relation between types that was identified in [18] provides a means for such pruning. Definition 2.1 presents this relation. Use is made here of the operation $|\cdot|$ that identifies the "head constant" of a type and that is given as follows: $|(a\ M_1\ \ldots\ M_n)|$ is a and $|\Pi x{:}A.\,B|$ is $|B|$.

Definition 2.1. The subordination relation induced by a well-formed signature Σ is the smallest relation between the type constants identified by Σ that satisfies the following conditions:

(1) Index subordination: For all declarations in Σ of the form
   (a) $a\colon \Pi x_1{:}A_1.\ \ldots \Pi x_n{:}A_n.$ Type, it is the case that $|A_i| \preceq a$; and
   (b) $c\colon \Pi x_1{:}A_1.\ \ldots \Pi x_n{:}A_n.\,A$, it is the case that $|A_i| \preceq |A|$ for $1 \leq i \leq n$.
(2) Reflexivity: For all a declared in Σ, $a \preceq a$.
(3) Transitivity: If $a_1 \preceq a_2$ and $a_2 \preceq a_3$ then $a_1 \preceq a_3$.

As an illustration, the subordination relation induced by the signature in Figure 2 is the following:

`tm` $\preceq$ `tm`	`tm` $\preceq$ `size`	`nat` $\preceq$ `nat`
`nat` $\preceq$ `plus`	`nat` $\preceq$ `size`	`plus` $\preceq$ `plus`
`plus` $\preceq$ `size`	`size` $\preceq$ `size`	

We write $a \not\preceq b$ to denote the fact that $a \preceq b$ does not hold. Subordination is extended from a relation between type constants to one between types by defining $A \preceq B$ to hold exactly when $|A| \preceq |B|$ does.

The intuitive understanding of the subordination relation is that a term of type B can appear in a term of type A only if $B \preceq A$. Thus, we may drop assignments of type B from a context without impacting the assessment that a term M has type A if $B \not\preceq A$. This idea underlies the minimization operation on contexts given by the rules below:

$$\frac{}{\cdot|_A^{\preceq} = \cdot} \qquad \frac{\Gamma|_A^{\preceq} = \Gamma' \quad B \preceq A}{(\Gamma, x:B)|_A^{\preceq} = (\Gamma', x:B)} \qquad \frac{\Gamma|_A^{\preceq} = \Gamma' \quad B \not\preceq A}{(\Gamma, x:B)|_A^{\preceq} = \Gamma'}$$

Although defined as a relation, for any given context Γ and type A, there is exactly one Γ' such that $\Gamma|_A^{\preceq} = \Gamma'$ holds. By an abuse of notation, we will write $\Gamma|_A^{\preceq}$ to denote that Γ'.

The properties concerning context minimization that are need in this paper are contained in the following proposition that follows from the results in [8].

Proposition 2.2. *Let Γ and Γ' be contexts and let A be a type such that $\vdash_\Sigma \Gamma$ `ctx` and $\Gamma|_A^{\preceq} = \Gamma'$. Then the following are true:*

(1) $\vdash_\Sigma \Gamma'$ `ctx` *holds.*
(2) $\Gamma \vdash_\Sigma A$ `type` *holds if and only if* $\Gamma' \vdash_\Sigma A$ `type` *holds.*
(3) *If* $\Gamma \vdash_\Sigma A$ `type` *is derivable, then* $\Gamma \vdash_\Sigma M \Leftarrow A$ *holds if and only if* $\Gamma' \vdash_\Sigma M \Leftarrow A$ *does. This equivalence extends to the judgements* $\Gamma \vdash_\Sigma R \Rightarrow A$ *and* $\Gamma' \vdash_\Sigma R \Rightarrow A$*, as well as* $\Gamma \vdash_\Sigma P \Rightarrow K$ *and* $\Gamma' \vdash_\Sigma P \Rightarrow K$.

3 Reasoning About LF Judgements

We are often interested in formalizing properties of LF specifications since these reflect properties of the object systems they describe. The logic $\mathcal{L}_{LF}$ provides a means for doing this. We describe this logic below towards providing a basis for posing and addressing the primary issue of interest in this paper.

3.1 Formulas in the Logic and their Meaning

The logic $\mathcal{L}_{LF}$ is parameterized by an LF signature that we will denote by Σ. The atomic formulas of $\mathcal{L}_{LF}$ are, at the outset, expressions that encode LF typing judgements of the form $\Gamma \vdash_\Sigma M \Leftarrow A$.[4] However, the syntax of these formulas differs somewhat from that of the LF expressions. To begin with, term variables that are bound by quantifiers are permitted to appear in types and terms. These variables have a different logical character from the variables that are bound by an LF context: they may be instantiated by terms in the domains of the quantifiers, whereas the variables bound by declarations in an LF context represent fixed entities that are also distinct from all other similar entities within the typing judgement. To capture the role of the variables bound in an LF context, they are represented by *nominal constants* [6, 17]; these are entities that we represent by the symbol n possibly with subscripts and that behave like constants except that they may be permuted in atomic formulas without changing the logical content. To support this treatment, nominal constants are also allowed to appear in expressions in the logic that encode LF types and terms. Finally, contexts may be represented by variables that can be quantified over. More specifically, the atomic formulas in $\mathcal{L}_{LF}$ take the form $\{G \vdash M : A\}$, where M and A represent an LF term and type with the caveats just described and G constitutes a *context expression* whose syntax is given by the following rule:

$$G \quad ::= \quad \Gamma \mid \cdot \mid G, n : A$$

The symbol Γ here denotes the category of variables that range over contexts.

There is actually another structural requirement that is imposed on the terms and types that appear in the formulas in $\mathcal{L}_{LF}$. They are expected to be in canonical form and to respect the functional structure determined by the dependent types and kinds associated with the constants and variables appearing in them. This requirement is realized through an *arity typing* relation $\Theta \vdash_{at} M : \alpha$ and an *arity kinding* relation $\Theta \vdash_{ak} A$ *type*, in which Θ is an *arity context* that assigns arity types to (term) constants, nominal constants and variables, α is an arity type, and M and A are, respectively, an LF term and type in which nominal constants may appear. We elide the specific definitions of these relations—the details may be found in [9]. The type assignment for constants in the arity context is obtained by taking each assignment of the form $c : A$ in Σ and replacing it with $c : (A)^-$; we denote this context induced by Σ by Σ^-. For nominal constants, we assume these are drawn from the set $\mathcal{N}$ that automatically assigns an arity type to each and, in fact, provides a denumerably infinite number of nominal constants for each arity type. The arity types for variables is determined by the location of the term or type in an expression in the logic.

[4] With one difference in interpretation: as will become clear in due course, the atomic formulas assert the well-formedness of Γ and A rather than assuming them. We also note that the primary interest within LF is in typing judgements of the form $\Gamma \vdash_\Sigma M \Leftarrow A$; judgements of the form $\Gamma \vdash_\Sigma R \Rightarrow P$ are useful mainly in defining those of the former kind for atomic types. In reasoning about typing judgements, it is therefore possible to dispense with the latter form of judgement via a derived rule for the judgement $\Gamma \vdash_\Sigma M \Leftarrow A$ when A is atomic [9].

$$\frac{}{\Theta \vdash_{dec} \cdot \Rightarrow \Theta} \qquad \frac{\Theta \vdash_{dec} \Delta \Rightarrow \Theta' \qquad y \text{ is not assigned by } \Theta' \qquad \Theta' \vdash_{ak} A\ type}{\Theta \vdash_{dec} \Delta, y : A \Rightarrow \Theta' \cup \{y : (A)^-\}}$$

$$\frac{x_1, \ldots, x_n \text{ are distinct variables} \qquad \Sigma^- \cup \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\} \vdash_{dec} \Delta \Rightarrow \Theta'}{\vdash \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}\Delta\ blk\ schema}$$

$$\frac{}{\vdash \cdot\ ctx\ schema} \qquad \frac{\vdash \mathcal{C}\ ctx\ schema \qquad \vdash \mathcal{B}\ blk\ schema}{\vdash \mathcal{C}, \mathcal{B}\ ctx\ schema}$$

Figure 3: Well-formedness Judgements for Block and Context Schemas

$$\frac{}{\cdot \rightsquigarrow_{dec} \cdot \bowtie \emptyset} \qquad \frac{\Delta \rightsquigarrow_{dec} G \bowtie \theta \qquad n : (A)^- \in \mathcal{N}}{\Delta, y : A \rightsquigarrow_{dec} G, n : A[\![\theta]\!] \bowtie \theta \cup \{\langle y, n, (A)^-\rangle\}}$$

$$\frac{\Delta \rightsquigarrow_{dec} G' \bowtie \theta \qquad \{\mathcal{N} \cup \Sigma^- \vdash_{at} t_i : \alpha_i \mid 1 \le i \le n\}}{\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}\Delta \rightsquigarrow_{bs} G'[\![\{\langle x_i, t_i, \alpha_i\rangle \mid 1 \le i \le n\}]\!]}$$

$$\frac{\mathcal{B} \rightsquigarrow_{bs} G}{\mathcal{C}, \mathcal{B} \rightsquigarrow^1_{cs} G} \qquad \frac{\mathcal{C} \rightsquigarrow^1_{cs} G}{\mathcal{C}, \mathcal{B} \rightsquigarrow^1_{cs} G} \qquad \frac{}{\mathcal{C} \rightsquigarrow_{cs} \cdot} \qquad \frac{\mathcal{C} \rightsquigarrow_{cs} G \qquad \mathcal{C} \rightsquigarrow^1_{cs} G'}{\mathcal{C} \rightsquigarrow_{cs} G, G'}$$

Figure 4: Instantiating a Context Schema

As we have already noted, in typical typing scenarios, instantiations for context variables adhere to a regular structure. To be able to reason effectively about such judgements, the logic must provide a means for imposing such structural constraints on instantiations. This requirement is realized by associating a special kind of typing with context variables that is inspired by the notion of *regular worlds* in Twelf [13, 15]. Formally, it is based on the use of *context schemas* whose forms are given by the following rules:

Block Declarations	Δ	::=	$\cdot \mid \Delta, y : A$
Block Schema	$\mathcal{B}$	::=	$\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}\Delta$
Context Schema	$\mathcal{C}$	::=	$\cdot \mid \mathcal{C}, \mathcal{B}$

Conceptually, a context schema comprises a collection of block schemas, each of which is parameterized by a set of variables and assigns types that are well-formed in the arity kinding sense to distinct variables that are also distinct from the variables parameterizing the block schema. These requirements are realized by the judgement $\vdash \mathcal{C}\ ctx\ schema$ that is defined by the rules in Figure 3. As an example, consider the context schema $\mathcal{B}_1, \mathcal{B}_2$ where $\mathcal{B}_1$ is the block schema $\{\}(x_1 : \mathtt{tm}, y_1 : \mathtt{size}\ x_1\ (\mathtt{s}\ \mathtt{z}))$ and $\mathcal{B}_2$ is the block schema $\{T : o\}(x_2 : \mathtt{tm}, y_2 : \mathtt{type\text{-}of}\ x_2\ T)$; we will refer to this context schema as $\mathcal{C}$. It is easy to see that this is a well-formed context schema, *i.e.*, that $\vdash \mathcal{C}\ ctx\ schema$ is derivable. Context schemas are intended to represent context expressions that are obtained via repeated instantiations of their block schemas. In this sense, the block schema $\mathcal{B}_1$ provides a template for generating a context expression that might arise when we try to check an LF term that is intended to encode a derivation that another LF term encoding an (object-language) lambda term has a size. Similarly, $\mathcal{B}_2$ provides a template for generating context expressions that might arise when we try to check an LF term that is intended to represent a typing derivation for an (object-language) lambda term. The context schema $\mathcal{C}$ allows both kinds of context expressions to be generated as well as those that mix the two kinds of "blocks" of type assignments.

The intended meaning of context schemas is codified in the judgement $\mathcal{C} \rightsquigarrow_{cs} G$ defined in Figure 4, which identifies G as an instance of the context schema $\mathcal{C}$. This definition uses the relation $\mathcal{B} \rightsquigarrow_{bs} G$, also formalized in the same figure, that identifies the context expression G as an instance of a block schema $\mathcal{B}$ if G is obtained by instantiating the variables parameterizing $\mathcal{B}$ with (closed) well-formed terms of the right arity types and replacing the variables it assigns types to with nominal constants. In elaborating the latter aspect, use is made of the further relation $\Delta \rightsquigarrow_{dec} G \bowtie \theta$, which holds if G is a context expression obtained from the block declaration Δ by replacing the variables that it assigns types to with nominal constants; the replacement is recorded in the substitution θ. The operation $\cdot[\![\cdot]\!]$ that is employed in these rules represents substitution application. The application of a substitution to $\mathcal{L}_{LF}$ terms and types is identical to that for LF, with the observation that nominal constants are treated just like other constants. The application to context expressions leaves context variables unaffected and simply distributes to the types in the explicit bindings.[5] As illustrations of this definition, we see that our example context schema $\mathcal{C}$ has as instances the context expressions $(n_1 : \mathtt{tm}, n_2 : \mathtt{size}\ n_1\ (\mathtt{s}\ \mathtt{z}))$ (resulting from instantiating the block schema $\mathcal{B}_1$), $(n_1 : \mathtt{tm}, n_2 : \mathtt{type\text{-}of}\ n_1\ T)$ (resulting from instantiating the block schema $\mathcal{B}_2$), and $(n_1 : \mathtt{tm}, n_2 : \mathtt{size}\ n_1\ (\mathtt{s}\ \mathtt{z}), n_3 : \mathtt{tm}, n_4 : \mathtt{type\text{-}of}\ n_3\ T)$ (resulting from instantiating both $\mathcal{B}_1$ and $\mathcal{B}_2$ once), where T is an LF expression representing a type in the corresponding signature.

The formulas of $\mathcal{L}_{LF}$ are given by the following syntax rule:

$$\textbf{Formulas} \quad F ::= \{G \vdash M : A\} \mid \top \mid \bot \mid F_1 \supset F_2 \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \forall x : \alpha.F \mid \exists x : \alpha.F \mid \Pi\, \Gamma : \mathcal{C}.F$$

The symbol Π represents universal quantification pertaining to contexts; note that such quantification is qualified by a context schema. The symbol x represents a term variable, i.e., the logic permits universal and existential quantification over LF terms. The symbol α that annotates such variables represents an arity type. A formula is said to be well-formed relative to an assignment Ψ of arity types to term variables and an assignment Ξ of context

[5] In the use manifest in Figure 4, context expressions do not have context variables in them, but they may have such variables in later uses of the substitution operation.

schemas to context variables if the following conditions hold: all context variables that occur in the formula do so within the scope of a context-level quantifier or are assigned a type by Ξ, and the terms and types in the formula are well-formed in an arity typing sense with respect to an arity context given by Σ^-, $\mathcal{N}$, and an arity typing for variables determined first by the (term-level) quantifiers within whose scope they appear and then by Ψ. A formal description of these requirements may be found in [9]. In what follows, we shall assume that any formula of interest is well-formed with respect to some Ψ and Ξ.

The meaning of a formula is clarified by instantiating the quantifiers and free variables in it with closed expressions, *i.e.*, expressions devoid of term and context variables, and then ascertaining the validity of the result using the LF typing rules. The instantiation must respect typing constraints. For a term variable quantifier, this requirement translates into the substitution term being well-formed in an arity typing sense and the substitution being arity type preserving. For a context variable quantifier, the instantiation must respect the governing context schema: if Γ is a context variable that is to be substituted for by G and Γ is qualified by the context schema C, then the judgement $C \rightsquigarrow_{cs} G$ must be derivable.

Quantifier instantiation requires the notion of substitution into a well-formed formula. A context variable substitution σ has the form $\{G_1/\Gamma_1, \ldots, G_n/\Gamma_n\}$ where, for $1 \leq i \leq n$, Γ_i is a context variable and G_i is a context expression. The application of σ to a formula F, denoted by $F[\sigma]$, corresponds to the replacement of the free occurrences of the variables $\Gamma_1, \ldots, \Gamma_n$ in F by the corresponding context expressions, renaming bound context variables appearing in F away from those appearing in $G_1, \ldots, G_n$. For term variables, we adapt the substitution operation described for LF expressions to formulas in $\mathcal{L}_{LF}$. This operation distributes over quantifiers and logical symbols in formulas, respecting the scopes of quantifiers through the necessary renaming. The application to an atomic formula of the form $\{G \vdash M : A\}$ distributes to the component parts.

Definition 3.1 formalizes validity for $\mathcal{L}_{LF}$ formulas. The first clause in this definition uses the judgements $\vdash_\Sigma G\ \texttt{ctx}$, $G \vdash_\Sigma A\ \texttt{type}$, and $G \vdash_\Sigma M \Leftarrow A$, where G, M, and A are closed expressions. Thus, these expressions are identical to the ones in LF with the exception that what were earlier understood to be variables bound in a context are now denoted by nominal constants. The typing rules in LF adapt naturally to this difference with perhaps the only noteworthy observation being that the rules for the binding operators λ and Π introduce new nominal constants into the typing context and correspondingly replace the bound variable by the chosen nominal constant in the body. We assume such an adaptation in the interpretation of these judgements in the definition. We also note here that the atomic formula $\{G \vdash M : A\}$ includes assertions of well-formedness for G and A in addition to the derivability of $G \vdash_\Sigma M \Leftarrow A$.

Definition 3.1. Let F be a closed well-formed $\mathcal{L}_{LF}$ formula.

- If F is $\{G \vdash M : A\}$, then it is valid exactly when $\vdash_\Sigma G\ \texttt{ctx}$, $G \vdash_\Sigma A\ \texttt{type}$, and $G \vdash_\Sigma M \Leftarrow A$ are derivable.
- If F is $\top$ it is valid and if it is $\bot$ it is not valid.
- If F is $F_1 \supset F_2$, it is valid if F_1 is not valid or F_2 is valid.
- If F is $F_1 \wedge F_2$, it is valid if both F_1 and F_2 are valid.
- If F is $F_1 \vee F_2$, it is valid if either F_1 or F_2 is valid.
- If F is $\Pi\, \Gamma : C.F$, it is valid if $F[G/\Gamma]$ is valid for every G such that $C \rightsquigarrow_{cs} G$ is derivable.
- If F is $\forall x : \alpha.F$, it is valid if $F[\![\langle x, M, \alpha\rangle]\!]$ is valid for every M such that $\mathcal{N} \cup \Sigma^- \vdash_{at} M : \alpha$ is derivable.
- If F is $\exists x : \alpha.F$, it is valid if $F[\![\langle x, M, \alpha\rangle]\!]$ is valid for some M such that $\mathcal{N} \cup \Sigma^- \vdash_{at} M : \alpha$ is derivable.

Example 3.2. To illustrate the capabilities of $\mathcal{L}_{LF}$, let us consider formalizing the property that the sum of two natural numbers is always defined. Based on the encoding presented in Figure 2, this can be done through the formula

$$\begin{array}{l}\Pi\, \Gamma : C.\forall N_1 : o.\forall N_2 : o. \\ \quad \{\Gamma \vdash N_1 : \texttt{nat}\} \supset \{\Gamma \vdash N_2 : \texttt{nat}\} \supset \\ \quad\quad \exists N_3 : o.\exists D : o.\{\Gamma \vdash D : \texttt{plus}\ N_1\ N_2\ N_3\}\end{array}$$

where C, the context schema qualifying the quantification over Γ, is a context schema with a single empty block schema $\{\}()$.[6] We can argue for the validity of this formula by induction on the height of the derivation of the judgement $\{\Gamma \vdash N_1 : \texttt{nat}\}$ that appears in it. In more detail, the argument proceeds by case analysis on its derivation. In the case that it is derivable because N_1 is $\texttt{z}$, we may pick the instantiations N_2 and $\texttt{plus-z}$ for N_3 and D, respectively, to show that the conclusion follows. When N_1 is $(\texttt{s}\ N_1')$, we invoke the inductive hypothesis with respect to the derivability of $\{\Gamma \vdash N_1' : \texttt{nat}\}$ to conclude that $\Gamma \vdash_\Sigma D' \Leftarrow \texttt{plus}\ N_1'\ N_2\ N_3'$ holds for some N_3' and D'. We can then instantiate N_3 and D to $(\texttt{s}\ N_3')$ and $\texttt{plus-s}\ N_1'\ N_2\ N_3'\ D'$, respectively, to complete the argument.

The context schema used in this example is the most natural one for the property: the only instantiation for the quantified context variable is the empty context $\cdot$, reflecting the fact that natural numbers do not embody binding notions. However, we can use richer context schemas that permit for non-empty instantiations with bindings that do not impact the typing judgements involved while still preserving the validity of the formula. For example, the formula would still be valid if the context variable quantification is governed by a context schema that comprises a single block schema of the form $x : \texttt{tm}, y : \texttt{size}\ x\ (\texttt{s}\ z)$. In the next two sections, we will develop a sound, mechanizable method for determining that formula validity is preserved under such changes in context schema qualification.

3.2 Proving the Validity of Formulas

The logic $\mathcal{L}_{LF}$ is complemented by a proof system that provides the basis for mechanizing arguments of validity for closed formulas; using this system, it is possible to formalize arguments such as the one described in Example 3.2. The rules in the system enable the derivation of sequents of the form $\mathbb{N}; \Psi; \Xi; \Omega \longrightarrow F$, where $\mathbb{N}$ is a finite set of nominal constants, Ψ is a finite set of term variables with associated arity types, Ξ is a finite set of context variables with associated context variable types, Ω is a finite set of *assumption formulas*, and F is a *conclusion* or *goal* formula. The formulas in $\Omega \cup \{F\}$ must be well-formed relative to Ψ and Ξ and must use only those nominal constants that are contained in $\mathbb{N}$; $\mathbb{N}$, Ψ, and Ξ are referred to as the *support set*, the *eigenvariables context*, and

[6] A more natural statement is perhaps one that does not involve quantification over a context, a matter we discuss in Section 6.

the *context variables context* of the sequent. The goal of showing that a closed formula F whose nominal constants are contained in the set $\mathbb{N}$ is valid translates into constructing a derivation for the sequent $\mathbb{N};\emptyset;\emptyset;\emptyset \longrightarrow F$. Using the quantifier rules will produce sequents with non-empty eigenvariables and context variables context as proof obligations. The context variable types will initially be equivalent to context schemas but will take on a more elaborate structure in the treatment of the typing of abstractions and case analysis over formulas of the form $\{G \vdash M : A\}$; we elide details since they are not relevant to this paper.

Showing the soundness of the rules in the system requires a definition of validity for sequents. This is done first for closed sequents, *i.e.*, ones of the form $\mathbb{N};\emptyset;\emptyset;\Omega \longrightarrow F$: this is valid if F is valid or one of the assumption formulas in Ω is not valid. A sequent of the general form $\mathbb{N};\Psi;\Xi;\Omega \longrightarrow F$ is then considered valid if all of its closed instances are valid, where such instances are obtained by substituting closed terms not containing the nominal constants in $\mathbb{N}$ and respecting arity typing constraints for the variables in Ψ and replacing the variables in Ξ with closed context expressions that respect their context variable types.

4 Context Schema Subsumption

We can now state the objective of this paper precisely: we would like to identify conditions under which the validity of a closed formula of the form $\Pi\,\Gamma : C.F$ implies that of the formula $\Pi\,\Gamma : C'.F$; the latter formula represents the transportation of the former to the context schema C'. For this formula to be valid, it must be the case that $F[G'/\Gamma]$ is valid for every closed context expression G' that instantiates the context schema C'. In describing sufficient conditions for this property, we will distinguish between instantiations for Γ that constitute well-formed context expressions with respect to the ambient signature and those that do not. For instantiations of the latter kind, we will impose a requirement on the structure of the formula F. We discuss this requirement in detail in the next section. We focus in this section on the case of well-formed contexts. For instantiations of this kind, we will identify a syntactically checkable relation between C and C' that, when combined with the validity of $\Pi\,\Gamma : C.F$, will ensure the validity of $F[G'/\Gamma]$ for every well-formed instance G' of C'.

The particular relation that we will identify between the context schemas C and C' is intended to have the following content: if this relation holds, then, corresponding to each well-formed instance G' of C' there must exist a well-formed instance G of C that is such that F is valid under the substitution of G' for Γ exactly when F is valid under the substitution of G for Γ. At an intuitive level, this is a subsumption relation between the context schemas C and C' that is parameterized by the context variable Γ and the formula F. We denote the relation by $C \sqsupseteq_F^\Gamma C'$.

We develop this relation and establish the essential property about it in the rest of this section. The first step in this direction is to identify a subsumption relation between context expressions. We do that in the first subsection. In relating context schemas, we will need to match up block schemas that comprise them. This process is most easily expressed when the names of the variables in block schemas are aligned. To facilitate this, we digress briefly to describe the idea of a variant of a block schema that enables the renaming of variables. The last subsection utilizes the preceding machinery to define the subsumption relation between context schemas and to show that it embodies the desired property.

4.1 Context Expression Subsumption

Given two closed context expressions G and G', we desire to characterize the situations in which G contains all the information that is needed from G' to determine the validity of a formula F when these context expressions are viewed as substitutions for a context variable Γ that possibly appears free in F. We denote this relation by $G \gg_F^\Gamma G'$. The first step in defining the relation is to identify those situations in which an object of a type A that appears in the instantiation of Γ can impact a typing judgement that appears within F. In this and other ensuing discussions, we assume the obvious extension of the subordination relation to types in which nominal constants appear.

$$\frac{A \preceq_\Gamma F}{A \preceq_\Gamma \Pi\,\Gamma' : C.F}\ \Gamma \neq \Gamma' \qquad \frac{A \preceq_\Gamma F}{A \preceq_\Gamma Qx\colon \alpha.F}\ Q \in \{\forall, \exists\}$$

$$\frac{A \preceq_\Gamma F_1}{A \preceq_\Gamma F_1 \bullet F_2}\ \bullet \in \{\supset, \wedge, \vee\} \qquad \frac{A \preceq_\Gamma F_2}{A \preceq_\Gamma F_1 \bullet F_2}\ \bullet \in \{\supset, \wedge, \vee\}$$

$$\frac{A \preceq A'}{A \preceq_\Gamma \{\Gamma \vdash M : A'\}} \qquad \frac{}{A \preceq_\Gamma \{\Gamma, n_1 : A_1, \ldots, n_k : A_k \vdash M : A'\}}$$

Figure 5: Subordination of a Type by a Formula

Definition 4.1. The relation $A \preceq_\Gamma F$, to be read as "the type A is subordinate to the formula F relative to the context variable Γ," holds exactly when it is derivable by virtue of the rules in Figure 5. Note that the last rule applies only if there is at least one explicit binding in the context expression that begins with Γ. It is easy to see that $A \preceq_\Gamma F$ is a decidable relation. We write $A \not\preceq_\Gamma F$ to mean that $A \preceq_\Gamma F$ is not derivable.

$$\frac{}{\cdot \gg_F^\Gamma \cdot} \qquad \frac{G \gg_F^\Gamma G'}{G, n : A \gg_F^\Gamma G', n : A} \qquad \frac{G \gg_F^\Gamma G' \qquad A \not\preceq_\Gamma F}{G \gg_F^\Gamma G', n : A}$$

Figure 6: Context Expression Subsumption Relative to a Formula and Context Variable

The desired relation between context expressions then amounts to noting that it is possible to leave out bindings where the type is not subordinate to the formula relative to the context variable for which they are being contemplated as substitutions.

Definition 4.2. The relation $G \gg_F^\Gamma G'$ holds between two context expressions G and G', a context variable Γ, and a formula F exactly when this relation is derivable by the rules shown in Figure 6. Anticipating a necessity in Section 4.3, we extend this relation to the situation where G and G' are block declarations by letting bindings be of the form $x : A$ instead of $n : A$.

The following lemma captures the essence of the subsumption relation for context expressions.

Lemma 4.3. *If a formula of the form* $\{\Gamma \vdash M : A'\}$ *occurs in the formula* F *and* $G \gg_F^\Gamma G'$ *is derivable, then* $G|_{A'}^{\preceq}$ *and* $G'|_{A'}^{\preceq}$ *must be identical. If a formula of the form* $\{\Gamma, n_1 : A_1, \ldots, n_k : A_k \vdash M : A'\}$ *occurs in* F *and* $G \gg_F^\Gamma G'$ *is derivable, then* G *must be identical to* G'.

Proof. In the first case, $A \preceq_\Gamma F$ is derivable for any type A such that $A \preceq A'$. In the second case, $A \preceq_\Gamma F$ is derivable for any type A. Using these observations, we can show both properties by an induction on the definition of $G|_{A'}^{\preceq} = G'|_{A'}^{\preceq}$. □

The key property pertaining to the subsumption relation is expressed in the following theorem.

Theorem 4.4. *Let* F *be a formula that is well-formed with respect to the type assignments* Ψ *and* Ξ *to term and context variables respectively. Further, let* θ *be a closed substitution for all the variables assigned types by* Ψ *that is such that* $F[\![\theta]\!]$ *is defined, let* Γ *be a context variable that possibly appears free in* F*, and let* σ *be a closed substitution for all context variables other than* Γ *that are assigned types by* Ξ*. Then for any closed context expressions* G *and* G' *such that* $\vdash_\Sigma G$ `ctx`, $\vdash_\Sigma G'$ `ctx` *and* $G \gg_F^\Gamma G'$ *are derivable, it is the case that* $F[\![\theta]\!][\sigma][G/\Gamma]$ *is valid if and only if* $F[\![\theta]\!][\sigma][G'/\Gamma]$ *is valid.*

Proof. The proof proceeds by induction on the structure of F. If F is $F_1 \vee F_2, F_1 \wedge F_2, F_1 \supset F_2$, the substitutions distribute over the connective and the induction hypothesis easily yields the desired result. If F is $\forall x\colon \alpha.F'$ or $\exists x\colon \alpha.F'$, we may assume without loss of generality that x is distinct from the free variables of F. But now we observe that the desired result would hold if $F[\![\theta \cup \{\langle x, M, \alpha\rangle\}]\!][\sigma][G/\Gamma]$ is valid if and only if $F[\![\theta \cup \{\langle x, M, \alpha\rangle\}]\!][\sigma][G'/\Gamma]$ is valid for any term M of the appropriate arity type. However, this must be true by virtue of the induction hypothesis. An argument that is similar in structure applies when F is of the form $\Pi\Gamma' : \mathcal{C}.F'$ and $\Gamma \neq \Gamma'$. The desired result follows trivially when F is $\top, \bot$, of the form $\Pi\Gamma : \mathcal{C}.F'$ or an atomic formula of the form $\{\Gamma', n_1 : A_1, \ldots, n_k : A_k \vdash M : A\}$ or $\{\Gamma' \vdash M : A\}$ where $\Gamma \neq \Gamma'$ because the formula is not affected by the substitution for Γ.

It is only left to consider the cases where F is an atomic formula of the form $\{\Gamma, n_1 : A_1, \ldots, n_k : A_k \vdash M : A\}$ or $\{\Gamma \vdash M : A\}$. In the former case, G must be identical to G' by Lemma 4.3 and the desired conclusion follows immediately. In the latter case, $F[\![\theta]\!][\sigma]$ is of the form $\{\Gamma \vdash M[\![\theta]\!] : A[\![\theta]\!]\}$. Since $G \gg_F^\Gamma G'$ is derivable, it follows using Lemma 4.3 that $G|_A^{\preceq} = G'|_A^{\preceq}$. The desired result now follows from Proposition 2.2, the fact that the head of a type remains unchanged under substitution, and the assumptions that G and G' are well-formed. □

4.2 Variants of a Block Schema

We now describe a process for renaming the variables that parameterize a block schema as well as the ones to which it assigns types. The key requirements of such a renaming is that it must not affect well-formedness and it must leave the instances of the block schema unchanged. At a conceptual level, the renaming process is straightforward: we simply apply a permutation of variable names to the block schema. However, the technical details are complicated by the fact that the application of the permutation must be capture avoiding and subsequent substitutions must be factored through the permutation. We handle these issues below by formulating the permutation as a substitution. The trusting reader may skip these details after noting the important properties for block schema variants embodied in the statements of Theorems 4.6 and 4.7.

In the discussions that follow, we say that a block declaration $y_1 : A_1, \ldots, y_m : A_m$ or a context expression $n_1 : A_1, \ldots, n_m : A_m$ is well-formed with respect to an arity context Θ if it is the case that $\Theta \vdash_{ak} A_i$ *type* is derivable for $1 \leq i \leq m$.

Definition 4.5. A variable permutation is a bijection on variables that differs from the identity map at only a finitely many points. We write $\{z_1/x_1, \ldots, z_n/x_n\}$ to denote the permutation that maps x_i to z_i for $1 \leq i \leq n$ and that is the identity for all other variables. Let π be a variable permutation. Then

(1) $\pi.x$ denotes the result of applying π to the variable x;
(2) the permutation substitution induced by an arity context Θ from π, denoted by $\hat{\pi}_\Theta$, is
$$\{\langle x, z, \alpha'\rangle \mid z/x \in \pi \text{ and } \alpha' \text{ is } \alpha \text{ if } x : \alpha \in \Theta \text{ and } o \text{ otherwise}\};$$
(3) if $\Delta = y_1 : A_1, \ldots, y_m : A_m$ is a block declaration that is well-formed with respect to the arity context Θ, then $\Delta[\![\pi, \Theta]\!]$ denotes the block declaration
$$\pi.y_1 : A_1[\![\hat{\pi}_\Theta]\!], \ldots, \pi.y_m : A_m[\![\hat{\pi}_\Theta]\!];$$
(4) if $\mathcal{B}$ is the block schema
$$\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}y_1 : A_1, \ldots, y_m : A_m$$
then $ctx(\mathcal{B})$ denotes the arity context
$$\Sigma^- \cup \{x_1 : \alpha_1, \ldots, x_n : \alpha_n, y_1 : (A_1)^-, \ldots, y_m : (A_m)^-\};$$
(5) if $\mathcal{B}$ is the block schema $\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}\Delta$, then $\pi.\mathcal{B}$ denotes the block schema
$$\{\pi.x_1 : \alpha_1, \ldots, \pi.x_n : \alpha_n\}\Delta[\![\pi, ctx(\mathcal{B})]\!].$$

The definition of $\Delta[\![\pi, \Theta]\!]$ and $\pi.\mathcal{B}$ assumes that relevant hereditary substitutions are well-defined, a requirement that can be seen to hold under the respective well-formedness assumptions. For any permutation π, we say that $\pi.\mathcal{B}$ is a variant of $\mathcal{B}$.

The important properties of block schema variants are contained in the following two theorems.

Theorem 4.6. *A variant of a well-formed block schema* $\mathcal{B}$ *is well-formed: if* $\vdash \mathcal{B}$ *blk schema has a derivation then* $\vdash \pi.\mathcal{B}$ *blk schema must also have one for any variable permutation* π.

Proof. If Θ is an arity context, let $\pi.\Theta$ represent the arity context obtained by replacing each assignment of the form $x : \alpha$ in Θ where x is a variable by $\pi.x : \alpha$. We first show that if Δ is a block declaration that is well-formed with respect to the arity context Θ'' and it is the case that $\Theta \vdash_{dec} \Delta \Rightarrow \Theta'$ has a derivation then so must $\pi.\Theta \vdash_{dec} \Delta[\![\pi, \Theta'']\!] \Rightarrow \pi.\Theta'$. We prove this by an induction on the derivation of $\Theta \vdash_{dec} \Delta \Rightarrow \Theta'$, making use of the fact that the arity kinding relation is preserved under permutations of variables. The theorem follows easily from this observation. □

Theorem 4.7. *Instances of block schemas are preserved across variants: if* $\mathcal{B} \rightsquigarrow_{bs} G$ *has a derivation then so must* $\pi.\mathcal{B} \rightsquigarrow_{bs} G$ *for any variable permutation* π.

Proof. Given a variable permutation π and a substitution θ, let $\pi.\theta$ denote the substitution $\{\langle \pi.x, t, \alpha\rangle \mid \langle x, t, \alpha\rangle \in \theta\}$. Let Δ be a block declaration that is well-formed with respect to Θ, let G be a context expression, and let θ be a substitution such that $\Delta \rightsquigarrow_{dec} G \bowtie \theta$ has a derivation. We then show by induction on this derivation that $\Delta[\![\pi, \Theta]\!] \rightsquigarrow_{dec} G[\![\pi, \Theta]\!] \bowtie \pi.\theta$ must have a derivation; the notation $G[\![\pi, \Theta]\!]$ represents an operation that replaces each assignment $n : A$ in G by $n : A[\![\hat{\pi}_\Theta]\!]$. Next, for a substitution ρ, let $ctx(\rho)$ be the arity typing context $\{x : \alpha \mid \langle x, t, \alpha\rangle \in \rho\}$. We then observe that if G is well-formed with respect to $\Sigma^- \cup ctx(\rho)$, then $G[\![\pi, \Theta]\!]$ is well-formed with respect to $\Sigma^- \cup ctx(\pi.\rho)$. We can now ascertain that $G[\![\pi, \Theta]\!][\![\pi.\rho]\!]$ is defined and must be identical to $G[\![\rho]\!]$. The desired result follows easily from these observations. □

4.3 Subsumption for Context Schemas

We now turn to defining the relation $C \sqsupseteq^{\Gamma}_{F} C'$. The definition we provide for it will have a constructive nature: given a well-formed instance G' of C', it will guide us in constructing a well-formed instance G of C that is such that F is valid under the substitution of G' for Γ exactly when F is valid under the substitution of G for Γ. More specifically, our definition of this relation will provide us a means for "pruning" the context expression G' to get G. Given the way context schema instances are generated, this ability will ultimately be based on a correspondence at the level of the block schemas that comprise context schemas. One part of the correspondence must pay attention to how context expressions impact the derivability of typing judgements within F. For this, the notion of context expression subsumption is useful. However, we also need to pay attention to the impact of the pruning on the well-formedness of the resulting context expression. In particular, the pruning in an earlier part of a context expression should not remove a binding that is necessary for the well-formedness of a later part of the same expression. The relation between block declarations relative to a context schema that is defined below provides the basis for ensuring this property.

Definition 4.8. Let C be a context schema and let Δ and Δ' be block declarations. Then the relation $\Delta \trianglelefteq_C \Delta'$ holds exactly when it is derivable using the rules in Figure 7. We extend this relation to the situation where Δ and Δ' are context expressions by letting bindings be of the form $n : A$ rather than $x : A$ and allowing nominal constants to appear in types.

$$\frac{}{\cdot \trianglelefteq_C \cdot} \qquad \frac{\Delta \trianglelefteq_C \Delta'}{\Delta, x : A \trianglelefteq_C \Delta', x : A}$$

$$\frac{\Delta \trianglelefteq_C \Delta' \qquad A \not\trianglelefteq A' \text{ for every } A' \text{ such that } y : A' \text{ appears in } C}{\Delta \trianglelefteq_C \Delta', x : A}$$

Figure 7: A pruning relation for block declarations relative to a context schema

The following lemma shows that the pruning operation on context expressions just described will not impact the well-formedness of subsequent assignments generated from the context schema C.

Lemma 4.9. *Let C be a context schema and let G and G' be context expressions such that $G \trianglelefteq_C G'$. Further, let $x : A$ be an assignment in a block declaration in C. Then $G|^{\trianglelefteq}_{A} = G'|^{\trianglelefteq}_{A}$.*

Proof. By an induction on the structure of G'. We omit the details. □

The next two lemmas together show that if $\mathcal{B}$ is a block schema whose block declaration is a pruned version of that of $\mathcal{B}'$ that also subsumes it, then there is an instance G of $\mathcal{B}$ that is a pruned version of G' and subsumes G' for any context expression G' that instantiates $\mathcal{B}'$.

Lemma 4.10. *Let C be a context schema and let Δ and Δ' be block declarations such that $\Delta \gg^{\Gamma}_{F} \Delta'$ and $\Delta \trianglelefteq_C \Delta'$ are derivable. Further, let $\mathbb{N}$, G' and θ' be such that $\Delta' \rightsquigarrow_{dec} G' \bowtie \theta'$ is derivable. Then there must be some G and θ such that $\Delta \rightsquigarrow_{dec} G \bowtie \theta$, $G \gg^{\Gamma}_{F} G'$, and $G \trianglelefteq_C G'$ are derivable.*

Proof. By induction on the derivation of $\Delta' \rightsquigarrow_{dec} G' \bowtie \theta'$; the generated G will lose some assignments to nominal constants present in G' and the θ will correspondingly lose some substitutions from θ'. The needed relationships between G and G' will follow easily from the given ones between Δ and Δ' and the fact that the subordination relation between type expressions is invariant under substitutions. □

Lemma 4.11. *Let C be a context schema and let G and G' be context expressions such that $G \gg^{\Gamma}_{F} G'$ and $G \trianglelefteq_C G'$ are derivable. For any substitution θ if $G'[\![\theta]\!]$ is defined then so must $G[\![\theta]\!]$ be. Moreover, $G[\![\theta]\!] \gg^{\Gamma}_{F} G'[\![\theta]\!]$ and $G[\![\theta]\!] \trianglelefteq_C G'[\![\theta]\!]$ must be derivable.*

Proof. The first property, that $G[\![\theta]\!]$ is defined, follows from observing that every type that appears in G also appears in G'. The remaining properties follow from the fact that the subordination between types is not impacted by substitutions. □

We are now in a position to define the subsumption relation between context schemas.

Definition 4.12. The relation $C \sqsupseteq^{\Gamma}_{F} C'$, to be read as "the context schema C subsumes the context schema C' relative to the formula F and the context variable Γ," holds exactly when it is derivable by virtue of the rules in Figure 8.

A key part of this definition is the relation $\mathcal{B}' \in^{\Gamma}_{F} C$ which matches block schemas in C' with ones in C. Intuitively, a block schema from C' matches one from C if bindings from a context expression that results from it can be pruned to get a context expression from the matching block schema, and the pruning will not impact the well-formedness of other context expressions generated from block schemas in C or typing judgements in F whose context is determined by Γ. The next two lemmas and culminating theorem formalize this intuition.

Lemma 4.13. *Let F be a formula in which the context variable Γ possibly appears free and let C and C' be context schemas such that $C \sqsupseteq^{\Gamma}_{F} C'$ is derivable. Further, let G' be a context expression such that $C' \rightsquigarrow^{1}_{cs} G'$ is derivable. Then there is a context expression G such that $C \rightsquigarrow^{1}_{cs} G$, $G \gg^{\Gamma}_{F} G'$, and $G \trianglelefteq_C G'$ are derivable.*

$$\frac{\mathcal{B}' = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}\Delta' \quad \mathcal{B} = \{y_1 : \alpha_1, \ldots, y_k : \alpha_k\}\Delta \quad \Delta \trianglelefteq_C \Delta' \quad \Delta \gg_F^\Gamma \Delta'}{\mathcal{B}' \in_F^\Gamma C, \mathcal{B}} \qquad \frac{\mathcal{B}' \in_F^\Gamma C}{\mathcal{B}' \in_F^\Gamma C, \mathcal{B}}$$

$$\frac{C \sqsupseteq_F^\Gamma C' \qquad \mathcal{B}' \text{ is a variant of } \mathcal{B} \qquad \mathcal{B}' \in_F^\Gamma C}{C \sqsupseteq_F^\Gamma C', \mathcal{B}} \qquad \frac{}{C \sqsupseteq_F^\Gamma \cdot}$$

Figure 8: A subsumption relation on context schemas

Proof. Since $C' \rightsquigarrow_{cs}^1 G'$, it must be the case that there is a derivation for $\mathcal{B}'' \rightsquigarrow_{bs} G'$ for some block schema $\mathcal{B}''$ in C'. Since $C \sqsupseteq_F^\Gamma C'$ is derivable, there must be a variant $\mathcal{B}'$ of $\mathcal{B}''$ of the form $\{y_1 : \alpha_1, \ldots, y_n : \alpha_n\}\Delta'$ and a block schema $\mathcal{B}$ of the form $\{x_1 : \alpha_1, \ldots, x_m : \alpha_m\}\Delta$ in C such that $\Delta \gg_F^\Gamma \Delta'$ and $\Delta \trianglelefteq_C \Delta'$ are derivable. By Theorem 4.7, there must be a derivation for $\mathcal{B}' \rightsquigarrow_{bs} G'$. But then it must be the case that there is a G_1' and a ρ' such that $\Delta' \rightsquigarrow_{dec} G_1' \bowtie \rho'$ is derivable and a θ such that $G' = G_1'[\![\theta]\!]$. By Lemma 4.10, there must be a context expression G_1 and a ρ such that $\Delta \rightsquigarrow_{dec} G_1 \bowtie \rho, G_1 \gg_F^\Gamma G_1'$ and $G_1 \trianglelefteq_C G_1'$ are derivable. Using these facts and Lemma 4.11, it is easily verified that $G_1[\![\theta]\!]$ is defined and that $\mathcal{B} \rightsquigarrow_{bs} G_1[\![\theta]\!]$ and, hence, $C \rightsquigarrow_{cs}^1 G_1[\![\theta]\!]$ is derivable. Letting G be $G_1[\![\theta]\!]$ and using Lemma 4.11 again, we can check that the requirements of the lemma hold. □

Lemma 4.14. *Let F be a formula, let Γ be a context variable that possibly appears in F, and let C and C' be context schemas such that $C \sqsupseteq_F^\Gamma C'$ is derivable. Let G and G' be context expressions such that $\vdash_\Sigma G$ ctx, $\vdash_\Sigma G'$ ctx, $G \trianglelefteq_C G'$ and $G \gg_F^\Gamma G'$ are derivable. Let G_1' be a context expression such that $\vdash_\Sigma G', G_1'$ ctx is derivable. Let G_1 be a context expression such that for every assignment of the form $n : A$ in G_1 it is the case that A is a substitution instance of some type B such that an assignment of the form $x : B$ appears in one of the block declarations in C. Further, let $G_1 \gg_F^\Gamma G_1'$ and $G_1 \trianglelefteq_C G_1'$ be derivable. Then $\vdash_\Sigma G, G_1$ ctx; $G, G_1 \gg_F^\Gamma G', G_1'$; and $G, G_1 \trianglelefteq_C G', G_1'$ must be derivable.*

Proof. By induction on the structure of G_1'. If G_1' is $\cdot$, the conclusion follows directly from the assumptions. Suppose, then, that G_1' is of the form $G_2', n : A$. Note that it must be the case that $\vdash_\Sigma G', G_2'$ ctx. Now, G_1 could either include the assignment for n or not. We consider each of these cases separately below.

Let us first consider the case where G_1 includes the assignment for n, *i.e.*, when it is of the form $G_2, n : A$. Here, it must be the case that $G_2 \gg_F^\Gamma G_2'$ and $G_2 \trianglelefteq_C G_2'$ are derivable. By the induction hypothesis, $G, G_2 \gg_F^\Gamma G', G_2'$; $G, G_2 \trianglelefteq_C G', G_2'$; and $\vdash_\Sigma G, G_2$ ctx must be derivable. The derivability of the first two judgements implies that the judgements $G, G_2, n : A \gg_F^\Gamma G', G_2', n : A$ and $G, G_2, n : A \trianglelefteq_C G', G_2', n : A$ must be derivable. Since there is a derivation for $\vdash_\Sigma G', G_2', n : A$ ctx, it must be the case that n does not appear in G', G_2' and that $G', G_2' \vdash_\Sigma A$ type has a derivation. Clearly, n must not appear in G, G_2 either since this comprises a subcollection of the bindings in G', G_2'. From Lemma 4.9, we see that $(G, G_2)|_B^{\trianglelefteq}$ is identical to $(G', G_2')|_B^{\trianglelefteq}$ for any type B such that $x : B$ appears in a block declaration in C. Using Proposition 2.2 and the fact that the head of a type remains unchanged under substitution, it follows that $G, G_2 \vdash_\Sigma A$ type must have a derivation if there is one for $G', G_2' \vdash_\Sigma A$ type. That $\vdash_\Sigma G, G_2, n : A$ ctx has a derivation is an easy consequence.

The argument in the case when G_1 does not include the assignment for n is simpler and, in fact, follows immediately from the induction hypothesis. We omit the details. □

Theorem 4.15. *Let F be a formula in $\mathcal{L}_{LF}$, let Γ be a context variable that possibly appears in F, and let C and C' be context schemas such that $C \sqsupseteq_F^\Gamma C'$ is derivable. Let G' be a context expression such that $C' \rightsquigarrow_{cs} G'$ and $\vdash_\Sigma G'$ ctx are derivable. Then there is a context expression G such that $C \rightsquigarrow_{cs} G$, $\vdash_\Sigma G$ ctx, $G \gg_F^\Gamma G'$, and $G \trianglelefteq_C G'$ are derivable.*

Proof. By an induction on the derivation of $C' \rightsquigarrow_{cs} G'$.

In the base case, G' is $\cdot$. The argument in this case is trivial: we simply observe that $C \rightsquigarrow_{cs} \cdot$ and then verify that the other three requirements are met by setting G to $\cdot$.

Now suppose that G' is of the form G_1', G_2' where $C' \rightsquigarrow_{cs} G_1'$ and $C' \rightsquigarrow_{cs}^1 G_2'$ are derivable; note that, from the assumption that $\vdash_\Sigma G'$ ctx it follows that $\vdash_\Sigma G_1'$ ctx. By Lemma 4.13 there is a context expression G_2 such that $C \rightsquigarrow_{cs}^1 G_2$, $G_2 \gg_F^\Gamma G_2'$ and $G_2 \trianglelefteq_C G_2'$ are derivable. By the induction hypothesis, there is a context expression G_1 such that $C \rightsquigarrow_{cs} G_1$, $\vdash_\Sigma G_1$ ctx, $G_1 \gg_F^\Gamma G_1'$, and $G_1 \trianglelefteq_C G_1'$ are derivable. From what we have at hand, it is clear that $C \rightsquigarrow_{cs} G_1, G_2$ has a derivation. Let us now take G to be G_1, G_2. All the pieces are then at hand for us to invoke Lemma 4.14 and thereby complete the proof. □

Theorems 4.15 and 4.4 together ensure that our subsumption relation captures the property we want of it: if $C \sqsupseteq_F^\Gamma C'$ holds, then corresponding to every well-formed instance G' of C' there is a well-formed instance G of C such that $F_1[G'/\Gamma]$ is valid exactly when $F_1[G/\Gamma]$ is valid, where F_1 is an instance of the formula F in which at most the variable Γ appears free.

5 Transporting Theorems Across Context Schemas

The analysis in the previous section goes only part of the way towards determining when we can transport a theorem of the form $\Pi\Gamma : C.F$ into one where the quantification of the context variable is governed by a different schema C': we still need to consider the validity of F when Γ is substituted for by an ill-formed instance of C' for Γ. An approach based on subordination that is similar to the one in Section 4 will not work in this situation: removing parts of a context expression could well render the result well-formed. In light of this, we use a different idea: we identify a structural property for formulas that ensures that they must be valid under ill-formed substitutions for a context variable Γ. We describe this

$$\frac{}{\|\bot\|_\Gamma = \mathbf{f}} \qquad \frac{\Gamma \text{ appears in } G}{\|\{G \vdash M : A\}\|_\Gamma = \mathbf{f}} \qquad \frac{\|F_1\|_\Gamma = \mathbf{t} \quad \|F_2\|_\Gamma = \mathbf{f}}{\|F_1 \supset F_2\|_\Gamma = \mathbf{f}} \qquad \frac{\|F_1\|_\Gamma = \mathbf{f} \quad \|F_2\|_\Gamma = \mathbf{f}}{\|F_1 \vee F_2\|_\Gamma = \mathbf{f}}$$

$$\frac{\|F_i\|_\Gamma = \mathbf{f}}{\|F_1 \wedge F_2\|_\Gamma = \mathbf{f}}\ i \in \{1,2\} \qquad \frac{\|F'\|_\Gamma = \mathbf{f}}{\|Qx\colon \alpha.F'\|_\Gamma = \mathbf{f}}\ Q \in \{\forall, \exists\} \qquad \frac{\|F'\|_\Gamma = \mathbf{f}}{\|\Pi\,\Gamma' : C.F'\|_\Gamma = \mathbf{f}}\ \Gamma' \neq \Gamma$$

$$\frac{}{\|\top\|_\Gamma = \mathbf{t}} \qquad \frac{\|F_i\|_\Gamma = \mathbf{t}}{\|F_1 \vee F_2\|_\Gamma = \mathbf{t}}\ i \in \{1,2\} \qquad \frac{\|F_1\|_\Gamma = \mathbf{t} \quad \|F_2\|_\Gamma = \mathbf{t}}{\|F_1 \wedge F_2\|_\Gamma = \mathbf{t}} \qquad \frac{\|F_1\|_\Gamma = \mathbf{f}}{\|F_1 \supset F_2\|_\Gamma = \mathbf{t}}$$

$$\frac{\|F_2\|_\Gamma = \mathbf{t}}{\|F_1 \supset F_2\|_\Gamma = \mathbf{t}} \qquad \frac{\|F'\|_\Gamma = \mathbf{t}}{\|Qx\colon \alpha.F'\|_\Gamma = \mathbf{t}}\ Q \in \{\forall, \exists\} \qquad \frac{\|F'\|_\Gamma = \mathbf{t}}{\|\Pi\,\Gamma' : C.F'\|_\Gamma = \mathbf{t}}\ \Gamma' \neq \Gamma$$

Figure 9: Structural analysis for formula validity under an ill-formed substitution

property in the first subsection below and then combine it with the results of the previous section to yield a sound proof rule for transporting theorems across context schemas.

5.1 Validity Under an Ill-formed Context Variable Substitution

By the definition of validity for closed atomic formulas, a formula of the form $\{G \vdash M : A\}$ is not valid if G is an ill-formed context expression. When such a formula appears negatively, it can provide the basis for concluding that the overall formula must be valid. We must, of course, pay attention to the overall propositional structure of the formula to leverage this observation. Figure 9 presents rules for inferring the relation $\|F\|_\Gamma = \mathbf{t}$ and the relation $\|F\|_\Gamma = \mathbf{f}$ that embody this kind of structural analysis of the formula F. The following theorem quantifies the intent of such an analysis.

Theorem 5.1. *Let F be a formula that is well-formed with respect to the type assignments Ψ and Ξ to term and context variables respectively. Further, let θ be a closed substitution for all the variables assigned types by Ψ that is such that $F[\![\theta]\!]$ is defined, let Γ be a context variable that possibly appears free in F, and let σ be a closed substitution for all context variables other than Γ that are assigned types by Ξ. Finally, let G be a closed context expression such that $\vdash_\Sigma G$ `ctx` is not derivable. Then*

1. *$F[\![\theta]\!][\sigma][G/\Gamma]$ is valid if $\|F\|_\Gamma = \mathbf{t}$ is derivable.*
2. *$F[\![\theta]\!][\sigma][G/\Gamma]$ is not valid if $\|F\|_\Gamma = \mathbf{f}$ is derivable.*

Proof. The two parts can be proved simultaneously by an induction on the structure of the formula F, using the definition of validity. □

5.2 A Proof Rule for Transporting Theorems

The desired proof rule for transporting theorems proved under one context schema to another context schema can now be presented. This is done below and shown to be sound in Theorem 5.2:

$$\frac{\mathcal{N}; \Psi; \Xi; \Omega \longrightarrow \Pi\,\Gamma : C.F \qquad C \sqsupseteq^\Gamma_F C' \qquad \|F\|_\Gamma = \mathbf{t}}{\mathcal{N}; \Psi; \Xi; \Omega \longrightarrow \Pi\,\Gamma : C'.F}$$

Theorem 5.2. *Suppose that $\mathcal{N}; \Psi; \Xi; \Omega \longrightarrow \Pi\,\Gamma : C.F$ is a well-formed sequent and that C' is well-formed context schema, such that $C \sqsupseteq^\Gamma_F C'$ is derivable. Further, let $\|F\|_\Gamma = \mathbf{t}$ be derivable. If $\mathcal{N}; \Psi; \Xi; \Omega \longrightarrow \Pi\,\Gamma : C.F$ is a valid sequent, then the sequent $\mathcal{N}; \Psi; \Xi; \Omega \longrightarrow \Pi\,\Gamma : C'.F$ must also be valid.*

Proof. Follows from the definition of validity for sequents, using Theorems 4.15, 4.4, and 5.1. □

6 Using the Proof Rule in Reasoning

We first consider how the proof rule can be used to realize the style of reasoning sketched in the introduction. Our interest is in showing that every term has an associated size. Assuming the LF declarations in Figure 2, this property is stated in $\mathcal{L}_{LF}$ by the following formula:

$$\forall M : o.\{\cdot \vdash M : \mathtt{tm}\} \supset \exists N : o.\exists D : o.\{\cdot \vdash D : \mathtt{size}\ M\ N\}.$$

This formula can be proved by an induction over the structure of the representation of an untyped lambda term. Noting the way a typing judgement is derived for the representation of an abstraction term, we see that we will actually need to prove the following generalized version of the formula that allows for non-empty typing contexts:

$$\Pi\,\Gamma : C'.\forall M : o.\{\Gamma \vdash M : \mathtt{tm}\} \supset \exists N : o.\exists D : o.\{\Gamma \vdash D : \mathtt{size}\ M\ N\},$$

Here, C' is the context schema $\{\}(x : \mathtt{tm}, y : \mathtt{size}\ x\ (\mathtt{s}\ \mathtt{z}))$. The abstraction case utilizes weakening to introduce an entry for the size of the associated term in the context of the antecedent, which preserves the height of the derivation. The case then follows easily from the induction hypothesis under this generalization. In the application case, i.e., when M has the form $\mathtt{app}\ M_1\ M_2$, the induction hypothesis allows us to conclude that there are terms D_1, D_2, N_1 and N_2 such that $\{\Gamma \vdash D_1 : \mathtt{size}\ M_1\ N_1\}$ and $\{\Gamma \vdash D_2 : \mathtt{size}\ M_2\ N_2\}$ have derivations. The derivability of these formulas also implies that there are derivations for $\{\Gamma \vdash N_1 : \mathtt{nat}\}$ and $\{\Gamma \vdash N_2 : \mathtt{nat}\}$. Thus, if $\Pi\,\Gamma : C'.F$ where F is the formula

$$\forall N_1 : o.\forall N_2 : o.\{\Gamma \vdash N_1 : \mathtt{nat}\} \supset \{\Gamma \vdash N_2 : \mathtt{nat}\} \supset \exists N_3 : o.\exists D : o.\{\Gamma \vdash D : \mathtt{plus}\ N_1\ N_2\ N_3\}$$

were derivable, then, using the `size-app` constant, we would be able to provide terms D_3 and N_3 such that

$$\{\Gamma \vdash D_3 : \mathtt{size}\ (\mathtt{app}\ M_1\ M_2)\ (\mathtt{s}\ N_3)\}$$

is derivable.

The proof obligation that we are left with looks much like the formula in Example 3.2. However, the context schema governing the quantification in the two cases is different: in the formula in Example 3.2, which is what we would expect to find in a "library" formalization of natural numbers, the quantification is governed by the context schema $\{\}()$. We could prove the property afresh with the new context schema but, from an efficiency and modularity perspective, it is desirable to obtain the conclusion directly from the

library version. We can use the proof rule for transporting theorems towards this end provided we are able to show that $\|F\|_\Gamma = \mathbf{t}$ and $C \sqsupseteq^\Gamma_F C'$ have derivations, where C is the context schema $\{\}()$. Both properties can be checked by a mechanizable process: for the latter, the LF specifications must be analyzed to conclude that $\mathtt{tm} \not\trianglelefteq \mathtt{nat}$, $\mathtt{tm} \not\trianglelefteq \mathtt{plus}$, $\mathtt{size} \not\trianglelefteq \mathtt{nat}$, and $\mathtt{size} \not\trianglelefteq \mathtt{plus}$ hold.

We have modified the Adelfa proof assistant to incorporate support for this kind of reasoning. Rather than providing a separate tactic for the proof rule, we have incorporated it into the *apply* tactic. This tactic utilizes premises $F_1, \ldots, F_n$ together with a "lemma" of the form $\Pi\overline{\Gamma : C}\forall\overline{x : \alpha}.F'_1 \supset \ldots F'_n \supset F'$ to infer a suitable instance of the formula F'. The matching of the premise formulas with instances of the antecedent formulas in the lemma may require the adjustment of the context schema and this is done automatically based on our proof rule in a new version of the *apply* tactic.

What we have considered above is the paradigmatic use we envisage for our rule. LF specifications often employ general relations (such as `plus`) in the definition of specific relations (such as `size`). Our rule permits theorems about the general relations proved without reference to where they may be used to be drawn upon in more specific reasoning tasks. We sketch three other examples drawn from the literature to illustrate such applications; complete proof developments can be found at the website mentioned in the introduction. The first concerns the task of showing that the size of a lambda term does not decrease under substitution [1]. The proof of this theorem must be relativized to typing contexts that arise in assessing the size of a term. An argument that is inductive on the structure of the term being substituted into requires the following property about natural numbers for the application case: if n_1 and n_2 are less than or equal to n'_1 and n'_2 respectively, then $n_1 + n_2$ is less than or equal to $n'_1 + n'_2$. The typing context would be empty in an independent statement of this property, but the gap can be filled using our rule.The second example is a "benchmark" property presented in [4] that concerns showing declarative equality for lambda terms implies algorithmic equality. The difference between these two notions is that the former explicitly builds in reflexivity, symmetry and transitivity. These properties are of independent interest for algorithmic equality and should be proved separately for it. The proof rule for transporting theorems allows us to lift these results to the context needed in establishing the desired equivalence. The third example is a solution to problem 1a of the PoplMark challenge [3] that emulates one provided using the Twelf system [2]. This problem concerns proving transitivity and narrowing for the subtyping relation in System $\mathsf{F}_{<:}$ when this relation is specified algorithmically. For technical reasons that are orthogonal to the concerns in this paper, it is most convenient to show these properties first using a context schema with multiple block schemas. However, it is desirable to present the final result relative to a context schema with a single block schema. The transportation mechanism that is supported by the proof rule developed in this paper allows for this change to be realized automatically. This example is interesting because it involves the use of context schema subsumption where one of the context schemas comprises multiple block schemas. Additionally, the example features the use of our ideas in a situation where the context quantification is not at the outermost level in the formula.

The most natural presentation for properties that do not involve a typing context is to omit the context quantification. Thus, the preferred presentation for the theorem about the existence of a sum for two natural numbers is perhaps the following:

$$\forall N_1 : o.\forall N_2 : o.\{\cdot \vdash N_1 : \mathtt{nat}\} \supset \{\cdot \vdash N_2 : \mathtt{nat}\} \supset \exists N_3 : o.\exists D : o.\{\cdot \vdash D : \mathtt{plus}\ N_1\ N_2\ N_3\}$$

The use of such presentations can be supported without sacrificing the ability to benefit from the transportation rule by essentially building the "context generalization" into the *apply* tactic. We support this ability in our implementation.

7 Conclusion

In this paper, we have described a method for transporting theorems about LF specifications that have been proved relative to contexts adhering to one schematic description to a situation where they adhere to a different such description. This method is based on the idea of context schema subsumption that uses in an intrinsic way the previously described notion of subordination for LF types [8, 18]. We have developed a sound proof rule around context schema subsumption and have illuminated its use through particular reasoning examples. We have also discussed the incorporation of the rule into the Adelfa proof assistant.

In [5], Felty et al. have described a collection of benchmarks for systems that use higher-order abstract syntax in reasoning tasks. The problems addressed in this paper fall into the category they describe as treating "non-linear context extensions." The method we have developed in fact treats problems that need a "generalized context" approach under their definition. The complementary category is that encompassed by a "relational" style of reasoning, where it is necessary to couple different contexts appearing in a formula through a relation. Supporting this approach within the context of $\mathcal{L}_{LF}$ is matter that is currently under study.

Two other systems that are in existence that support reasoning about LF specifications are Twelf [12] and Beluga [14]. Both systems are based on the idea of describing functions whose inputs and outputs are determined by LF typing judgements: by exhibiting a function of this type that an external checker determines to be total, one obtains a "proof" of a theorem that has a $\forall\exists$ structure. In Twelf, all theorems include an implicit universal quantification over a context variable at the outermost level. The domain of this quantifier is determined by a "world" description that is similar to our context schemas and all the typing judgements in the theorem are assumed to be relativized to the context that instantiates the quantifier. In contrast, Beluga allows contexts to be indicated explicitly with the typing judgements. Twelf allows for transportation of theorems between different world descriptions when these are related by a property called *world subsumption* [8] that, intuitively, encodes the idea that all the contexts corresponding to one world description are covered, relative to the theorem to be proved, by those corresponding to the other. The notion of context equivalence used is, in a sense, a bi-directional version of the context expression subsumption that we have described in Section 4.3. However, by exploiting the particular way in which theorems are stated in Twelf, context equivalence can be based directly on subordination by a type and the context minimization operation described in Section 2.2. World subsumption can also use subordination directly,

but how exactly it is to be determined is left unspecified in the development in [8]. Beluga, on the other hand, appears to support a functionality similar to the one we have developed here. However, we have not been able to find a formal development of the idea and what is described in [4] (see item 2 on page 322) seems to lead to unsound reasoning: when trying to transport a theorem of the form $\Pi\,\Gamma : C.F$ into one of the form $\Pi\,\Gamma : C'.F$, it is necessary to check that every block schema of C' is covered (in a suitable sense) by one of C, rather than the other way around as is indicated in [4].[7]

In summary, the ideas we have investigated here have been considered previously by others. However, our work differs from that manifest in prior systems in that it has been carried out in the framework of a logic that allows for explicit quantification over contexts and that supports a more flexible statement of relationships between LF typing judgements. Moreover, we have developed a fully automated approach to detecting context schema subsumption and have utilized the setting of the logic to show the soundness of the transportation process that we have based on this notion.

References

[1] [n. d.]. The Twelf Project. http://twelf.org/.

[2] Michael Ashley-Rollman, Karl Crary, and Robert Harper. 2005. A solution in Twelf to the PoplMark Challenge problems 1 and 2. Available from the URL https://www.seas.upenn.edu/ plclub/poplmark/cmu.html.

[3] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference (LNCS, 3603)*. Springer, 50–65. doi:10.1007/11541868_4

[4] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. 2015. The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations. *Journal of Automated Reasoning* 55, 4 (Dec. 2015), 307–372. doi:10.1007/s10817-015-9327-3

[5] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. 2015. The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 1-A Common Infrastructure for Benchmarks. *CoRR* abs/1503.06095 (2015). arXiv:1503.06095 http://arxiv.org/abs/1503.06095

[6] Andrew Gacek, Dale Miller, and Gopalan Nadathur. 2011. Nominal Abstraction. *Information and Computation* 209, 1 (2011), 48–73. doi:10.1016/j.ic.2010.09.004

[7] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184. doi:10.1145/138027.138060

[8] Robert Harper and Daniel R. Licata. 2007. Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming* 17, 4–5 (July 2007), 613–673. doi:10.1017/S0956796807006430

[9] Gopalan Nadathur and Mary Southern. 2021. A Logic for Reasoning About LF Specifications. (June 2021). Available from http://arxiv.org/abs/2107.00111..

[10] Gopalan Nadathur and Mary Southern. 2022. A Logic for Formalizing Properties of LF Specifications. In *Proceedings of the 24th International Symposium on Principles and Practice of Declarative Programming* (Tbilisi, Georgia) *(PPDP '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 13 pages. doi:10.1145/3551357.3551377

[11] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 199–208. doi:10.1145/960116.54010

[12] Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th Conf. on Automated Deduction (CADE) (LNAI, 1632)*, H. Ganzinger (Ed.). Springer, Trento, 202–206. doi:10.1007/3-540-48660-7_14

[13] Frank Pfenning and Carsten Schürmann. 2002. *Twelf User's Guide*. Available from http://www.cs.cmu.edu/~twelf/guide-1-4.

[14] Brigitte Pientka and Joshua Dunfield. 2010. Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description). In *Fifth International Joint Conference on Automated Reasoning (LNCS, 6173)*, J. Giesl and R. Hähnle (Eds.). 15–21. doi:10.1007/978-3-642-14203-1_2

[15] Carsten Schürmann. 2000. *Automating the Meta Theory of Deductive Systems*. Ph. D. Dissertation. Carnegie Mellon University. http://www.cs.yale.edu/homes/carsten/papers/S00b.ps.gz

[16] Mary Southern and Gopalan Nadathur. 2021. Adelfa: A System for Reasoning about LF Specifications. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*. doi:10.4204/eptcs.337.8

[17] Alwen Tiu. 2006. A Logic for Reasoning about Generic Judgments. In *Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06) (ENTCS, Vol. 173)*, A. Momigliano and B. Pientka (Eds.). 3–18. doi:10.1016/j.entcs.2007.01.016

[18] Roberto Virga. 1999. *Higher-order Rewriting with Dependent Types*. Ph. D. Dissertation. Carnegie Mellon University.

[19] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. 2003. *A concurrent logical framework I: Judgments and properties*. Technical Report CMU-CS-02-101. Carnegie Mellon University. Revised, May 2003.

[7] Experiments with version 1.1.2 of the Beluga system indicate that what is implemented in the system is indeed the unsound rule.

On the Automated Verification of BGP Convergence

Haoyun Qin*
Gerald Whitters*
qhy@seas.upenn.edu
whitters@seas.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

Boon Thau Loo
boonloo@seas.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

Carolyn Talcott
carolyn.talcott@gmail.com
SRI
Menlo Park, California, USA

Abstract

The Border Gateway Protocol (BGP) is employed by autonomous systems (ASes), such as network operators or ISPs, to build routing tables. However, depending on the routing policies implemented by these ASes, BGP may fail to converge, potentially rendering the network inoperative. This paper introduces a workflow that leverages SMT solvers and rewriting tools to automate the verification of BGP convergence within a given AS network. We encode the convergence conditions defined by the Metarouting theoretical framework as an SMT problem. While SMT solvers can automatically determine whether BGP will converge, they do not generate counterexample traces in cases of divergence. To overcome this shortcoming, we propose a sound divergence criterion. We also construct an executable model for verifying BGP convergence, which can be automated using the Maude rewriting tool to produce witness traces in divergent scenarios. The effectiveness of our approach is demonstrated through a series of experiments.

CCS Concepts

• **Theory of computation → Automated reasoning**; **Logic and verification**; **Equational logic and rewriting**; **Verification by model checking**; • **Networks → Routing protocols**; **Formal specifications**; **Protocol testing and verification**; • **Mathematics of computing → Solvers**.

Keywords

BGP, Maude, SMT, Metarouting

ACM Reference Format:
Haoyun Qin, Gerald Whitters, Boon Thau Loo, and Carolyn Talcott. 2025. On the Automated Verification of BGP Convergence. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

*Both authors contributed equally to this research.

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Due to the scale and concurrent nature of modern networks, identifying configuration bugs is a significant challenge. When such issues go undetected, they can lead to severe and sometimes catastrophic network failures. The widely used Border Gateway Protocol (BGP) enables autonomous systems (ASes) to build routing tables according to their packet forwarding policies, also known as preference relations. Prior work has shown that poorly configured or conflicting policies can prevent BGP from converging, leading to unstable routing tables and rendering the network non-operational [5, 8, 9]. We refer to the liveness property concerning whether routing tables eventually stabilize as the *BGP convergence problem*.

Reasoning about BGP convergence has long been a difficult problem due to the protocol's asynchronous execution, path-dependent decisions, and policy-driven behavior. Existing formal verification methods for BGP convergence generally fall into two categories.

(1) *Correct-by-design approaches*, such as Metarouting [7, 12], allow network engineers to ensure BGP convergence by proving that the preference relations of ASes satisfy certain monotonicity conditions [7, 12]. However, a major limitation of these methods is their lack of automation, which prevents them from efficiently detecting policy conflicts.
(2) *Model-checking approaches* [15, 16], in contrast, are designed to identify policy errors automatically. However, either the techniques proposed so far are not sound, meaning they may produce false positives, i.e., flagging problems that do not actually exist, or they are not scalable [17], not able to determine divergence even for networks with less than 5 nodes.

This paper revisits the BGP convergence problem with the goal of developing sound, scalable, and automated verification methods. The central innovation lies in combining the strengths of the two existing approaches. First, we automate correct-by-design techniques to verify whether a given network instance is guaranteed to converge. We show how SMT solvers can be employed to perform this verification automatically.

When convergence cannot be guaranteed, we turn to model-checking methods to uncover concrete counterexamples that demonstrate divergence. These counterexamples provide valuable diagnostic information, enabling network engineers to identify and correct misconfigurations.

One of the main challenges in using model checking to find a witness for BGP divergence is the potentially large size of the network. Our key insight is to leverage the SMT solver's ability to

generate unsat cores, which are minimal sets of constraints responsible for unsatisfiability in the convergence check. In the context of BGP, the unsat core highlights the specific network links and policy interactions that contribute to divergence. These critical links are then used to focus the model checker's exploration, significantly reducing the overall search space and improving efficiency.

To achieve this goal, this paper makes the following key contributions:

- **SMT Encoding of Metarouting:** We present an encoding of the Metarouting criteria for BGP convergence as an SMT problem, enabling the use of SMT solvers to perform convergence checks automatically. This approach overcomes a central limitation of Metarouting, which is its lack of automation. Our approach allows engineers to systematically verify convergence conditions.
- **Sound Criterion for Divergence:** Verifying liveness properties such as BGP convergence is inherently difficult. A common approach is to reduce liveness verification to a safety check [2]. We introduce a sound safety-based criterion for detecting BGP divergence. This criterion relates to the sequence of messages exchanged between ASes and identifies conditions under which BGP fails to stabilize. It leverages links identified in the SMT solver's unsat core, i.e., candidate links that are involved in non-terminating BGP computations, as input. To the best of our knowledge, this is the first sound divergence criterion formulated for the BGP convergence problem.
- **Executable Model for BGP Verification**: We demonstrate that our divergence criterion can be fully automated. By formalizing both the criterion and BGP 's operational semantics in rewriting logic, we enable executable verification using the Maude rewriting tool [3]. Furthermore, we show how the SMT-derived unsat core can be used to guide the search process, significantly reducing the state space and enabling the generation of meaningful counterexamples.

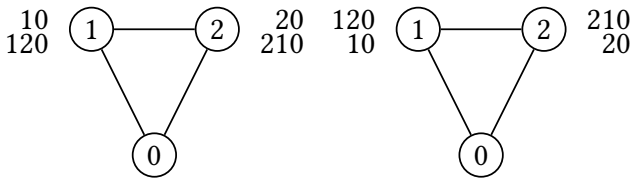
Our methods have been validated on networks combining several *networks gadgets*, i.e., network patterns, which appear in the literature. The experiments demonstrate that the methods can scale to realistic size networks containing 500 nodes and 1200 edges.

Section 2 describes by example the BGP protocol, and the BGP convergence problem including Metarouting. Section 3 provides a general overview of the verification flow proposed in this paper. Section 4 describes how SMT solvers can be used for BGP verification, while Section 5 specifies the sound criterion for used for model-checking network gadgets. The proposed criterion is encoded in Maude enabling optimization as described in Section 6. Section 7 validates the proposed verification flow with several experimental results. Finally, sections 8 and 9 conclude by discussing related and future work.

2 Border Gateway Protocol

2.1 BGP By Example

The Border Gateway Protocol (BGP) is a standardized exterior gateway protocol used for exchanging routing and reachability information between autonomous systems (ASes) on the Internet [10]. BGP is categorized as a path-vector routing protocol [6], and it



a. Agree Gadget **b. Disagree Gadget**

Figure 1: Example gadgets to showcase convergence and divergence in BGP

determines routing decisions based on path attributes, network policies, or rule sets configured by network administrators.

We illustrate BGP using the network gadgets shown in Figure 1, while the exact algorithm is described in detail in [10]. These gadgets, called *Agree Gadget* and *Disagree Gadget*, respectively demonstrate cases where BGP converges and where it may fail to do so.

Each node of a gadget is an AS. Each of these nodes have their own policy for determining a preference on the paths for routing. This policy is represented as an ordered list of paths in the figure (written top to bottom next to the node that contains the policy). A path p that appears before a path q in such a list is preferred over q in the corresponding policy. The distinction between Agree Gadget and Disagree Gadget is that the former prefers to route paths directly to 0, while the latter prefers to route paths through a neighbor before reaching 0. A key objective of BGP is to determine the best routing paths with respect to the ASes's preference relations.

BGP is run asynchronously among all nodes in a gadget to compute a satisfying path for each node. Each node receives paths from the nodes it is directly connected to that allows it to learn valid routes to a destination. Nodes will process these paths, and for each path will determine a selected best path from all of its known valid routes. The selected path is stored in a Routing Information Base (**RIB**) for each node. When the node has no current valid path to a destination we denote this with a special path $\perp$.

For every neighbor a node has, it stores the most recent path received from that neighbor in what is known as Routing Information Base - Inbound (**RIB-IN**). Each node also keeps a FIFO queue for each of its neighbors, these queues contain the paths received from that neighbor but haven't been processed by the node yet. When a node processes a path to a destination, it checks whether its policy prefers a valid path received from a neighbor over the current path stored in its **RIB** for that destination. If so, the node updates its **RIB** with the new path and broadcasts this updated path to its neighbors.

In some cases, a node may later receive a different path from the same neighbor that had previously sent the selected path **RIB**. In this case, the path in the **RIB** is no longer a valid route. A new one must be selected from its known paths stored in the **RIB-IN**. If there are no valid paths that can be selected from a node's **RIB-IN**, the special path $\perp$ is announced to its neighbors instead.

To showcase scenarios when BGP can find a solution and converge, as well as, when BGP might oscillate forever and diverge resulting in no solution, we explore an execution of BGP on Agree

Step		Node 1		Node 2	
		Node 0	Node 2	Node 0	Node 1
0	RIB-IN	⊥	⊥	⊥	⊥
	QUEUE	[(1, 0)]	[]	[(2, 0)]	[]
1	RIB-IN	(1,0)	⊥	⊥	⊥
	QUEUE	[]	[]	[(2, 0)]	[(2, 1, 0)]
2	RIB-IN	(1,0)	⊥	(2,0)	⊥
	QUEUE	[]	[(1, 2, 0)]	[]	[(2, 1, 0)]

Table 1: Initial BGP Steps For Agree Gadget and Disagree Gadget

Gadget and Disagree Gadget. For both gadgets, assume that BGP is used to determine the best paths to node 0. Due to the two gadgets only differing in path preferences, the first few steps taken when executing BGP will essentially be the same. Paths will be written in the form $(n_1, n_2, ..., n_k)$ for nodes $n_1, n_2, ..., n_k$ and $k \in$. Queues will be written in the form $[p_1, p_2, \ldots, p_k]$ for paths $p_1, p_2, \ldots, p_l$ and $l \in$.

We show those steps here running it on both gadgets in parallel and will discuss the gadgets individually when their execution would begin to differ. The results of each step of BGP is shown in the tables 1, 2, and 3. We show for nodes 1 and 2 the currently stored values of the **RIB-IN** and queue for each of their neighbors. The **RIB** for each node is designated by the **RIB-IN** value that is boxed when applicable. Initially, in step 0, node 0 will have already advertised to its neighboring nodes 1 and 2 the unit path (0). The corresponding paths, $(1, 0)$ and $(2, 0)$ are stored at the queues for node 1 and node 2 respectively. Next, both of these nodes can process the path. In step 1, node 1 will process path $(1, 0)$ and then in step 2, node 2 will process path $(2, 0)$. At both these steps, the **RIB-IN** entries at the node are $\perp$ before any processing is done. Thus, when each node processes its path in the queue, it will store this in its **RIB-IN** and then select that path as its **RIB** and announce this path to its neighbor. From step 3 and on we consider two different cases for Agree Gadget and Disagree Gadget respectively.

Agree Gadget consistently converges to a solution, regardless of the execution order in BGP. A straightforward example of its convergence can be demonstrated by continuing from the previously described steps. Let both nodes 1 and 2 process the single path in each of their non-empty queues in step 3 and in step 4 in sequence. Each queue contains a path that routes to a neighbor before finally reaching node 0. However, as discussed earlier, both nodes have a policy to prefer routes directly to 0. Since both nodes already have such a path selected as their **RIB**, when they process the path from their queues, they will store the path in the **RIB-IN**, but no new **RIB** will be selected, resulting in no new paths being announced from either node. All the queues are now exhausted, so BGP has terminated for Agree Gadget, giving the solution $(1, 0)$ as the **RIB** for node 1 and $(2, 0)$ as the **RIB** for node 2.

In general, a gadget may converge for some ordering of executions for BGP and diverge for others. For example, Disagree Gadget is not guaranteed to converge and we can display an example of its divergence. We again continue from the previously discussed

Step		Node 1		Node 2	
		Node 0	Node 2	Node 0	Node 1
3	RIB-IN	(1,0)	⊥	(2,0)	⊥
	QUEUE	[]	[]	[]	[(2, 1, 0)]
4	RIB-IN	(1,0)	⊥	(2,0)	⊥
	QUEUE	[]	[]	[]	[]

Table 2: BGP Steps for Agree Gadget

Step		Node 1		Node 2	
		Node 0	Node 2	Node 0	Node 1
3	RIB-IN	(1,0)	(1, 2, 0)	(2,0)	⊥
	QUEUE	[]	[]	[]	[(2, 1, 0); (2, 1, 2, 0)]
4	RIB-IN	(1,0)	(1, 2, 0)	(2,0)	(2, 1, 0)
	QUEUE	[]	[(1, 2, 1, 0)]	[]	[(2, 1, 2, 0)]
5	RIB-IN	(1,0)	(1, 2, 1, 0)	(2,0)	(2, 1, 0)
	QUEUE	[]	[]	[]	[(2, 1, 2, 0); (2, 1, 0)]
6	RIB-IN	(1,0)	(1, 2, 1, 0)	(2,0)	(2, 1, 2, 0)
	QUEUE	[]	[(1, 2, 0)]	[]	[(2, 1, 0)]

Table 3: BGP steps for Disagree Gadget

common steps and start at step 3. Let both nodes 1 and 2 process the single path in each of their non-empty queues in step 3 and in step 4 in sequence. This time, when each node processes the path in their queue and stores it in their **RIB-IN**, they will determine that this new path is more preferred by its policy, selecting it as its current **RIB** and sending it to its neighbor. At step 5 both nodes again only have a single path in any of its queues. Let both nodes 1 and 2 process the single path in each of their non-empty queues in step 5 and in step 6 in sequence. These paths replace the **RIB-IN** entry that is selected as the node's **RIB** but are not allowed by the policy, so each node will now need to fallback to its previous **RIB** that routed directly through node 0, and announce this path to its neighbors. The **RIB**, **RIB-IN**, and queues in step 6 are identical to those in step 2. It's clear to see that steps 3 to 6 can be repeated after step 6 indefinitely, resulting in BGP never terminating and thus diverging.

2.2 Metarouting

Metarouting [7, 12] showcases the use of Routing Algebras to design and represent routing protocols like BGP. The algebra can be written as a tuple $(W, \preceq, L, \Sigma, \phi, \oplus, f)$ [7, 12],

- W is a set of weights used to order the elements of Σ
- $\preceq$ is a total order on W
- L is a set of labels for the nodes in the network
- Σ is a set of signatures containing the paths in the network
- $\phi \in \Sigma$ is the special signature representing paths not allowed by the network's policy
- $\oplus$ is a binary operator that creates a new signature from an input of a label and a signature by prepending the node corresponding to the label and the path corresponding to the signature

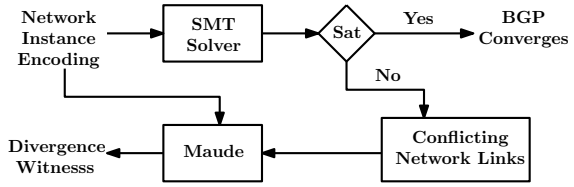


Figure 2: Verification Flow for BGP Verification

- f is a function that maps elements from Σ to elements in W

One of the benefits of this representation is that it is sufficient to show that if a Routing Algebra is monotonic, $\forall l \in L, \forall \alpha \in \Sigma : f(\alpha) \preceq f(l \oplus \alpha)$, then the corresponding network will converge. We use $\prec$ when $a \preceq b$ and $a \neq b$.

We use Agree Gadget and Disagree Gadget from Figure 1 to provide examples for a routing algebra. As noted previously, the policy of Agree Gadget prefers to traverse directly to the destination while Disagree Gadget prefers to traverse using a neighbor before reaching the destination, otherwise the gadgets are identical. When defining a routing algebra we can use the same symbols and definitions for both gadgets except when defining the function f. We use the notation f_A for Agree Gadget and f_D for Disagree Gadget, otherwise the other symbols will be shared between the two. We have that $W = \{1, 2\}$, $L = \{0, 1, 2\}$, $\Sigma = \{(1, 0), (1, 2, 0), (2, 0), (2, 1, 0)\}$. For the functions f_A and f_D, we define them such that:

$$\begin{aligned} f_A((1,0)) &= 1 & f_D((1,0)) &= 2 \\ f_A((1,2,0)) &= 2 & f_D((1,2,0)) &= 1 \\ f_A((2,0)) &= 1 & f_D((2,0)) &= 2 \\ f_A((2,1,0)) &= 2 & f_D((2,1,0)) &= 1 \end{aligned} \tag{1}$$

Using these definitions, it is straightforward to confirm the monotonicity property for the algebra, or provide a counter example. For Agree Gadget it is clear that $1 = f_A((1,0)) \preceq f_A((2,1,0)) = 2$ and $1 = f_A((2,0)) \preceq f_A((1,2,0)) = 2$. Hence, the algebra is monotonic and Agree Gadget converges. On the other hand, for the algebra corresponding for Disagree Gadget to be monotonic we need the following to hold $2 = f_D((1,0)) \preceq f_D((2,1,0)) = 1$, implying that $2 \preceq 1$, which is clearly false. Hence, we cannot make the same statement about convergence for Disagree Gadget.

Theorem 2.1. *[7, 12] For a network gadget $\mathcal{N}$, if a Routing Algebra corresponding to $\mathcal{N}$ is monotonic then BGP always converges for $\mathcal{N}$.*

3 Verification Workflow

Figure 2 shows the main steps and tools proposed for the BGP verification. The key insight is to combine (1) SMT solvers to automatically check for BGP convergence and (2) the Maude rewriting tool to enable the generation of a witness trace of BGP divergence. The witness trace can then be used by engineers to correct network configurations, e.g., path preference policies.

The main challenge of producing such a witness is the great size of networks which render model-checkers impractical. The solution uses SMT solver's capability of producing unsat cores to help pinpoint which part of the network might be contributing to BGP divergence, called the conflicting network links. This information is then used to enhance model-checking performance making verification feasible for larger networks.

The main steps of the verification flow are as follows and are made precise in the subsequent sections:

(1) **BGP Network Instance:** The input of the verification flow is the encoding of the network instance in SMT. The encoding specifies the network topology and path preference policies. This encoding is used by both the SMT solver and Maude in subsequent steps.
(2) **SMT Solver:** From the encoding of the network instance, an SMT problem is generated specifying the BGP convergence criterion specified by Metarouting [7, 12]. The satisfiability of this problem implies that BGP convergence is always guaranteed (Section 4).
(3) **Conflicting Network Links Extraction:** If the SMT problem for BGP convergence is not satisfiable, then the second part of the flow starts with the objective of producing a witness for divergence. In particular, the unsat core produced by the SMT solver contains the network links that may contribute to the divergence.
(4) **Maude Model-Checking:** The model-checking problem (Section 6) consists of proposed sound criterion for divergence (Section 5). It takes as inputs the conflicting network links and returns a witness of BGP divergence. Since the method is sound, but not shown to be complete, computation may not terminate.

4 Automating BGP Convergence Check with SMT

4.1 Computing a Monotonic Global Ranking

The metarouting work described previously shows that if a Routing Algebra is monotonic, then the corresponding network will converge (Theorem 2.1). Automating this check has been an open problem for which we address in this section. The key insight is instead of proving monotonicity, we check whether there is a global ranking among the paths in the network such that preserves the local preferences of paths and is monotonic.

Definition 4.1. A global ranking of paths in a network is an ordering among the paths such that if a node prefers path p over path q then path p has a higher rank than path q. A global ranking is monotonic if $p = n \oplus q$ then path p has a higher rank than path q.

Theorem 4.2. *A routing algebra is monotonic if and only if there exists a monotonic global ranking.*

Proof. The backward direction is immediate, as if there is a global rank that is monotonic, then the algebra is also monotonic. For the forward direction, assume that a Routing Algebra is monotonic. Then we can obtain a global order by using the total order obtained by the topological sorting of the partial order of in the routing algebra that gives us the monotonic global ranking. □

We utilize an SMT solver, cvc5 [1], to find a global ranking and determine if a network will converge. cvc5 supports the defining of custom datatypes and the theory of sequences. We create a custom datatype consisting of the nodes in a given gadget. This should

allow the solver to more effectively search for a solution as for any given symbol of the custom datatype, there are only finitely many choices. Paths are represented as sequences of this custom datatype. The sequence theory allows us to reason about sequences and various built in operations on sequences, e.g., concatenation, length, subsequence, etc.

First, we define two helper functions, H and T, to be used by the SMT Solver:

- H takes as input a path p and returns the first node in that path
- T takes as input a path p and returns that path without its first node $\mathrm{H}(p)$

Theories in SMT require that all functions are total, so the SMT solver is free to assign any value to H or T for inputs that these functions are not properly defined for, e.g., the empty path. Though, this shouldn't present any issues in our implementation.

We also define the function $\mathcal{S}$ that takes a path and returns an integer to represent the local policy at each node. Let p and q be paths that start from the same node n, i.e., $\mathrm{H}(p) = n = \mathrm{H}(q)$.

- If $p \neq q$ then $\mathcal{S}(p) \neq \mathcal{S}(q)$
- If $\mathcal{S}(p) > \mathcal{S}(q)$ then node n prefers path p to path q
- If p is the empty path then $\mathcal{S}(p) = 0$
- If p is allowed by the policy at node n then $\mathcal{S}(p) > 0$
- If p is not allowed by the policy at node n then $\mathcal{S}(p) < 0$

In our examples, each policy at a node is represented as an ordered list of allowed paths. We assign the last element in the list the integer 1 and iterate backwards assigned increasingly higher integers for each subsequent member iterated through. Any non empty path not seen at one of the policies is mapped to -1.

Lastly, we define the function $\mathcal{R}$ that takes a path and returns a natural number to represent the global ranking. Let p and q be paths that start from the same node n.

- If $p \neq q$ then $\mathcal{R}(p) \neq \mathcal{R}(q)$
- The ranking is in ascending order of the naturals such that if $\mathcal{R}(p) < \mathcal{R}(q)$ then p is ranked higher than q
- If p is the empty path then $\mathcal{R}(p) = 0$
- The ranking must preserve the ordering of the local preferences for each node's policies, i.e., if $\mathrm{H}(p) = \mathrm{H}(q)$ and $\mathcal{S}(p) > \mathcal{S}(q)$ then $\mathcal{R}(p) < \mathcal{R}(q)$
- The ranking must also be monotonic, i.e., if $\mathrm{T}(p) = q$ then $\mathcal{R}(p) > \mathcal{R}(q)$

Let P be all the paths allowed by a node in the network. To define $\mathcal{R}$ in SMT we use the following formulas: $\forall p, q \in P$

(1) $p \neq q \implies \mathcal{R}(p) \neq \mathcal{R}(q)$
(2) $p \neq q \wedge \mathrm{H}(p) = \mathrm{H}(q) \wedge \mathcal{S}(p) < \mathcal{S}(q) \implies \mathcal{R}(p) > \mathcal{R}(q)$
(3) $\mathrm{T}(p) = q \implies \mathcal{R}(p) > \mathcal{R}(q)$

Formulas consisting of quantifiers like $\forall$ can be extremely difficult for SMT solvers to deal with. To avoid this problem, we instead iterate through all the paths in P and construct corresponding formulas for every pair $p, q \in P$ to send to the solver. (1) requires that distinct paths are given distinct global rankings, (2) requires that local preferences are preserved, and (3) requires monotonicity. If the solver returns satisfiable (SAT), then it was able to find a global ranking.

Proof. Suppose that the SMT solver returns SAT after given the formulas described above. Then a monotonic global ranking can be extracted from $\mathcal{R}$ directly as the preferences of the local policies will be preserved from the constraints correspond to (2) and the formulas corresponding to (3) force the ranking to be monotonic. Now, suppose that such a monotonic global ranking exists. Then, there must be an ordering of the paths that preserves the preferences of the local policies and is monotonic. This ordering can be used directly to define the output for each input path for $\mathcal{R}$ and the solver would find that the formulas are SAT. □

The result of SAT to compute a global ranking implies that the network converges for BGP. If the solver returns unsatisfiable (UNSAT), then no such global ranking exists. The monotonicity condition is sufficient but not necessary, so it cannot be determined whether the gadget converges or diverges given an UNSAT result alone from the solver. To address this limitation we take advantage of the solver's unsat core and use this in a heuristic to attempt to find an example of divergence to both confirm the gadget does indeed diverge and to reveal the troublesome properties of the gadget that lead to the divergence so that a user may make appropriate changes to the gadget to ensure BGP converges for their network for BGP.

4.2 Extracting Links of Interest From an Unsatisfiable Core

To achieve this we use what is known as an UNSAT core. An UNSAT core is a subset of the input constraints that would lead to an UNSAT result. In particular, we query in search for a minimal UNSAT core in cvc5, a subset of the constraints UC that the solver produces UNSAT for, but any strict subset of UC produces SAT. This core may not be the minimum possible size but it can have drastically fewer elements than the original input set. Any computed UNSAT core would reveal a conflict in propositions that make it impossible to satisfy the monotonicity condition to correctly define $\mathcal{R}$. Since we form each constraint from a concrete pair of paths p, q, rather than using any quantifiers, when examining the UNSAT core the set of paths that fail to preserve monotonicity are explicitly displayed. For each path $p_i = (n_1, n_2, \ldots, n_k)$ that appears in a constraint from a minimal UNSAT core such that $k > 2$, we compute l_i as the link from node n_1 to node n_2. All such links, l_i, are passed to our implementation using Maude as described in the following section 6 to aid in locating a sequence of states from executing BGP that diverges for a given gadget.

To provide an example of what an unsat core might look like for a gadget that diverges, consider *Naughty Gadget* as shown in Figure 3. It is not difficult to show that this gadget can diverge, oscillating between processing paths from node 3 and node 4. Building the corresponding boolean constraints for this gadget as described above and sending it to the SMT solver gives an UNSAT result. Further, taking a look at the unsat core provided by the solver will look like the following:

- Score Constraints
  - $\mathcal{S}((3, 0)) = 1$
  - $\mathcal{S}((3, 4, 2, 0)) = 2$
  - $\mathcal{S}((4, 2, 0)) = 1$

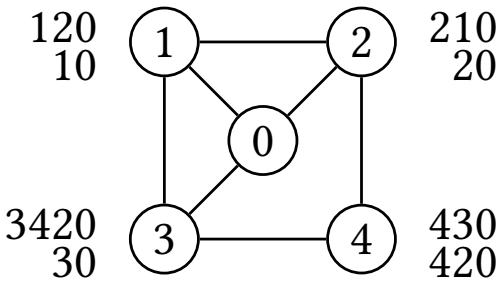


Figure 3: Naughty Gadget

- $\mathcal{S}((4,3,0)) = 2$
- Local Pref Constraints
  - $\mathrm{H}((3,0)) = \mathrm{H}((3,4,2,0)) \wedge \mathcal{S}((3,0)) < \mathcal{S}((3,4,2,0)) \implies \mathcal{R}((3,0)) > \mathcal{R}((3,4,2,0))$
  - $\mathrm{H}(4,2,0)) = \mathrm{H}((4,3,0)) \wedge \mathcal{S}(4,2,0)) < \mathcal{S}((4,3,0)) \implies \mathcal{R}((4,2,0)) > \mathcal{R}((4,3,0))$
- Monotonic Constraints
  - $\mathrm{T}((3,4,2,0)) = (4,2,0) \implies \mathcal{R}((3,4,2,0)) > \mathcal{R}((4,2,0)$
  - $\mathrm{T}(4,3,0)) = (3,0) \implies \mathcal{R}((4,3,0)) > \mathcal{R}((3,0))$

It is apparent that each constraint listed containing an implication symbol has its antecedent as evaluating to true, so we focus on just the conclusion when discussing these propositions. A chain of inequalities can be created such that:

$$\begin{aligned}\mathcal{R}((3,0)) &> \mathcal{R}((3,4,2,0)) \\ &> \mathcal{R}((4,2,0)) \\ &> \mathcal{R}((4,3,0)) \\ &> \mathcal{R}((3,0))\end{aligned}$$

This would require that $Rank(3,0) > Rank(3,0)$, but this is a contradiction. Hence, no ranking can be found satisfying the monotonicity conditions. This is shown using only 8 of the 217 total constraints originally given to the solver. Moreover, these paths found in the unsat core only begin from nodes 3 and 4 and only directly route through nodes $2, 3, 4$ from its starting node, excluding the common destination 0. We compute the links: node 3 to node 4, node 4 to node 2, and node 4 to node 3 as the set of links to send to Maude as a parameter to help in the search for an oscillating sequence of states that would cause divergence, using only 3 of the 11 total links in Naughty Gadget.

5 Sound Criterion of BGP Divergence

Following the verification flow described in Section 3, if the SMT solver does not produce SAT result for convergence, the task is to provide a counterexample for engineers to use for debugging purposes if possible.

In principle, model-checkers can determine counterexamples for liveness properties such as the BGP convergence property, but these checks are notoriously difficult. We follow instead the solution of reducing the checking of a liveness property to the checking of a safety property [2]. This reduction enables the model-checker to make more effective use of invariants thus improving considerably performance.

This section introduces such a safety property. Intuitively, we observe that divergence often manifests as cycles in the flow of messages across links, where specific paths are continuously produced and consumed in a repeating pattern without ever leading to stabilization. This motivates us to study the link-level message dynamics of BGP execution and to define divergence in terms of patterns in message production and consumption.

The following subsections formalize the BGP divergence criterion. Section 5.1 reviews the trace semantics of BGP networks based on the Simple Path Vector Protocol introduced by Griffin et al. [6]. Section 5.2 introduces the divergence property based on the messages that are produced and consumed by network links. Section 5.3 illustrates the criterion with an example.

5.1 Formal Network Setup and Execution Model

We begin by formalizing the BGP network abstraction. A BGP network instance is represented by a gadget $G = (\mathcal{N}, \mathtt{peer}, \mathcal{P}, S_0)$, where $\mathcal{N}$ is the finite set of nodes, `peer` maps each node to its peers, $\mathcal{P}$ is the local preference table assigning rankings to paths, and $S_0 = (\mathbf{Rib}, \mathbf{Rib\text{-}In}, \mathcal{Q})$ is the initial state of the system. Each state, including the initial state S_0, can be viewed as the gathering of the states maintained by each node $N \in \mathtt{node}$, i.e.,

- $\mathbf{Rib}(N)$: the best path currently selected by N;
- $\mathbf{Rib\text{-}In}(N)$: a mapping from peers to their most recent path advertisements;
- $\mathcal{Q}(N \Leftarrow N')$: a FIFO queue of messages pending from neighbor N' to N, per neighbor N'.

Execution Model Abstraction. The semantics of BGP route selection and message propagation follow the Simple Path Vector Protocol (SPVP) model introduced by Griffin et al. [6]. SPVP simplifies BGP to a message-passing system where each node asynchronously processes received paths, applies local preferences, and may update its selection and generate new messages.

The behavior of the BGP network is modeled via a scheduling algorithm $\mathcal{A}$, which selects, at each step, a valid transition $t = (N_r \Leftarrow N_s, P)$, representing node N_r processing path P received from neighbor N_s. A transition is said to be valid for a state if P is the head of the corresponding queue in that state. The execution of a transition results in the update of the the system state via the SPVP transition function.

Built on the above specification, an execution trace $\mathcal{T}$ is a sequence of states $S_0, S_1, \ldots, S_n$, where each transition corresponds to a valid transition applied according to an algorithm $\mathcal{A}$. Each trace implicitly induces a sequence of message productions and consumptions along each directed link.

5.2 BGP Divergence Criterion

To capture divergence in terms of observable system behavior, we introduce the notions of message production and consumption on each link.

Definition 5.1. For each directed link $N \Leftarrow N'$ and a trace $\tau = (S_A, S_1, \ldots, S_B)$, we denote,

- $P^{A\rightsquigarrow B}_{N\Leftarrow N'}$: the list of paths produced by N' and sent to N in the segment $S_A \rightsquigarrow S_B$ of the trace τ;
- $C^{A\rightsquigarrow B}_{N\Leftarrow N'}$: the list of paths consumed by N from N' in the segment $S_A \rightsquigarrow S_B$ of the trace τ.

These sequences are computed recursively over the trace, respecting the FIFO semantics of BGP message queues. The divergence of a link can further be indicated by a cyclic pattern in the production sequence that constantly replenishes the queue, thereby preventing stabilization.

We now define the central notion of a divergence point, which characterizes a trace segment that establishes recurring behavior indicative of global divergence.

Definition 5.2. A state S_B is said to be a divergence point of a prior state S_A if the following requirements are met:

(1) The control plane state, i.e., the selected paths (**Rib**) and the received advertisements (**Rib-In**), are same for both states;
(2) There exists a trace $\tau = (S_A, S_1, \ldots, S_B)$ such that for every link $N \Leftarrow N'$ that is active in the trace, the message production sequence $P^{A\rightsquigarrow B}_{N\Leftarrow N'}$, consumption sequence $C^{A\rightsquigarrow B}_{N\Leftarrow N'}$, and the S_A message queue $\boldsymbol{Q}_A(N \Leftarrow N')$ exhibit the following relationship,
   - We can find three natural numbers $k_p, k_c, k_q \in \mathbb{N}$, and a basic recurrent path pattern $R = R_p + R_s$, where the prefix R_p and the suffix R_s together form the loop body R, such that,
   - the production sequence can be written as $R_s + R^{k_p} + R_p$,
   - the consumption sequence is some repetition of R, i.e., R^{k_c},
   - the original message queue is in the form of $R^{k_q} + R_p$, and
   - $k_p + 1 \geq k_c$ ensures that the queue never drains.

This condition ensures that the queue state is self-sustaining, i.e., the system can re-enter the same queue configuration at S_B after consuming and producing such configuration of paths, enabling further repetition. This is formalized by the following statements.

Theorem 5.3. *If a trace segment $S_A \rightsquigarrow S_B$ satisfies the divergence point conditions, then there must exist a further segment $S_B \rightsquigarrow S_C$ that also ends in a divergence point S_C with respect to S_B.*

This means that any trace segment satisfying the divergence point conditions can be extended to an infinite trace.

Corollary 5.4. *If a BGP instance has a trace with a trace segment $S_A \rightsquigarrow S_B$ satisfying the divergence point conditions, then BGP does not always converge for the given BGP instance.*

5.3 Example

We illustrate the divergence criterion using the *Disagree Gadget* described in Section 2. The topology and local preferences are shown in Figure 1.

Starting from the initial state S_0, the following trace can be constructed,

$$S_0 \xrightarrow[(N_0)]{N_1\Leftarrow N_0} S_1 \xrightarrow[(N_0)]{N_2\Leftarrow N_0} S_2 \xrightarrow[(N_2,N_0)]{N_1\Leftarrow N_2} S_3 \xrightarrow[(N_1,N_0)]{N_2\Leftarrow N_1} \boldsymbol{S_4}$$
$$\xrightarrow[(N_2,N_1,N_0)]{N_1\Leftarrow N_2} S_5 \xrightarrow[(N_1,N_2,N_0)]{N_2\Leftarrow N_1} S_6 \xrightarrow[(N_2,N_0)]{N_1\Leftarrow N_2} S_7 \xrightarrow[(N_1,N_0)]{N_2\Leftarrow N_1} \boldsymbol{S_8},$$

where each transition $t = (N_r \Leftarrow L_s, P)$ is denoted as $S \xrightarrow[P]{N_r\Leftarrow N_s} S'$, where N_r processes a path P (the front of the queue) received from N_s. We now show that the subtrace $S_4 \rightsquigarrow S_8$ demonstrates a violation in our proposed safety property for divergence, as S_8 is a divergence point with respect to S_4.

Control Plane. First, we observe that the control plane state remains unchanged, thus satisfying condition (1) of Definition 5.2:

$$S_4.\mathbf{Rib}(N_1) = S_8.\mathbf{Rib}(N_1) = (N_1, N_2, N_0),$$
$$S_4.\mathbf{Rib}(N_2) = S_8.\mathbf{Rib}(N_2) = (N_2, N_1, N_0),$$
$$S_4.\mathbf{Rib\text{-}In}(N_1) = S_8.\mathbf{Rib\text{-}In}(N_1) = \{N_0 : (N_0), N_2 : (N_2, N_0)\},$$
$$S_4.\mathbf{Rib\text{-}In}(N_2) = S_8.\mathbf{Rib\text{-}In}(N_2) = \{N_0 : (N_0), N_1 : (N_1, N_0)\}.$$

Message Dynamics. Next, we examine the message dynamics on the two active links,

- For $N_1 \Leftarrow N_2$,
$$C^{4\rightsquigarrow 8}_{N_1\Leftarrow N_2} = [(N_2, N_1, N_0), (N_2, N_0)]$$
$$P^{4\rightsquigarrow 8}_{N_1\Leftarrow N_2} = [(N_2, N_0), (N_2, N_1, N_0)]$$
$$S_4.\boldsymbol{Q}(N_1 \Leftarrow N_2) = [(N_2, N_1, N_0)]$$
- For $N_2 \Leftarrow N_1$,
$$C^{4\rightsquigarrow 8}_{N_2\Leftarrow N_1} = [(N_1, N_2, N_0), (N_1, N_0)]$$
$$P^{4\rightsquigarrow 8}_{N_2\Leftarrow N_1} = [(N_1, N_0), (N_1, N_2, N_0)]$$
$$S_4.\boldsymbol{Q}(N_2 \Leftarrow N_1) = [(N_1, N_2, N_0)]$$

We now demonstrate that the divergence point condition holds for both links. Consider the following lasso recurring structure,

- for $N_1 \Leftarrow N_2$, $R_p = [(N_2, N_1, N_0)], R_s = [(N_2, N_0)], k_q = k_p = 0, k_c = 1$,
- for $N_2 \Leftarrow N_1$, $R_p = [(N_1, N_2, N_0)], R_s = [(N_1, N_0)], k_q = k_p = 0, k_c = 1$.

Thus, the divergence point condition is satisfied: the message queue maintains a self-replenishing pattern, and the system can re-enter the same configuration after each cycle, enabling indefinite repetition.

This demonstrates that BGP fails to converge in this instance, as the trace segment $S_4 \rightsquigarrow S_8$ satisfies the safety-based divergence condition.

Further Remarks. In order to automate the search for a diverging point one needs to determine the candidate trace from $S_A \rightsquigarrow S_B$. This is in principle not feasible when verifying large networks with hundreds of nodes. A key insight, as we describe in more detail in Section 6, is to provide as input the set of links that are activate in the candidate trace. This set of candidates, as illustrated in Figure 2, can be determined from the unsat core obtained from the SMT solver. Given this input of candidate links, verification is feasible as the model-checker can use this fact to reduce state-space.

The proposed criterion is sound, but its completeness is still open and left to future work. Indeed the bad gadget shown in Figure 4 is an example for which the criterion does not seem to work. In particular, the model-checker fails to converge, that is, it does not terminate. It is not clear whether it is because for this gadget the problem is indeed hard or whether the criterion is incomplete. We suspect the latter.

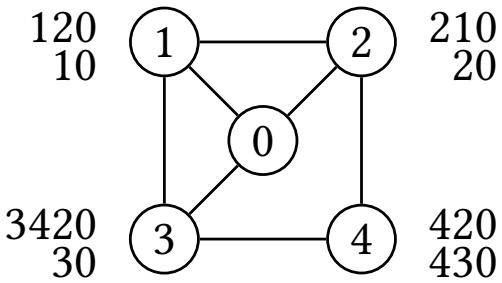


Figure 4: Bad Gadget

6 Formalizing Criteria in Rewriting Logic

Building on the divergence criterion proposed in the previous section, this section describes how we formalize and implement the criterion in rewriting logic using the Maude system. Our goal is to model BGP network behavior faithfully and to enable automated search for divergence traces that satisfy our safety-based condition.

We organize the presentation into three parts: Section 6.1 describes the formal system model in Maude, Section 6.2 presents the rewrite rules for protocol execution, and Section 6.3 discusses how the search space is reduced using the SMT-generated unsat core and other rewriting heuristics.

6.1 System Modeling

We model a BGP network as a set of nodes (i.e., autonomous systems) and a booking object for our proposed divergence criterion.

Nodes: Each node is represented as an object in Maude, as illustrates the following term:

```
< N1 : NodeClass |  id : nid(1),
    rib : nid(1) nid(2) nid(0),
    rib-in : (
        (nid(1) <= nid(0)) !-> nid(0),
        (nid(1) <= nid(2)) !-> nid(2) nid(0)
    ),
    permitted : (
        (nid(2) nid(0)) :: nid(0)
    ),
    neighbours : nid(2),
    queue : (
       (nid(1) <= nid(2)) !-> nid(2) nid(1) nid(0) ) >,
```

A node term encapsulates its local state, including its identifier (`id`), selected path (`rib`), received advertisements (`rib-in`), preferences (i.e. policies) (`permitted`), and pending message queues (`queue`). For example, in the instance above for a node `nid(1)`, the path selected to node `nid(0)` passes through nodes `nid(2)`; `(nid(1) <= nid(2)) !-> (nid(2) nid(0))` denotes that node `nid(2)` advertised to node `nid(1)` its best path to node `nid(0)` as passing through node `nid(2)`; and `(nid(2) nid(0)) :: nid(0)` indicates that the path going through `nid(2)` to `nid(0)` is more preferable than directly routing to `nid(0)`, and these are the only two permitted paths; finally, the queue stores the advertisements to be processed by the node.

Bookkeeping object: In addition to the network nodes, the system also includes a bookkeeping object for our proposed divergence criterion. This object maintains trace information necessary for evaluating the proposed divergence point condition.

```
< DPC : DPClass | sz : 2,
  consume : (
       (nid(1) <= nid(2)) !-> nid(2) nid(1) nid(0)
    ++ (nid(2) <= nid(1)) !-> nid(1) nid(2) nid(0)
  ),
  produce : (
       nid(1) !-> nid(1) nid(0)
    ++ nid(2) !-> nid(2) nid(0)
  ),
  init : (
    (nid(2) <= nid(1)) !-> nid(1) nid(2) nid(0),
    (nid(1) <= nid(2)) !-> nid(2) nid(1) nid(0)
  ),
  all-rib : (
       (nid(1) nid(2) nid(0) ; nid(2) nid(1) nid(0))
    ++ (nid(1) nid(0) ; nid(2) nid(1) nid(0))
  ),
  all-rib-in : ((
      (nid(1) <= nid(0)) !-> nid(0),
      (nid(2) <= nid(0)) !-> nid(0),
      (nid(1) <= nid(2)) !-> nid(2) nid(0),
      (nid(2) <= nid(1)) !-> nid(1) nid(0)
    ) ++ (
      (nid(1) <= nid(0)) !-> nid(0),
      (nid(2) <= nid(0)) !-> nid(0),
      (nid(1) <= nid(2)) !-> nid(2) nid(1) nid(0),
      (nid(2) <= nid(1)) !-> nid(1) nid(0)
    )
  ) >
```

The above term illustrates a concrete example of a divergence point object. It maintains a history of size 2, which records the past two link consumption (`consume`), path announcements (`produce`), control plane information (`all-rib, all-rib-in`), and records the initial queue contents (`init`) two-step before the current state accordingly.

Nodes and the bookkeeping structure are wrapped in a single term to ensure full rewriting on the whole system.

```
sort BGP .
op {_} : Configuration -> BGP [ctor] .
```

To signal successful detection of divergence during a search, we introduce a special constant object:

```
op diverged : -> Configuration [ctor] .
```

This object is inserted into the configuration when the divergence point condition is met, terminating the search.

6.2 Rewrite Rules

Maude transitions are specified via rewrite rules, which model the operational semantics of BGP node behavior as defined by the SPVP model.

Rewrite Configuration. To achieve efficient bookkeeping and searching, we introduce two rewrite configurations to be specified by the user before the search starts to narrow down and divide the search space.

- *sp-links*: We categorize all message links in the system into two types: *sp-links* and *non-sp-links*, where *sp-links* are links that are involved in the trace segment between divergence points, while *non-sp-links* are the remaining links. By specifying *sp-links*, the system will search for states that oscillate only between those links. This further eliminates the need for storing history queue and control plane information of *non-sp-links*, as well as supporting us to clear the history when *non-sp-links* are touched, and can thus largely help cut the state space.
- *sp-recur-size*: The exact number of steps involved in the oscillation proof, i.e., between two divergence points. By fixing this number, the system no longer needs to save history for longer than *sp-recur-size* steps, and will populate bookkeeping state when exceeded.

Rule. The main rewrite rule takes the following form:

```
rl [process-queue] : {
  C < A : NodeClass | id : N,
    queue : ((N <= N') !-> Msg :: Tail) ;  Q, ... >
  < DPC : DPClass | ... >
}
=>
if dp-check( dpc: < DP : DPClass | ... > )
then { diverged ... }
else {
  nodes-update( ... )
  if (N <= N') in sp-links then
    dp-update( updates: ..., sys: ... )
  else
    dp-clear( sys: ... )
  fi
}
fi .
```

At each step, the rule will first check if the divergence criterion is satisfied based on the bookkeeping term. If so, a divergence flag will be inserted. Otherwise, as modeled in SPVP, a node processes the first message in one of its incoming queues through pattern matching, potentially updates its state, and produces new path announcements to neighbors if needed. The bookkeeping object is cleared whenever the rewritten link is not in the *sp-links*, and is updated otherwise.

Rules are defined to preserve asynchrony: at each rewrite step, a node processes the message from an arbitrary non-empty queue. To enable verification, the rewrite engine can be asked to search for a state where the divergence flag is inserted.

6.3 Search Space Reduction

While our rewriting logic framework enables precise modeling and analysis of BGP convergence behavior, the state space for model checking remains prohibitively large in general. This agrees with observations in the literature [17]. This section introduces two key optimizations that significantly reduce the search space and make verification feasible even for larger networks.

We identify two sources of combinatorial explosion:

- **Rewrite Configuration Combinatorics.** As introduced in Section 6.2, users must specify a set of *sp-links* – the subset of network links suspected of participating in divergence. However, for a network with n links, the number of such subsets is exponential in size. Exhaustively enumerating these configurations is computationally infeasible.
- **Message Interleaving Explosion.** BGP's asynchronous nature means that messages can be processed in arbitrarily different orders. For large networks, this leads to a vast number of rewrite interleavings, even when only a small portion of the network contributes to divergence.

We address these issues using two complementary techniques.

Extract sp-links from SMT unsat core. A key insight from Section 4 is that the SMT solver's unsat core pinpoints the specific constraints responsible for the failure of guaranteeing BGP convergence. These constraints correspond to path preference conflicts that arise from particular network links. Therefore, links appearing in the unsat core can be useful to be specified as the *sp-links* for the system to search for divergence. We thus use the SMT-derived unsat core to initialize the set of `sp-links` automatically. This avoids the need for manual or exhaustive enumeration and sharply reduces the search space to only those parts of the network that may be involved in the potential oscillation.

Atomic Rewrite for Non-sp-links. Our second optimization is based on the observation that links outside of *sp-links* usually do not participate in the divergence pattern. Therefore, the specific order in which messages on non-*sp-links* are processed might not affect the existence of a divergence trace, in which cases, interleavings among them are semantically irrelevant.

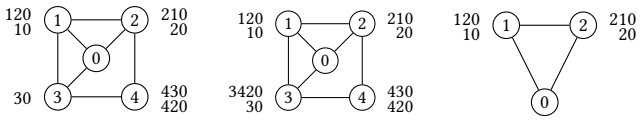
To exploit this, we introduce an atomic rewrite rule that processes the head of all non-empty non-*sp-links* in a single step. This reduces the interleaving among non-critical parts of the network and allows the rewrite engine to focus on meaningful variations in the behavior of *sp-links*.

```
crl [process-atomic] : { Conf1 } => { Conf2 }
if Nodes2 := nodes-update-all(
                       sys: Conf1,
                     links: non-empty-non-sp-links )
/\ DPC2 := dp-clear( sys: Nodes2 )
/\ Conf2 := DPC2 Nodes2 .
```

where `nodes-update-all` computes all the nodes in the configuration `Conf1` with non-*sp-links* and non-empty queues; and `dp-clear` executes the BGP algorithm, i.e., consuming the head of the nodes's queues, and adding to queues messages produced by link sources.

It must be pointed out that these heuristics are sound as no new behaviors are introduced by using atomic steps for non-*sp-links*. They, however, may lead to the tool not identifying a witnessing counterexample with the divergence. This happens, for example, with the bad gadget described at the end of Section 5.2 for which our formalization is not capable of finding a suitable counterexample.

7 Evaluation



a. Good Gadget **b. Naughty Gadget** **c. Disagree Gadget**

Figure 5: Base Gadgets

All experiments were run on a Windows 11 Home 24H2 machine, with an Intel Core i7-14700KF CPU, consisting of 64 GB of RAM, with Python 3.12.7 using Maude python bindings 1.4.0 [11] and cvc5 1.2.0. We run experiments on Good Gadget, Naughty Gadget, Disagree Gadget as seen in Figure 5. Naughty Gadget and Disagree Gadget have been discussed in previous sections, both are known to be able to diverge. Good Gadget instead is known to have a corresponding routing algebra that is monotonic and therefore always converges. We also run experiments by building larger gadgets using combinations of the three previously mentioned gadgets as base gadgets. When we create a combination in these experiments, we use gadgets where all but one are equivalent to the Good Gadget and the remaining is the gadget denoted in the first column. The total number of base gadgets used to create the gadget used for each experiment is denoted *size* and is reported in the second column. Thus, $size - 1$ gadgets are Good Gadget and the remaining gadget is the one recorded in the first column. The third column shows a tuple consisting of the number of nodes (#N), the number of links (#L), and the number of paths (#P) respectively in the corresponding gadget. The fourth column shows the amount of time in seconds it took to create the corresponding formulas for the SMT solver and then running the solver to check for convergence of the gadget, including the time it takes to compute the unsat core if necessary. The fifth and last column displays the amount of time in seconds it took to run the Maude implementation to find a trace that would oscillate resulting in the divergence. We provide experiments for when both SMT and Maude (or just SMT when the gadget converges) are able to terminate with a solution within an hour. If the gadget converges, there's no oscillating trace that can be found and we instead report **N/A** since we don't run the Maude program for these gadgets. There are three tables corresponding to the three types of gadgets we used, the base gadgets, *rooted combination* gadgets, and *nested combination* gadgets. The two combinations are constructed in a way such that the combined gadget diverges if and only if at least one of of the gadgets used to create it diverges. This is shown as we discuss how these combined gadgets are created.

Gadget	Size	(#N, #L, #P)	SMT (s)	Maude (s)
Good	1	(5, 11, 7)	0.026	**N/A**
Disagree	1	(5, 11, 8)	0.028	0.369
Naughty	1	(3, 4, 4)	0.018	0.121

Table 4: Base Gadget Results

The first combination type we denote as a *rooted combination*. We take n gadgets, $g_1, g_2, ..., g_n$ as input to construct a new gadget. First, we create a new node O as the new common destination. Then, we draw a link from each common destination O_{g_i} for g_i to O. Finally, every existing path in each gadget is appended with the new common destination O, and each O_{g_i} is given a policy for path preferences only consisting of (O_{g_i}, O).

For example, suppose we construct a rooted combination of Good Gadget as g_1 and Disagree Gadget as g_2 as seen in Figure 6. Then the path $(1, 2, 0)$ from Disagree Gadget becomes $(2:1, 2:2, 2:0, 0)$ by first tagging each node with the instance number 2, the index of Disagree Gadget from the list of inputs, then appending the global node 0.

It is clear that no two gadgets g_i, g_j for $i \neq j$ will have a path that traverses nodes from both gadgets. This means that we can essentially treat each gadget independently in the new combined gadget for the purposes of determining if BGP will converge or diverge. Hence, if some gadget g_i diverges then the new combined gadget will also diverge. Our machinery is able to handle gadgets constructed using the rooted combination consisting of up to 500 nodes, 1200 links, and 800 paths.

Gadget	Size	(#N, #L, #P)	SMT (s)	Maude (s)
Good	2	(11, 24, 16)	0.122	**N/A**
Naughty	2	(11, 24, 17)	0.113	1.39
Disagree	2	(9, 17, 13)	0.068	0.238
Good	5	(26, 60, 40)	0.728	**N/A**
Naughty	5	(26, 60, 41)	0.66	3.585
Disagree	5	(24, 53, 37)	0.542	0.596
Good	10	(51, 120, 80)	3.074	**N/A**
Naughty	10	(51, 120, 81)	2.636	9.601
Disagree	10	(49, 113, 77)	2.352	1.794
Good	50	(251, 600, 400)	223.523	**N/A**
Naughty	50	(251, 600, 401)	68.336	209.484
Disagree	50	(249, 593, 397)	66.953	58.908
Good	100	(501, 1200, 800)	2655.43	**N/A**
Naughty	100	(501, 1200, 801)	324.879	862.888
Disagree	100	(499, 1193, 797)	292.157	240.22

Table 5: Rooted Combination Results

The other combination type we call a *nested combination*. This combination takes two gadgets g_a and g_b as inputs to construct a new gadget g_{ab}. The first step of this type is to replace every node in g_a that is not its common destination O_{g_a} with the nodes and links (but not paths) of gadget g_b. O_{g_a} becomes the common destination of the newly created nested combination g_{ab}. Only the nodes corresponding to the common destination in g_b, O_{g_b}, will have a link that directly connects them to O_{g_a}. We denote each newly created node as $A : B$ where A refers to a node from g_a and B refers to a node from g_b. Now, at each node $A : B$ to determine the policy for its path preference we compute the following: for each path $p_a = (u_1, u_2, \ldots, u_m, O_{g_a})$ in the policy for node A from g_a, for each path $p_b = (v_1, v_2, \ldots, v_n, O_{g_b})$ in the policy for node B from g_b, we create the path:

$$\begin{aligned} p_a \circ p_b &= (u_1, u_2, \ldots, u_m, O_{g_a}) \circ (v_1, v_2, \ldots, v_n, O_{g_b}) \\ &= (B : v_1, \ldots, B : v_n, B : O_{g_b}, u_2 : O_{g_b}, \ldots, u_m : O_{g_b}, O_{g_a}) \end{aligned}$$

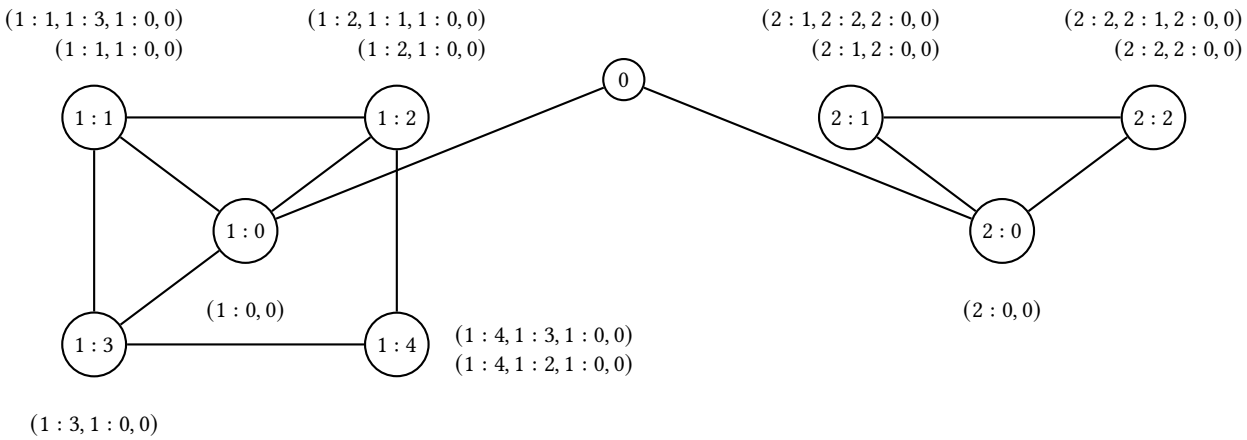


Figure 6: Rooted Combination Using Good and Disagree Gadgets

An example using Disagree Gadget as g_a and Good Gadget as g_b to create a nested combination is shown in Figure 7. As an example of generating the image of the paths of a gadget, the good gadget path $(1, 3, 0)$ becomes two paths by composing the image of the disagree paths $(1, 2, 0)$ and $(1, 0)$, namely $(1 : 0, 2 : 0, 0)$ and $(1 : 0, 0)$ to the image $(1 : 1, 1 : 3, 1 : 0)$ of $(1, 3, 0)$. Thus we have:

$$(1 : 1, 1 : 3, 1 : 0) \circ (1 : 0, 2 : 0, 0) = (1 : 1, 1 : 3, 1 : 0, 2 : 0, 0)$$

and

$$(1 : 1, 1 : 3, 1 : 0) \circ (1 : 0, 0) = (1 : 1, 1 : 3, 1 : 0, 0)$$

To create larger nested combinations from more than two inputs we chain together the constructions using intermediate gadgets created as input for the next combination. For example if we want to combine gadgets g_a, g_b, g_c in a nested combination we first compute the nested combination g_{ab} using gadgets g_a and g_b, then we compute the combination using g_{ab} and g_c for the final combination.

The way these type of gadgets are constructed preserves the path preferences of the original gadgets. If path p_a was more preferred than q_a from gadget g_a, then path $p_a * p_b$ is more preferred than path $q_a * q_b$ for any paths p_b, q_b from gadget g_b. A similar argument can be made to show that path preferences are conserved for gadget g_b. Suppose that g_b diverges and we can find a trace that oscillates. For any image of g_b in the nested gadget, this oscillating trace can be translated to an oscillating trace involving only nodes of the image gadget.

Now, suppose that g_a diverges and we can find a trace that oscillates. Then, consider only the nodes that were created from O_{g_b}. This will resemble the original g_a and a similar trace can be found, thus showing divergence. Hence, if g_a or g_b diverges then g_{ab} diverges.

In our experiments we choose g_a as the gadget named in column one in the tables and all other gadgets used in the construction are chosen as the Good Gadget. We are only able to handle gadgets consisting of up to 100 nodes and 400 links, roughly a fifth of the amount of nodes for root combination before reaching the time limit of an hour. The number of paths in nested combinations grows much faster than the number of paths in rooted combinations, reaching up to 700 paths after two nested combinations on three gadgets. This seems to cause the SMT solver to become a bottleneck much faster, as the monotonicity constraints require us to iterate over every pair of paths in the gadget, making it more difficult for the solver to find a solution. The Maude implementation may find a suitable trace within the hour time if given the proper parameters as input, since the absolute number of paths does not impact the performance nearly as much as the number of nodes and links do.

Gadget	Size	(#N, #L, #P)	SMT (s)	Maude (s)
Good	2	(21, 72, 63)	4.86	**N/A**
Naughty	2	(21, 72, 71)	5.251	4.518
Disagree	2	(11, 34, 32)	0.889	0.25
Good	3	(101, 384, 623)	1744.585	**N/A**
Naughty	3	(101, 384, 687)	420.453	114.87
Disagree	3	(51, 190, 312)	96.773	2.608

Table 6: Nested Combination Results

8 Related Work

We take inspiration from related work, in particular, the Metarouting framework [7, 12] and existing Model-Checking approaches [15–17]. The main difference, however, is our goal of providing automated checks, which are sound and scalable.

Metarouting [7, 12] provides a general mathematical framework for determining whether a BGP network gadget converges. A key limitation of the original work is the lack of fully automated methods for checking for convergence. The recent paper by Daggit and Griffin [4] has formalized the theoretical framework in Agda, which provides a means for proving convergence of network gadgets in a semi-automated fashion. The framework also enables the exploration of different convergence conditions through mechanized reasoning. We take a different approach by encoding the conditions proposed as a SMT problem. This enables the fully automated verification of network gadgets.

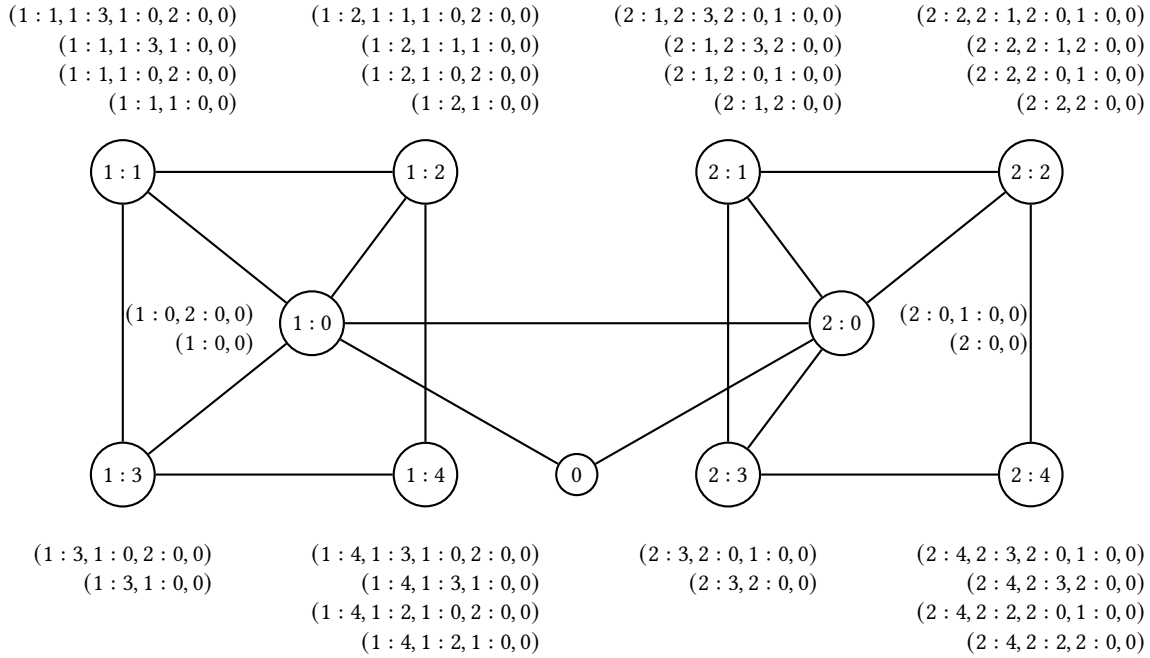


Figure 7: Nested Combination Using Disagree and Good Gadgets

The second body of work uses model-checking approaches to determine BGP convergence and also to determine witnesses for divergence. Previous work [17] formalized the BGP algorithm and the BGP problem as LTL liveness formulas in Promela. However, the experiments demonstrate the complexity of the BGP convergence problem as Promela was not able to determine divergence of gadgets with less than 5 nodes. Our proposal is to reduce the liveness problem to a safety problem as suggested in the literature [2]. Model-checkers typically perform better when proving safety properties as one can exploit different types of reduction techniques. This enabled our model-checking to verify networks with hundreds of nodes.

Wang et al. [16] proposed another formalization in Maude. However, while the performance is adequate, it is not sound. In particular, it can generate false counter examples. This seems to occur in gadgets for which BGP may sometimes converge, but sometimes also diverge. This is because the definition of divergence is not precise enough. We propose, on the other hand, a definition for determining divergence that is sound and scalable.

Wang et al. [13, 14] have also proposed methods to improve the scalability of model-checking. In [14], the authors exploit the structure of network to reduce the verification problem. For example, they identify sub-nets that are duplicated which can be reduced to a single sub-net when checking for divergence. We take an alternative approach by exploiting the unsat core of the SMT solver checking the formalization of Metarouting conditions. In [13], Wang et al. encode the BGP convergence problem in Answer Set Programming. The approach scales well reaching some thousands of nodes. However, it is not capable of providing concrete examples for divergence.

This also illustrates another key difference to existing methods. We propose a verification flow that combines SMT solver and executable specification in Maude. This enables us to exploit the advantages enabled by each method, e.g., the use of SMT solvers to simplify the model-checking problem.

9 Conclusions

This paper revisits the problem of BGP convergence verification, aiming to develop a sound, scalable, and automated verification approach. The key insight is to combine different automated verification techniques. By encoding the Metarouting convergence criterion using an SMT solver, we not only automate the convergence check but also leverage the solver's ability to generate unsatisfiable cores. These cores highlight the critical links responsible for monotonicity failing to hold, which in turn helps reduce the complexity of the model-checking problem. We reduce the model-checking task to a reachability analysis of divergent points, based on a sound divergence criterion introduced in this paper.

There are several future directions. The proposed criterion is sound, but not shown to be complete. We are also considering how to express more realistic implementations of preference relations, e.g., using pseudo-code.

Acknowledgments

This research is funded in part by the Ashton Fellowship at the University of Pennsylvania, the GEM Fellowship, and NSF IIS-2436080.

References

[1] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres

Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

[2] Armin Biere, Cyrille Artho, and Viktor Schuppan. 2002. Liveness Checking as Safety Checking. In *7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems, FMICS 2002, ICALP 2002 Satellite Workshop, Málaga, Spain, July 12-13, 2002 (Electronic Notes in Theoretical Computer Science, Vol. 66)*, Rance Cleaveland and Hubert Garavel (Eds.). Elsevier, 160–177. doi:10.1016/S1571-0661(04)80410-9

[3] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2007. *All About Maude: A High-Performance Logical Framework*. LNCS, Vol. 4350. Springer.

[4] Matthew L. Daggitt and Timothy G. Griffin. 2024. Formally Verified Convergence of Policy-Rich DBF Routing Protocols. *IEEE/ACM Trans. Netw.* 32, 2 (2024), 1645–1660. doi:10.1109/TNET.2023.3326336

[5] Lixin Gao and Jennifer Rexford. 2000. Stable Internet routing without global coordination. *SIGMETRICS Perform. Eval. Rev.* 28, 1 (June 2000), 307–317. doi:10.1145/345063.339426

[6] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. 2002. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.* 10, 2 (April 2002), 232–243. doi:10.1109/90.993304

[7] Timothy G. Griffin and João Luís Sobrinho. 2005. Metarouting. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Philadelphia, Pennsylvania, USA) *(SIGCOMM '05)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/1080091.1080094

[8] Timothy G. Griffin and Gordon Wilfong. 1999. An analysis of BGP convergence properties. *SIGCOMM Comput. Commun. Rev.* 29, 4 (Aug. 1999), 277–288. doi:10.1145/316194.316231

[9] Ratul Mahajan, David Wetherall, and Tom Anderson. 2002. Understanding BGP misconfiguration. *SIGCOMM Comput. Commun. Rev.* 32, 4 (Aug. 2002), 3–16. doi:10.1145/964725.633027

[10] Yakov Rekhter, Susan Hares, and Tony Li. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271. doi:10.17487/RFC4271

[11] Rubén Rubio. 2022. Maude as a Library: An Efficient All-Purpose Programming Interface. In *Rewriting Logic and Its Applications: 14th International Workshop, Revised Selected Papers*. Springer-Verlag, Berlin, Heidelberg, 274–294. doi:10.1007/978-3-031-12441-9_14

[12] João Luis Sobrinho. 2003. Network routing with path vector protocols: theory and applications. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Karlsruhe, Germany) *(SIGCOMM '03)*. Association for Computing Machinery, New York, NY, USA, 49–60. doi:10.1145/863955.863963

[13] Anduo Wang and Zhijia Chen. 2019. Internet Routing and Non-monotonic Reasoning. In *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11481)*, Marcello Balduccini, Yuliya Lierler, and Stefan Woltran (Eds.). Springer, 51–57. doi:10.1007/978-3-030-20528-7_5

[14] Anduo Wang, Alexander J. T. Gurney, Xianglong Han, Jinyan Cao, Boon Thau Loo, Carolyn L. Talcott, and Andre Scedrov. 2014. A reduction-based approach towards scaling up formal analysis of internet configurations. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*. IEEE, 637–645. doi:10.1109/INFOCOM.2014.6847989

[15] Anduo Wang, Carolyn Talcott, Alexander J. T. Gurney, Boon Thau Loo, and Andre Scedrov. 2012. Reduction-based formal analysis of BGP instances. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Tallinn, Estonia) *(TACAS'12)*. Springer-Verlag, Berlin, Heidelberg, 283–298. doi:10.1007/978-3-642-28756-5_20

[16] Anduo Wang, Carolyn Talcott, Limin Jia, Boon Thau Loo, and Andre Scedrov. 2011. Analyzing BGP instances in Maude. In *Proceedings of the Joint 13th IFIP WG 6.1 and 30th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems* (Reykjavik, Iceland) *(FMOODS'11/FORTE'11)*. Springer-Verlag, Berlin, Heidelberg, 334–348.

[17] Ping Yin, Yinxue Ma, and Zhe Chen. 2014. Model Checking the Convergence Property of BGP Networks. *J. Softw.* 9, 6 (2014), 1619–1625. doi:10.4304/JSW.9.6.1619-1625

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

Formalizing Languages with Binding in Rewriting Logic

Maribel Fernández
Maribel.Fernandez@kcl.ac.uk
Department of Informatics, King's College London
London, UK

José Meseguer
meseguer@illinois.edu
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA

ABSTRACT

Formalizing languages with binders (such as programming languages and logics) within a logical framework with zero representational distance (i.e., the calculus and its representation look the same) poses non-trivial challenges, including: faithful representation of syntax and binding operators; support for calculus-specific equivalences; faithful representation of the language dynamics (operational semantics); and generation of correct-by-construction implementations. We show how rewriting logic can meet these challenges. More precisely, we show how a general notion of binder signature can be axiomatized in rewriting logic and we propose a general methodology to specify languages with binders, including their structural congruences and operational semantics. We also show how rewriting logic methods provide executable specifications of languages with binders. We use the π-calculus as a running example because it illustrates well the above-mentioned challenges.

CCS CONCEPTS

• **Theory of computation → Logic and verification**; **Lambda calculus**; **Equational logic and rewriting**.

KEYWORDS

Binding operator, Nominal Logic, Rewriting Logic, Equational Theories

ACM Reference Format:
Maribel Fernández and José Meseguer. 2025. Formalizing Languages with Binding in Rewriting Logic. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Languages with binders appear extensively in mathematics and computer science: formal calculi, logics with quantifiers and programming languages are well-known examples. Various general frameworks for specifying them and for reasoning about their properties have been proposed, including, for example, frameworks based on dependent type theory or higher-order logics (e.g., [30, 42, 57, 81]), on nominal theories (e.g., [17, 46, 87, 88]), on higher-order rewriting (e.g., [64, 68]). Each of these frameworks provides an answer to the question: *what is a calculus with binders?* This raises the issue of a framework's *expressiveness* in four different ways: (i) its expressiveness *in principle* to represent certain calculi features, (ii) its *representational distance*, i.e., how different is a calculus from its framework representation; (iii) its *computational expressiveness* to derive correct-by-construction implementations of a calculus from its formal framework specification; and (iv) its support for *meta-logical* reasoning, i.e., to reason about the properties of a calculus using its framework representation. In this work, we propose rewriting logic as an expressive meta-logical framework for calculi with binders in these four senses. This is not a new idea, since rewriting logic is a logical [66, 67] and meta-logical [15, 82] framework that has been used extensively to represent calculi with binders [5, 36, 95, 97, 99], logics with quantifiers and sequent systems [67, 82], and programming languages [75, 76, 93]. However, to the best of our knowledge, a general study on how to represent and mechanize general calculi with binders in rewriting logic, which is the subject of this paper, has not been carried out so far.

Regarding expressiveness dimensions (i)–(ii), in §3 we show in general and illustrate with the running example of the π-calculus that *features such as a calculus' context-free syntax, its calculus-specific equivalences besides α-equivalence, and its operational semantics can be specified in rewriting logic with* 0 *representational distance*. Regarding expressiveness dimensions (iii)-(iv), in §4 we show that the wealth of reasoning methods about equational rewriting, e.g., [11, 49, 61, 72, 83] and coherence of rules with respect to equations modulo axioms [34, 104] available in rewriting logic support meta-logical reasoning to derive correct-by-construction implementations in calculi that are not obviously executable. Furthermore, since rewriting logic has initial models and contains equational logic as a sublogic, we explain in §6 that inductive theorem proving and model checking tools are available to reason about the initial models of calculi with binders as rewrite theories.

The general notion of syntax with binding we introduce can be tailored to be used with different kinds of names, binding operators and structural congruences. For example, in a polymorphic λ-calculus we can have different sort names for term variables and for type variables, with corresponding binders. To represent this in a direct way, our order-sorted specifications are parametric on the types of names. Different approaches can be used to axiomatize α-equivalence in rewriting logic. In § 3, *we take inspiration from the nominal approach [44, 46, 89] and define α-equivalence via conditional equations using swapping and freshness operators*. Being first-order (indeed, nominal logic [86] is defined as a theory in first-order logic, see § 3.1), the nominal approach fits well with rewriting logic. Alternative first-order specifications of α-equivalence could also be used, for example using the dual notion of free name, or using renaming (or substitution) instead of swapping (for a detailed description of the various combinations that can be used see [90]).

PPDP '25, September 10–11, 2025, Rende, Italy

ACM ISBN 979-8-4007-2085-7…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Approaches that rely on higher-order logics can be represented in rewriting logic, but the specifications are more involved (see § 6).

We illustrate the methodology by giving a full specification of Milner's π-calculus [79] — the first, to our knowledge, that takes into account not only the binding behaviour of its operators, but also their structural congruence and dynamic behaviour.

Summarizing, the main contributions are:

- *A calculus-generic methodology to formally specify languages with binders that addresses the above-mentioned challenges.* Not only the resulting specifications can be executed but also they support reasoning using a plethora of techniques and tools available for rewriting-based specifications (e.g., [10, 18, 25, 47, 50, 51, 56, 58, 60, 62, 74, 77, 80]).
- *As an application, we provide a formal specification of the π-calculus including its binding behaviour, structural congruence and operational semantics.* We show how the formalization can be made executable by equational rewriting in Maude [26], obtaining an implementation that is correct by construction and faithful: the grammar of the calculus is directly reflected in the order-sorted signature ensuring that only meaningful terms are represented. The notations used in the paper are inspired by Maude. Appendix A gives a full specification of the π-calculus in Maude. Proofs have been omitted due to space constraints.

2 PRELIMINARIES

We recall the basic notions underlying rewriting logic and the π-calculus, which we use as a running example.

2.1 Rewriting logic as a logical framework

Rewriting logic [19, 69, 70] is, simultaneously, an expressive computational logic, a logical framework, and a semantic framework for concurrency. A rewrite theory is a tuple $\mathcal{R} = (\Sigma, E, \phi, R)$, where (Σ, E) is an equational theory with order-sorted signature [53] Σ consisting of sorts $(S, <)$ and function symbols F, and Σ-equations E, R is a set of (possibly conditional) rewrite rules, and $\phi : \Sigma \to \mathbb{N}^*$ is a so-called *frozenness map* indicating, for each function symbol $f \in \Sigma$, its *frozen* argument positions, where rewriting with rules R is forbidden. As a *semantic framework* for concurrency, $\mathcal{R}$ specifies a concurrent system whose states are elements of the initial algebra $\mathbb{T}_{\Sigma/E}$ and whose local concurrent transitions are specified by the rules R and have an operational semantics that obeys the frozenness restrictions ϕ. How, for example, rewriting logic faithfully models the π-calculus will be explained in §3. As a *logical framework* [66, 67], $\mathcal{R}$ faithfully models (usually with 0 representational distance, i.e., the logic and its representation look exactly the same) the formulas (or, e.g., sequents) for the logic being represented ($\mathbb{T}_{\Sigma/E}$) and the logic's *inference rules* (R). A wide variety of logics beyond propositional and first-order logic have been represented, such as Girard's linear logic, various kinds of modal logics and sequent systems [67, 82], and Barendregt's λ-cube [97]. The two viewpoints are not incompatible. For example, the π-calculus can be seen both as a concurrent system and as a formal calculus.

Rewriting logic has been used to represent calculi with binders such as the π-calculus [95, 99], pure type systems [97], the ς-calculus [1, 95], KLAIM [36] and Orc [5]. However, to the best of our knowledge, a general study on how to represent and mechanize general calculi with binders under very general conditions on such calculi—which is the subject of this paper—has not been carried out so far. Such a general study provides a formal answer to the question: *What is a general calculus with binders?*

Equational Rewriting. Rewriting with rewrite rules R *modulo* an equational theory E is the relation $\to_{R/E} =_{def} (=_E; \to_R; =_E)$ (with $\_;\_$ relation composition, $=_E$ the provable E-equality relation, and $\to_R$ the term rewriting relation for R [31]). Although $\to_{R/E}$ is in general undecidable, *equational rewriting* techniques going back to [83] and including, e.g., [11, 49, 61, 72] provide conditions on R and E that make $\to_{R/E}$ decidable. A first requirement for executabilty of $\to_{R/E}$ is an *E-matching algorithm.*

Ground Convergence Modulo. When R are the *oriented equations* $\overrightarrow{G}$ of an equational theory $G \cup E$, $\to_{\overrightarrow{G}/E}$ is *not* deemed executable unless $\overrightarrow{G}$ is also at least *ground confluent*; that is, unless for each ground term t, if $t \to^\star_{\overrightarrow{G}/E} u$ and $t \to^\star_{\overrightarrow{G}/E} v$ there exists a ground term w such that $u \to^\star_{\overrightarrow{G}/E} w$ and $v \to^\star_{\overrightarrow{G}/E} w$, where $\to^\star_{\overrightarrow{G}/E}$ is the relation $\to^*_{\overrightarrow{G}/E} \cup =_E$, with $\to^*_{\overrightarrow{G}/E}$ the reflexive-transitive closure of $\to_{\overrightarrow{G}/E}$. This property has an immediate consequence for initial algebras, i.e., for the *statics* of calculi with binders: two $G \cup E$-equivalence classes $[t], [t'] \in \mathbb{T}_{\Sigma/G\cup E}$ are equal iff there exists a ground term w such that $t \to^\star_{\overrightarrow{G}/E} w$ and $t' \to^\star_{\overrightarrow{G}/E} w$ [61]. But, since the relation $\to_{\overrightarrow{G}/E}$ may be *non-terminating*, i.e., non-well-founded, this does not yet make $\mathbb{T}_{\Sigma/G\cup E}$ a *computable* data type, which we very much want for the statics of a calculus with binders. It does so if $\to_{\overrightarrow{G}/E}$ is: (i) ground confluent; (ii) terminating; and (iii) sort-preserving (also called "sort-decreasing"). Conditions (i)–(iii) are summarized by saying that $\to_{\overrightarrow{G}/E}$ is *ground convergent* (modulo E).

2.2 Running example: The π-calculus

We use Milner's π-calculus of mobile, concurrent processes [79] as a running example because it illustrates several *representational challenges* that do not arise in simpler calculi such as the λ-calculus.

1. Sophisticated Syntax. Disregarding name polarity (x vs. $\overline{x}$), which we view as a CCS legacy and inessential for the main ideas, the π-calculus has four syntactic categories:

(i) *Names*, $x, y, z, \ldots$, called *channels*, used both as communication channels and as *values* sent along such channels;

(ii) *π-Processes*, denoted $\pi.P$, with syntax[1]:

$$in\, x\,[y].\, P \;\mid\; out\, x\,\langle y\rangle.\, P$$

that is, π-processes have a receive or send *action prefix*, followed by a *process* continuation P (with syntax explained below) where free occurrences of y become bound in $in\, x\,[y].\, P$;

(iii) *Sum Processes*, $N, M, \ldots$, with syntax:

$$0 \;\mid\; \pi.P \;\mid\; N + M$$

that is, *sum processes* are either 0 or sums of one or more π-processes, where $\_ + \_$ denotes non-deterministic choice;

[1] To facilitate readability, we use a less Spartan π-process notation than Milner's.

(iv) *Processes*, $P, Q, R, \ldots$, the most general category, with syntax:

$$N \mid P \mid Q \mid new[x]\{P\} \mid !P$$

that is, a *process* is either a sum process, or a parallel composition $P \mid Q$ of processes, or a *new* process abstraction (binding name x in P), or a process *replication*, denoted $!P$.

2. Complex Equivalences. Besides (i) α-*equivalence* for the *in* and *new* binders, the π-calculus obeys the following additional *equivalences* defining a *structural congruence* between processes:

(ii) $\_ + \_$ is *associative and commutative* and has 0 as its *unit* element;

(iii) $\_|\_$ is *associative and commutative* and has 0 as its *unit* element;

(iv) (extrusion) $new[x]\{P|Q\} = P \mid new[x]\{Q\}$ *if* $x \notin fn(P)$ (where $fn(P)$ is the set of names free in P);

(v) $new[x]\{0\} = 0$;

(vi) $new[x]\{new[y]\{P\}\} = new[y]\{new[x]\{P\}\}$;

(vii) $!P = P \mid !P$.

3. Operational Semantics. The only computation rule is:

$$react: \quad N + out\ x\langle z\rangle.P) \mid (M + in\ x[y].Q) \to P \mid [y := z]Q$$

where $[y := z]$ replaces free occurrences of y by z in Q avoiding capture. However, the *react* rule, unlike, e.g., β-reduction, cannot be applied just anywhere inside a process. It has a precise *operational semantics*, which forbids its application below a $\_+\_$ symbol as well as in the continuation part of a π-process (see Def. 9.16 in [79]).

4. Executability Challenges. If we abbreviate by E_π the above equivalences (i)–(vii), according to Milner's specification the *react* rule is applied *modulo* E_π. Even if, as in the λ-calculus, we ignore α-equivalence for execution purposes, it is challenging to work modulo complex equivalences such as (iv), (vi) and (vii). So, as given, the π-calculus is *not* executable by standard equational rewriting techniques [61, 72], because, to the best of our knowledge, no matching algorithm modulo E_π is known – more on this in §4.2.

3 GENERAL CALCULI WITH BINDERS

We start by discussing the notion of a name, before showing how syntax with binding can be defined using order-sorted signatures.

3.1 Specifying Names of the Object Language

Names are the entities bound in calculi with binders. To specify them, we take inspiration from Nominal logic [87], which offers a formal account of names and binding: It is a many-sorted theory in first-order logic, with *sorts* τ_j including *name sorts* α_i and *abstraction sorts*, written $[\alpha]\tau$, whose elements correspond to elements of τ with a distinguished *bound* name. The signature includes function symbols $(- -) \cdot - : \alpha \times \alpha \times \tau \to \tau$ and $[-]- : \alpha \times \tau \to [\alpha]\tau$ denoting *name swapping* and *abstraction*, for any name sort α and sort τ, and predicate symbols for *equality* $(=)$ at any sort and *freshness* $(\#)$ relating any name sort and any sort. The axiom schemes of nominal logic are given in Figure 1[2].

[2]The axiom (Q) $\forall \overline{x}.(\mathit{V}a.\phi \iff \exists a.a \# \overline{x} \wedge \phi) \quad (FV(\mathit{V}a.\phi) \subseteq \overline{x})$ defining the V quantifier has been omitted since it is an abbreviation for an existential formula.

$$(a\ a) \cdot x = x \tag{S1}$$
$$(a\ a') \cdot (a\ a') \cdot x = x \tag{S2}$$
$$(a\ a') \cdot a = a' \tag{S3}$$
$$(a\ a') \cdot (b\ b') \cdot x = (((a\ a') \cdot b)\ ((a\ a') \cdot b')) \cdot (a\ a') \cdot x \tag{E1}$$
$$b \# x \Rightarrow (a\ a') \cdot b \# (a\ a') \cdot x \tag{E2}$$
$$(a\ a') \cdot f(\overline{x}) = f((a\ a') \cdot \overline{x}) \tag{E3}$$
$$p(\overline{x}) \Rightarrow p((a\ a') \cdot \overline{x}) \tag{E4}$$
$$(b\ b') \cdot [a]x = [(b\ b') \cdot a](b\ b') \cdot x \tag{E5}$$
$$a \# x \wedge a' \# x \Rightarrow (a\ a') \cdot x = x \tag{F1}$$
$$a \# a' \iff a \neq a' \tag{F2}$$
$$\forall a : \alpha, a' : \alpha'.a \# a' \qquad (\alpha \neq \alpha') \tag{F3}$$
$$\forall \overline{x}.\exists a.a \# \overline{x} \tag{F4}$$
$$[a]x = [a']x' \iff (a = a' \wedge x = x') \vee (a \# x' \wedge (a\ a') \cdot x' = x) \tag{A1}$$
$$\forall x : [\alpha]\tau.\exists a : \alpha, y : \tau.x = [a]y \tag{A2}$$

Figure 1: Axioms of (classical) Nominal Logic

The semantics of nominal logic is defined using nominal sets [88], which feature *name permutations* acting on sets. Let $\mathbb{A}$ be a countably infinite set of atoms and let G be the symmetric group on $\mathbb{A}$, which is generated by the swappings $(a\ b)$ where $a, b \in \mathbb{A}$. A *G-set* is a structure $(X, \cdot)$, where X is the carrier set and $\cdot$ the action of G on X: $\pi, x \mapsto \pi \cdot x$ such that (1) $id \cdot x = x$ and (2) $(\pi \circ \pi') \cdot x = \pi \cdot \pi' \cdot x$. A *support* of an element x of a G-set is a set S of atoms such that for all π with $dom(\pi) \cap S = \varnothing$ we have $\pi \cdot x = x$. A *nominal set* is a G-set in which all elements have a finite support, which implies that each element has a unique, least finite support $supp(x) = \bigcap\{S \subseteq \mathbb{A} \mid \forall \pi.supp(\pi) \cap S = \varnothing \Rightarrow \pi \cdot x = x\}$. Atoms that are not in $supp(x)$ are *fresh* for x.

Thus, names are described by two requirements:

(i) there should be a *countably infinite* supply of names allowing choice of *fresh* names (c.f. axiom $F4$, Figure 1), and
(ii) for any names x and y, an equality predicate can tell us whether they are equal or not.

Many data types of names satisfying requirements (i)–(ii) can be chosen, but fixing one choice a priori is not a flexible approach. It may even be the *wrong* approach, since in various calculi, such as in many-sorted first-order logic or in Martin Löef type theory with term and type variables, *different* name data types are needed. This suggests that requirements (i)–(ii) on names should be specified in a *generic* way by a *parameter theory*, which can then be *instantiated* to many choices of name data types. By the upper Löwemheim-Skolem Theorem, requiring a names data type to be *countably infinite* cannot be achieved in first-order logic. But a simple specification for requirements (i)–(ii) can be given in the following *NAME* (conditional) equational theory.

theory *NAME* **protects** *NAT, BOOL*
sort *Name*
functions : $i : Name \to Nat$, $j : Nat \to Name$, $\_ .=. \_ : Name\ Name \to Bool$
vars $a, b : Name$, $n : Nat$
equations :
$a .=. a = true$, $a .=. b = true \Rightarrow a = b$, $j(i(a)) = a$, $i(j(n)) = n$
endtheory

where the **protects** keyword specifies that the equational theories *NAT* and *BOOL*, with respective sorts *Nat* and *Bool*, have initiality constraints [32, 52] forcing the reducts to them of any algebra $\mathbb{A}$ of *NAME* to be isomorphic to, respectively, $\mathbb{N}$ and $\mathbb{B}$.

Theorem 3.1. *The models of NAME satisfy requirements (i)-(ii). Moreover, a data type of names satisfying requirements (i)-(ii) is a model of NAME. Such a data type of names will furthermore be* computable *iff it is isomporphic to a reduct to the subsignature* $\Omega = \{i, j, \_ .=. \_\}$ *of an initial algebra* $\mathbb{T}_{\Sigma/E}$ *with* (Σ, E) *convergent.*

For example, the set $\{\mathbf{x}_n \mid n \in \mathbb{N}\}$ is an instance of the *NAME* theory where $i(\mathbf{x}_n) = n$, $j(n) = \mathbf{x}_n$, and $\mathbf{x}_n .=. \mathbf{x}_m = true \Leftrightarrow n = m$.

3.2 Specifying Syntax Categories and Binders in Order-Sorted Logic

Since ADJ [54], it has been well understood that a compositional, algebraic approach to *abstract syntax* is provided by the initial algebra $\mathbb{T}_\Sigma$ of an algebraic signature Σ associated to such a syntax. But to achieve the double goals of: (i) automatically deriving user-readable implementations of calculi with binders from their executable rewriting logic specifications (as we shall illustrate in §4.2 for the π-calculus); and (ii) achieving 0 *representational distance* in those rewriting logic specifications, one should support the actual *context-free grammar* of each calculus. This can be achieved by allowing *mix-fix* syntax[3] in the names of function symbols in an algebraic signature (which we already did to specify $\_ .=. \_$ in *NAME*). The transparent correspondence between context-free grammar rules and function symbols with mix-fix syntax can be exemplified with the following grammar rule for *in* π-processes:

$$PiProc \rightarrow in\ Channel\ [Channel].\ Process$$

whose specification as a function symbol with mix-fix syntax is:

$$in\_[\_].\_ : Channel\ Channel\ \ Process \rightarrow PiProc$$

The one-to-one correspondence between context-free grammar rules and a signature's function symbol specifications is obtained by: (i) identifying a grammar's *non-terminal* symbols with the *sort names* of the associated signature; (ii) reversing the arrow's direction; and (iii) replacing the non-terminals from the right-hand side of the grammar rule by underbars indicating *argument positions*, while simultaneously displaying the corresponding non-terminals/argument sorts as the function symbol's *arity* and the left-hand side nonterminal as its *coarity*.

There is, however, a missing feature making it impossible to achieve 0 representational distance when representing context-free grammars with many-sorted algebraic signatures, namely, the incapacity to faithfully support so-called ϵ-rules in such signatures. They showed up in our description of the π-calculus syntax in BNF format by the fact that *π-processes* are a special case of *sum processes*, which are a special case of *processes*. So we have ϵ-rules $Process \rightarrow Sum \rightarrow PiProc$. In the one-to-one correspondence between non-terminals and sort names, these ϵ-rules exactly correspond to *subsort containments*, namely, $Process > Sum > PiProc$. Such containtments are impossible in many-sorted signatures but are the very *raison d'être* of *order-sorted* algebraic signatures [53]. In summary, we can exactly represent the context-free syntax of a calculus by means of the initial algebra $\mathbb{T}_\Sigma$ of the order-sorted signature Σ associated to the calculus' grammar.

But how can we specify not just algebraic operators, but *binding operators*? Quite easily, namely, by specifying what *argument positions* bind which other *argument positions* in the given operator. This can be specified by a function β that gives, for each operator f, its *binding information* $\beta(f)$. For example, the *in* operator in the π-calculus binds any occurrence of the name given as second argument within the third argument, so that $\beta(in) = (2, 3)$. Similarly, the π-calculus *new* binder has $\beta(new) = (1, 2)$, and in the λ-calculus $\beta(\lambda\_.\_) = (1, 2)$. A non-binding operator like *out* in the π-calculus has $\beta(out) = \epsilon$.

A *Binder Signature* is therefore a pair (Σ, β) of an order-sorted signature $\Sigma = ((S, <), F)$ and a function β with domain F. Naturally, the question arises as to what constraints should we impose on β. Several notions of signature with binders enjoying various degrees of generality have been proposed, e.g., [3, 43, 48, 101]. For the sake of a simpler exposition we present here a notion of binding signature that, although more general than those in [3, 43, 48], does not cover operators with nested binding behaviour. We give a more general notion of binding signature covering nested binders in §5.

First of all, a signature should itself be *parametric* on one or more copies of the *NAME* theory. Let us denote by $Name_1, \ldots, Name_k$ the corresponding parameter sorts in those copies of *NAME*. Second, for greater expressiveness we should consider three kinds of binding relationships: (i) binding a *single* name; (ii) binding a *tuple* of names; and (iii) binding a *non-empty* (*Ne*) *list* of names. In order-sorted algebra, the fact that these three possibilities are increasingly general is reflected in the existence, for each parameter sort $Name_i$, $1 \leq i \leq k$, and for each $m > 1$, of subsort inclusions $Name_i < m.Tuple_i < NeList_i < List_i$. Therefore, we should assume that in the given calculus there may also exist a finite number of sorts of the forms $m.Tuple_i$ and $NeList_j$ for various $m \in \mathbb{N}$ and $i, j \in \{1, \ldots, k\}$. The name-parametric data types $List_i$ abbreviate the instantiation of the parameterized data type *List* to $Name_i$, $1 \leq i \leq k$. *List* has an associative concatenation operation $\_\_$ with identity *nil*, so that list matching is modulo associativity and identity. Furthermore, *List* has a list membership predicate $x \in L$ equationally defined in the usual way. Besides what we call *potentially binding sorts*,[4] namely, the just-described name, or name-tuple, or (non-empty) name-list sorts[5], a calculus will typically have other *calculus-specific* sorts. Sorts in these two categories will determine the calculus' sort set S, which will have a specific *subsort inclusion relation*, giving us a sort poset $(S, <)$. The calculus' *operators* will then be typed function symbols $f \in F$ in an order-sorted signature $\Sigma = ((S, <), F)$. We will be done if we explain how $\beta(f)$ is defined for each $f \in F$.

Definition 3.2 (Binding Function β and Binding Signature). For any operator $f : s_1 \ldots s_n \rightarrow s$ in Σ, $\beta(f) \in (\mathbb{N} \times \mathbb{N}^+)^*$ is a list of pairs, each of the form $(i, j_1 \ldots j_m)$, with $j_1 < \ldots < j_m$, and with $i < i'$

[3] "mix-fix" syntax means function symbol names like $\_.=.\_$ above, or $if\_then\_else\_fi$, where argument positions are indicated by underbars and any character symbols (or none) are allowed in between.

[4] For example, the first argument in the π-calculus' *in* binder is a channel, but it does *not* bind anything.

[5] We can also allow lists with names of various kinds in a parametric sort $NeList_{1..k}$. All we say in what follows allows this possibility. For example, k-sorted first order logic quantification could use $List_{1..k}$.

if the pair $(i, j_1 \dots j_m)$ precedes $(i', j'_1 \dots j'_{m'})$ in the list $\beta(f)$. In each pair $(i, j_1 \dots j_m)$, sort s_i is a potentially binding sort, and sorts $s_{j_1} \dots s_{j_m}$ are those whose names from s_i are *bound*. $\beta(f)$ satisfies the *disjointness condition* that each index $l \in \{1, \dots, n\}$ appears *at most once* in the list of pairs $\beta(f)$, i.e., each position either binds only once some others, or is bound only once by some other, or is neither binding not bound. If $\beta(f) = \epsilon$, f is a *non-binder*.

The binding signature for a calculus $\mathscr{C}$, denoted $\Sigma_{\mathscr{C}}$, is a pair $\Sigma_{\mathscr{C}} = (((S, <), F), \beta)$ consisting of an order-sorted signature and a binding function.

For example, if $\omega : A\ Name\ B\ 3.Tuple\ C\ List\ D\ E\ Name\ G \to H$ and $\beta(\omega) = (2,1)\ (4,3\ 5)\ (6,7\ 10)$ then the second argument (of sort *Name*) binds the first (of sort *A*), the 4th argument (of sort 3.*Tuple*) binds the 3rd and 5th (of sorts *B* and *C*), and the 6th (of sort *List*) binds arguments of sorts *D* and *G*, whereas the 8th and 9th arguments (of sorts *E* and *Name*) are neither bound nor binding.

Remark 1 (Notational Convention). *Order-sorted algebra allows* subsort overloading *of operators [53]. To simplify the exposition below we will assume that: (i) for each binding operator $f : s_1 \dots s_n \to s$ having binding positions $i_1 < \dots < i_m$ of corresponding name sorts $l_i, \dots, l_m$, the signature Σ also has a more general typing of f of the form: $f : s_1 \dots s'_{i_1} \dots s'_{i_m} \dots s_n \to s$, where, $s'_{i_j} = List_{l_j}$. I.e., binding sorts are generalized to be name lists of the corresponding name sort, with β the same for* all *overloadings; and (ii) without loss of generality, up to a permutation of argument positions, the most general typing of any binding operator in Σ is of the form $f : List_1\ \overline{B}_1 \dots List_k\ \overline{B}_k\ \overline{B}_{k+1} \to C$, where for $1 \le i \le k$, each $List_i$ is a name-list sort, which binds all sorts in the next sequence of sorts $\overline{B}_i$, and that all neither bound nor binding sorts are exactly those in the sort list $\overline{B}_{k+1}$. This makes β easier to specify, as the name argument is followed by the arguments where the name is bound. For example, if $f : List_1\ s\ List_2\ s\ s \to s'$ is a binding operator with $\beta(f) = (1,2)(3,4)$, the argument of sort $List_1$ is a list of names that are bound in the second argument (of sort s), and the argument of sort $List_2$ is a list of names that are bound in the fourth argument (also of sort s); the fifth argument is also of sort s but has no bound names.*

3.3 Structural Congruence

Calculi with binders often include structural equivalences, which need to be formalized together with the α-equivalence relation generated by the binding operators. For example, in the π-calculus $\_ + \_$ is associative and commutative. Rewriting logic is ideally suited for the formalization of structural congruences: it is sufficient to include the equations defining the congruence as part of the rewrite theory. In this section we show how to define the structural congruence $E_{\mathscr{C}}$ for a calculus $\mathscr{C}$ in a generic way.

Any calculus $\mathscr{C}$ with binders has an associated calculus-generic α-equivalence defined by equations $E^{\alpha}_{\mathscr{C}}$ (see §3.3.3). As illustrated by equations (ii)–(vi) in the π-calculus (see §2.2), a calculus $\mathscr{C}$ may also have other *calculus-specific* equivalences $E^{cs}_{\mathscr{C}}$ (see §3.3.2). In addition, we often need other calculus-generic equations $E^{aux}_{\mathscr{C}}$ defining *auxiliary functions*, e.g, name swapping, a freshness predicate, renaming or substitution operations (see §3.3.1). But not all calculi need all these auxiliary equations. For example, in the π-calculus only *renaming* (as opposed to substitution) equations are needed.

The equations $E_{\mathscr{C}} = E^{\alpha}_{\mathscr{C}} \cup E^{cs}_{\mathscr{C}} \cup E^{aux}_{\mathscr{C}}$ define $\mathscr{C}$'s *structural congruence*. For a calculus $\mathscr{C}$ with signature Σ, the Σ-*congruence* defined by a set of equations $E_{\mathscr{C}}$ is the provable equality relation $=_{E_{\mathscr{C}}}$ on the set of Σ-terms (i.e., $\mathbb{T}_{\Sigma/E}$, see § 2.1).

3.3.1 Auxiliary Functions and Associated Equivalences $E^{aux}_{\mathscr{C}}$. The following are standard examples of auxiliary operators used to define calculi with binders (not all may be needed).

Swapping. We define a swapping operation $(\_\ \_) \cdot \_$ for each name sort $Name_i$ $(1 \le i \le k)$. It is specified by the equational axiom $(a\ a') \cdot a = a'$ which corresponds to $(S3)$ in Figure 1. It extends homomorphically to terms of the form $f(t_1, \dots, t_n)$ where f could be a binding or a non-binding operator (cf axiom $(E3)$ in Figure 1).

$$(a\ b) \cdot f(t_1, \dots, t_n) = f((a\ b) \cdot t_1, \dots, (a\ b) \cdot t_n)$$

Thus, a swapping acts on ground terms as expected.

Theorem 3.3. *The congruence generated by the equations above on ground terms satisfies axioms $(S1), (S2), (E1)$ in Figure 1.*

We consider the rest of the nominal logic axioms later in this section, after defining freshness and the α-equivalence relation. Note that as specified in axiom $(E5)$ in Figure 1, the swapping operators act uniformly on all the arguments, without distinguishing bound or unbound names. For example, if $f : List_1 B \to C$ is a binding operator with $\beta(f) = (1,2)$ (i.e., the names occurring in the first argument of f are bound in the second argument) and a, b, c are pairwise different names then $(a\ b) \cdot f(ac, g(a)) = f(bc, g(b))$.

Freshness. For each sort B in Σ and each $1 \le i \le k$ we also define a freshness operator $\_ \#\ \_ : Name_i\ B \to Bool$ indicating whether name a in $Name_i$ is fresh in a term of sort B. There are three cases, depending on whether the term in the second argument is a name b in $Name_i$, is rooted by a binding operator $f : List_1\ \overline{B}_1 \dots List_j\ \overline{B}_k\ \overline{B}_{k+1} \to C$, or by a non-binding operator g (including constants g such as names in $Name_j$ with $i \neq j$):

$$\begin{array}{lcl} a \# b & = & not(a .=. b) \\ a \# f(L_1, \overline{t_1}, \dots, L_k, \overline{t_k}, \overline{u}) & = & (a \in L_1 \vee a \# \overline{t_1}) \wedge \dots \\ & & \wedge (a \in L_k \vee a \# \overline{t_k}) \wedge a \# \overline{u} \\ a \# g(\overline{u}) & = & a \# \overline{u} \end{array}$$

For example, if $f : List_1\ B\ Name_i \to C$ is a binding operator with $\beta(f) = (1,2)$ and a, b are different names, then $a \# f(ab, g(a), b)$ holds (since a is bound in the second argument), but $b \# f(ab, g(a), b)$ does not hold since b occurs unbound in the third argument.

The second equation ensures that binding operators "abstract" the bound names in the nominal logic sense. Indeed, a binding operator $f : List_1\ \overline{B}_1 \dots List_j\ \overline{B}_k\ \overline{B}_{k+1} \to C$ can be understood as a combination of a function symbol f and abstractions, so that in $f(l_1, \overline{t_1}, \dots, l_k, \overline{t_k}, \overline{u})$ the names in l_i are abstracted in $\overline{t_i}$ $(1 \le i \le k)$ and no names are abstracted by f in $\overline{u}$. Recall from §3.1 that a name is fresh for t if it is not in $supp(t)$, and $supp([a]t) = supp(t) \setminus \{a\}$, which is why we check $a \in L_i$ or $a \# t_i$ for each $1 \le i \le k$ in the second equation. The third equation can be seen as a particular case of the second one if $k = 0$, i.e., f is non-binding.

Theorem 3.4. *The congruence generated by the equations above satisfies axioms $(F2), (F3), (F4)$ in Figure 1.*

Note that equation $(E2)$ in Figure 1 is satisfied for any ground term, as $b \# t$ implies that b can only occur abstracted in t. Equation $(E4)$ is not relevant since we do not distinguish predicate symbols and function symbols in the signature (predicate symbols are just function symbols where the output sort is Bool).

Renaming. For each pair of sorts $Name_i$ and B, the operators $[\_ := \_]\_ : Name_i\ Name_i\ Name_i \to Name_i$, and $[\_ := \_]\_ : Name_i\ Name_i\ B \to B$, are defined by the equations (with a, b, c of sort $Name_i$): (i) $[a := b]\, c = \mathit{if}\ a \doteq c\ \mathit{then}\ b\ \mathit{else}\ c$; (ii) ($f$ non-binder) $[a := b]\, f(t_1, \ldots, t_n) = f([a := b]\, t_1, \ldots, [a := b]\, t_n)$; (iii) otherwise, $[a := b]\, f(l_1, \bar{t}_1, \ldots, l_m, \bar{t}_m, \bar{u}) = f(l'_1, \overline{t'}_1, \ldots, l'_m, \overline{t'}_m, [a := b]\, \bar{u})$, where $[a := b]\, \bar{u}$ denotes the pointwise application of $[a := b]$ to term tuples and $(l'_i, \overline{t'}_i) = \mathit{if}\ a \# l_i\ \mathit{then}\ (\mathit{if}\ b \# l_i\ \mathit{then}\ (l_i, [a := b]\, \bar{t}_i)\ \mathit{else}\ ((b\ c)l_i, [a := b](b\ c)t_i)\ \mathit{else}\ (l_i, \bar{t}_i)$, where $c \# l_i, t_i$.

Due to lack of space we do not include calculus-generic equations to specify free names, and substitution.

3.3.2 Calculus-specific equivalences $E^{cs}_{\mathscr{C}}$. Examples of calculus specific equivalences are (ii)–(vii) in the π-calculus (see §2.2.2). In a specification of first-order logic, E^{cs}_{FOL} may contain, e.g., associativity, commutativity, idempotency and unit element (resp. $\bot$ and $\top$) equations for $\vee$ and $\wedge$.

3.3.3 α-Equivalence $E^{\alpha}_{\mathscr{C}}$. To complete the specification of $E_{\mathscr{C}}$, we now define $E^{\alpha}_{\mathscr{C}}$. Note that, if $E^{cs}_{\mathscr{C}} \neq \emptyset$, the α-equivalence relation does not exist in isolation from $E^{cs}_{\mathscr{C}}$. What exists is the $\Sigma_{\mathscr{C}}$-congruence $=_{E_{\mathscr{C}}}$ and the equations $E^{\alpha}_{\mathscr{C}}$. The fact that provable equality is a $\Sigma_{\mathscr{C}}$-congruence is *automatically guaranteed* for *any* $\Sigma_{\mathscr{C}}$-equational theory by the inference rules of equational logic.

The α-equivalence equations $E^{\alpha}_{\mathscr{C}}$ can be defined using swappings (or permutations) and freshness checks, or using renamings (or substitution) instead of swapping and a function to compute free names instead of fresh names, or other combinations (swapping and free-names function, substitution and freshness operators, ...): see [90] for a detailed analysis of the various alternatives available. Here, we define α using swapping and freshness; more precisely, for a calculus $\mathscr{C}$ with binding signature $\Sigma_{\mathscr{C}} = ((S, <), F, \beta)$, $E^{\alpha}_{\mathscr{C}}$ includes a conditional equation (α) for each binding operator $f : List_1\ \bar{B}_1 \ldots List_j\ \bar{B}_j\ \bar{B}_{j+1} \to C$ where $List_i$ is a name-list sort and $\bar{B}_i = B_{i1} \ldots B_{ik}$:

$$(\alpha)\quad f(\bar{s}, LaL', t_1, \ldots, t_k, \bar{u}) \;=\; f(\bar{s}, LbL', (a\ b) \cdot t_1, \ldots, (a\ b) \cdot t_k, \bar{u}) \quad \mathtt{if}\ a \# L' \wedge b \# L, L', t_1, \ldots, t_k$$

where in the term LaL' of sort $List_i$, the variables L and L' (also of sort $List_i$) will be instantiated by (possibly empty) lists of names (recall that list concatenation is denoted by juxtaposition and obeys associativity and identity axioms), a and b are variables ranging over names, and $t_1 : B_{i1}, \ldots, t_k : B_{ik}$.

The equations $E^{\alpha}_{\mathscr{C}}$ identify terms in the expected way, as illustrated by the following example. Assume names $\mathbf{x}_0, \ldots, \mathbf{x}_n, \ldots$ in a concrete instance of the parameter theory *NAME*. To show $f(\mathbf{x}_0, g(\mathbf{x}_0), \mathbf{x}_1, h(\mathbf{x}_1)) = f(\mathbf{x}_3, g(\mathbf{x}_3), \mathbf{x}_4, h(\mathbf{x}_4))$, where $\beta(f) = (1,2)(3,4)$, that is, $\mathbf{x}_0$ is bound in the second argument and $\mathbf{x}_1$ in the fourth, first note that by direct application of (α): $f(\mathbf{x}_0, g(\mathbf{x}_0), \mathbf{x}_1, h(\mathbf{x}_1)) = f(\mathbf{x}_5, g(\mathbf{x}_5), \mathbf{x}_1, h(\mathbf{x}_1))$ since $\mathbf{x}_5$ is a fresh name and $(\mathbf{x}_0\ \mathbf{x}_5) \cdot g(\mathbf{x}_0) = g(\mathbf{x}_5)$.

Similarly, $f(\mathbf{x}_5, g(\mathbf{x}_5), \mathbf{x}_1, h(\mathbf{x}_1)) = f(\mathbf{x}_5, g(\mathbf{x}_5), \mathbf{x}_4, h(\mathbf{x}_4))$, and $f(\mathbf{x}_5, g(\mathbf{x}_5), \mathbf{x}_4, h(\mathbf{x}_4)) = f(\mathbf{x}_3, g(\mathbf{x}_3), \mathbf{x}_4, h(\mathbf{x}_4))$. The result follows by transitivity.

Repetitions of bound names are permitted: for example, assuming that c, d, e are pairwise different and $\beta(f) = (1,2)$, we can deduce $f(cc, g(c)) = f(dd, g(d))$ by equational reasoning as follows. First, by direct application of (α), $f(cc, g(c)) = f(ce, g(e))$ since $e \# c, g(c)$, and similarly $f(ce, g(e)) = f(de, g(e))$ and $f(de, g(e)) = f(dd, g(d))$. We conclude again by transitivity.

From a nominal logic point of view, the equations (α) ensure that the axiom $(A1)$ in Figure 1 is satisfied: in the particular case in which f has two arguments, a name and a term, as the abstraction symbol in nominal logic, then (α) is exactly $(A1)$. In the context of (α), the swapping and freshness operations defined in §3.3.1 satisfy axiom $(F1)$. Hence the following theorem follows.

Theorem 3.5. *The α-equivalence relation generated by (α) for a signature where the only binding operator is $[\_]\_$ with $\beta([\_]\_) = (1,2)$ (as in nominal logic) coincides with the equality relation of nominal logic.*

The only axiom from Figure 1 that we have not yet mentioned is $(A2)$, which holds assuming abstraction sorts are defined in S with a constructor symbol $[\_]\_$ and no operators have an abstraction sort as output.

3.4 Dynamics: Rules and Operational Semantics

Given a calculus $\mathscr{C}$ with binders, its signature $\Sigma_{\mathscr{C}}$ together with its equations $E_{\mathscr{C}}$ define the calculus' *statics*. That is, the *states* of calculus $\mathscr{C}$ are precisely the elements of the *initial algebra* $\mathbb{T}_{\Sigma_{\mathscr{C}}/E_{\mathscr{C}}}$. But such a calculus would lack a purpose without a *dynamics*, which is defined by *rules*, such as *react* for the π-calculus, and β and η for the λ-calculus. But what is the *operational semantics* of rules R? It is exactly defined by the rewrite relation $\rightarrow^{\phi}_{R/E_{\mathscr{C}}}$, which, by definition, holds between $\Sigma_{\mathscr{C}}$-terms u, v iff there exist u', v' such that: (i) $u =_{E_{\mathscr{C}}} u'$ and $v =_{E_{\mathscr{C}}} v'$, and (ii) $u' \rightarrow^{\phi}_{R} v'$, where the relation $\rightarrow^{\phi}_{R}$ restricts the standard *term-rewriting* relation $\rightarrow_R$ (see, e.g., [31]) by forbidding rewriting with R at frozen positions [19]. That is, $\rightarrow^{\phi}_{R/E_{\mathscr{C}}}$ rewrites terms *modulo* the equations $E_{\mathscr{C}}$, forbidding reductions at certain *frozen* positions. For example, in the π-calculus the react rule cannot apply inside a prefix *in*, so $\phi(in) = \{1,2,3\}$. The map ϕ and the rules $R_{\mathscr{C}}$ specify a small-step SOS for $\mathscr{C}$[6].

3.5 Calculi with Binders as Parameterized Rewrite Theories

Sections 3.1–3.4 described how all the aspects of a calculus with binders can be fomalized with 0 representational distance in rewriting logic. We now put all these pieces together by briefly describing the rewrite theory $\mathcal{R}_{\mathscr{C}}$ formalizing a calculus $\mathscr{C}$ with binders.

The first point to observe is that $\mathcal{R}_{\mathscr{C}}$ is a *parameterized* rewrite theory $\mathcal{R}_{\mathscr{C}}\{NAME_1, \ldots, NAME_k\}$ where $NAME_1, \ldots, NAME_k$ are k disjoint copies of the *NAME* parameter theory. What this exactly means is that: (i) the *parameter theory* of $\mathcal{R}_{\mathscr{C}}$ is the disjoint union

[6] This level is *different* from that of *evaluation strategies* such as, e.g., a leftmost-outermost one. Such strategies must respect the SOS but restrict its computations.

of theories $NAME_1 \uplus \ldots \uplus NAME_k$ and there is a *subtheory inclusion map* $J : (NAME_1 \uplus \ldots \uplus NAME_k) \hookrightarrow \mathcal{R}_{\mathscr{C}}$; (ii) as explained in §3.1, we can instantiate such a parameter theory by choosing (set-theoretically *disjoint*, to avoid confusion between different kinds of names) *computable algebraic data types* $\mathbb{T}_{\Sigma_i/E_i}$, with E_i convergent, and theory interpretation maps $H_1 : NAME_1 \to (\Sigma_i, E_i)$, $1 \leq i \leq k$, with each (Σ_i, E_i) *initially constrained*; (iii) the corresponding *instantiation* of $\mathcal{R}_{\mathscr{C}}\{NAME_1, \ldots, NAME_k\}$ by the theory interpretation map $H_1 \uplus \ldots \uplus H_n : (NAME_1 \uplus \ldots \uplus NAME_k) \to ((\Sigma_1, E_1) \uplus \ldots \uplus (\Sigma_k, E_k))$, denoted $\mathcal{R}_{\mathscr{C}}\{H_1, \ldots, H_k\}$, is defined as the *pushout* in the category of rewrite theories [32, 52] of the theory interpretation map $H_1 \uplus \ldots \uplus H_n$ along the parameter theory inclusion map J; and (iv) since rewriting logic has *initial models* [19, 69], the *initial model semantics* of calculus $\mathscr{C}$ as instantiated by $H_1 \uplus \ldots \uplus H_n$ is precisely the initial model of the rewrite theory $\mathcal{R}_{\mathscr{C}}\{H_1, \ldots, H_k\}$. This initial model has a very intuitive meaning: it is a *transition system*, already described in §3.4, whose states are the elements of the initial algebra $\mathbb{T}_{\Sigma_{\mathscr{C}}/E_{\mathscr{C}} \cup E_H}$, where, by definition, E_H is $E_1 \cup \{x.=.y = H_1(x.=.y)\} \uplus \ldots \uplus E_k \cup \{x.=.y = H_k(x.=.y)\}$, where $x.=.y = H_i(x.=.y)$ is the *definitional extension* of $.=.$ in (Σ_i, E_i) specified by H_i, and whose one-step transition relation is defined by $\to^{\phi}_{R/E_{\mathscr{C}} \cup E_H}$. Remarks (i)–(iv) above have practical applications to deriving correct-by-construction implementations of calculi with binders. All constructions in (i)–(iv) are directly supported by the Maude language [33]. We discuss the π-calculus example in the next section.

The equations of $\mathcal{R}_{\mathscr{C}}\{NAME_1, \ldots, NAME_k\}$ are the equations $E_{\mathscr{C}}$, where we note that (i) the equations $E^{\alpha}_{\mathscr{C}}$ have extra variables in their right-hand sides and are *not* executable by equational term rewriting; (ii) the equations in $E^{aux}_{\mathscr{C}}$ and $E^{cs}_{\mathscr{C}}$ can both be *mixed bags*: some of their equations may be executable by equational term rewriting, and some (as in the π-calculus) may not be; (iii) this can make the rules $R_{\mathscr{C}}$ and therefore the calculus *not executable* by equational rewriting *as given*; (iv) however, the *semantics-preserving* theory enrichments of $\mathcal{R}_{\mathscr{C}}\{NAME_1, \ldots, NAME_k\}$ described in §4 below can make $\mathcal{R}_{\mathscr{C}}\{NAME_1, \ldots, NAME_k\}$ executable by equational rewriting or greatly reduce the set of non-executable equations, which is crucial for correct-by-construction mechanization purposes; and (v) it is useful to adopt a *modular* approach to $E^{aux}_{\mathscr{C}}$, since (a) not all auxiliary functions may be needed for a given $\mathscr{C}$, and (b) alternative versions of $\mathcal{R}_{\mathscr{C}}\{NAME_1, \ldots, NAME_k\}$ such as a nominal-based version versus a renaming- or substitution-based one may or may not require some equations.

4 CANONICAL FRESH NAME CHOICE AND EXECUTABILITY OF CALCULI

As pointed out in §3.5, not all equations in $E_{\mathscr{C}}$ are executable by equational rewriting. The intrinsic reason for the non-executability of some equations in $E_{\mathscr{C}}$, such as renaming in the π-calculus, is the need to *choose* some fresh names. This leads to equations with new variables of sort Name in the right-hand side, which are intrinsically non-executable. The good news is that all these problems evaporate in one fell swoop because we can automatically endow *any* computable algebraic data type of names $\mathbb{T}_{\Sigma/E}$, with a function *next* that *canonically chooses a fresh name*. In this way, the equations in $E_{\mathscr{C}}$ that require fresh names can be made executable by equational rewriting.

4.1 Generation of Fresh Names

The endowment of the data type of names $\mathbb{T}_{\Sigma/E}$ with a *next* function can be done in a *parametic* way using the following parameterized algebraic data type *NEXT*:

itheory $NEXT\{NAME\}$ **protects** $SET\{NAME\}, BOOL, NAT$
functions :
$\_ \gg \_ : Name\ Name \to Bool$, $max : NeSet\{NAME\} \to Name$,
$inc : Name \to Name$, $next : NeSet\{NAME\} \to Name$
vars $a, b : Name$, $S, U : Set\{NAME\}$, $Q : NeSet\{NAME\}$
equations :
$a \gg b = i(a) > i(b)$, $max(a) = a$, $max(a, a) = a$,
$max(a, b) = a$ *if* $a \gg b$, $max(a, a, Q) = max(a, Q)$,
$max(a, b, Q) = max(a, Q)$ *if* $a \gg b$, $inc(a) = j(i(a) + 1)$,
$next(a) = inc(a)$, $next(a, Q) = next(max(a, Q))$
endtheory

where the **itheory** keyword specifies that *NEXT* has an *initiality constraint* at the top, in just the same way as the imported (and protected) **itheory** $SET\{NAME\}$, which has subsort inclusions $Name < NeSet\{NAME\} < Set\{NAME\}$ and an associative-commutative union operator $\_,\_$ with unit element $\varnothing$. What *NEXT* achieves is quite simple, yet very useful: it transfers the s, $\_ > \_$, and *max* functions from $\mathbb{N}$ to *Name* using the inverse bijections i, j.

The initial algebra semantics of $NEXT\{NAME\}$ relative to a theory interpretation map of its parameter theory $H : NAME \to (\Sigma, E)$, with E convergent and (Σ, E) initially constrained, is defined exactly as in remarks (i)–(iv) in §3.5, taking into account that order-sorted equational logic is a *sublogic* of rewriting logic that also enjoys the existence of initial models. This exactly means that, as claimed, each computable name data type $\mathbb{T}_{\Sigma/E}$ is automatically endowed with a *next* function to choose fresh names and allows transforming the non-executable equations in $E^{aux}_{\mathscr{C}}$ that rely on fresh names (e.g., the equations defining renaming) into executable ones, which was the whole point of this section. But this still leaves open the, in general thornier, issue of the executability by equational rewriting of the equations $E^{cs}_{\mathscr{C}}$ and the rules $R_{\mathscr{C}}$, which we address in §4.2.

4.2 Executability and Mechanization

We first consider executability of the equational part of the rewrite theory $\mathcal{R}_{\mathscr{C}}$ of a calculus $\mathscr{C}$, i.e., the executability of its *statics*, and then the executability of its rules $R_{\mathscr{C}}$ modulo its equations, i.e., the executability of its *dynamics*. Both use *equational rewriting*.

Example: Executability of E^{cs}_{π}. Equations (ii)–(vi) in E^{cs}_{π} (see §2.2) have several non-trivial executability problems. They can be classified into:

(1) *Equational axioms*: the *associative and commutative* of $\_ + \_$ and $\_|\_$ as well as 0 as a *unit element* for both $\_ + \_$ and $\_|\_$. Let B_{π} denote these axioms. Since a matching algorithm exists for B_{π}, it is a natural choice for executing other (oriented) equations in E^{cs}_{π} by equational rewriting modulo B_{π}.

(2) *Equations executable modulo B_{π}*: The only such equation is:
(v) $new\,[x]\,\{0\} = 0$.

(3) *Equations not executable modulo B_{π}*:

(iv) $new[x]\{P|Q\} = P \mid new[x]\{Q\}$ *if* $x \notin fn(P)$;
(vi) $new[x]\{new[y]\{P\}\} = new[y]\{new[x]\{P\}\}$;
(vii) $!P = P|!P$.

Equation (iv) is non-terminating modulo B_π since, instantiating P to 0, its lefthand side B_π-matches $new[x]\{Q\}$, which rewrites to $0 \mid new[x]\{Q\}$; likewise, equation (vi) is non-terminating since its righthand side matches its lefthand side. Equation (vii) could be made terminating by orienting it as the rule $P|!P \to !P$, but this would be ill-advised, since this would thwart the very purpose of the replication operator, which, by design, generates an unbounded number of copies of P.

Is it possible to transform some of the equations in (3) into semantically equivalent convergent equations modulo B_π? The answer is yes. Order-sorted logic provides a simple solution to both the non-termination problem for extrusion and the identification of a ground convergent special case. The key idea is to declare a subsort *NeProcess* < *Process*, so that only the 0 process does not have that subsort. This allows us to replace extrusion by two semantically equivalent equations (in the context of (v) and B_π: (iv.2) subsumes (v) and is obtained from (iv) when $Q = 0$):

(iv.1) $new[x]\{P'|Q'\} = P' \mid new[x]\{Q'\}$ *if* $x \notin fn(P') \wedge x \in fn(Q')$ with P', Q' of sort *NeProcess*, and
(iv.2) $new[x]\{P\} = P$ *if* $x \notin fn(P)$.

Two useful advantages are that the pathological non-termination problem caused by instantiating P to 0 evaporates, and that (iv.2) is both executable as a ground confluent rule and subsumes (v). Second, taking advantage of the *NEXT* module we can make (vi) executable as the conditional equation:

(vi') $new[x]\{new[y]\{P'\}\} = new[y]\{new[x]\{P'\}\}$ *if* $x \gg y \wedge y \in fn(P') \wedge x \in fn(P') \setminus y$. Thus, set (2) of executable equations now has (iv.2) and (vi'), dropping (v), leaving (iv.1) and (vii) in set (3), which still blocks the executability of E_π^{cs}. To solve this problem we need an intermezzo on *coherence*.

Coherence of Rules R with Oriented Equations $\overline{G}$ Modulo Axioms B. Let us put ourselves in the best possible situation: a rewrite theory $\mathcal{R} = (\Sigma, G \cup B, \phi, R)$ where the oriented equations $\overline{G}$ are ground convergent and executable by equational rewriting modulo B. As explained in §3.4, the basic state transitions of $\mathcal{R}$ are defined by the relation $\to^{\phi}_{R/G\cup B}$. By assumption B has a matching algorithm; but $G \cup B$ might not have one. Yet, to make $\mathcal{R}$ executable we would need to execute rules R by *equational rewriting* modulo $G \cup B$, which seems impossible. Or is it? The solution to this problem is provided by ensuring that the rules R are *ground coherent* [34, 104] with the oriented equations $\overline{G}$ modulo B. This means that for each ground term t and each rewrite $t \to^{\phi}_{R/G\cup B} u$ there exists a rewrite $t!_{\overline{G}/B} \to^{\phi}_{R/B} v$ (where $t!_{\overline{G}/B}$ denotes the *normal form* of t obtained by rewriting t with $\to_{\overline{G}/B}$ to termination), such that $u =_{G\cup B} v$. This solves the problem because we then have the following equivalence:

$$u \to^{\phi}_{R/G\cup B} v \quad \Leftrightarrow \quad (\exists v')\ u!_{\overline{G}/B} \to^{\phi}_{R/B} v' \to^{\star}_{\overline{G}/B} v!_{\overline{G}/B}$$

This implies that: (i) the relation $\to^{\phi}_{R/G\cup B}$ is *decidable* (up to B-equality, finitely many such v' will exist); and (ii) it can be *executed by equational rewriting* modulo B using the much simpler relation $!_{\overline{G}/B}; \to^{\phi}_{R/B}; !_{\overline{G}/B}$, where $u(!_{\overline{G}/B})v$ iff $v = u!_{\overline{G}/B}$.

Example: Executability of $\mathcal{R}_\pi$. We left our analysis of E_π^{cs} in better shape but still unresolved: equations (iv.1) and (vii) can be applied in both directions and have an intrinsically *computational* nature: (iv.1) allows *different* kinds of applications of the *react* rule involving P: with other processes inside the *new* bubble, or with processes outside that bubble. Likewise, equation (vii) best expresses the essence of replication when used in the inflationary direction. It seems futile to try to transform them into ground convergent equations modulo B_π. Also, they make states *too abstract*.

This suggests that their computational meaning may best be expressed axiomatizing them as *additional bi-directional rules* besides *react* of the form:

extrude : $new[x]\{P'|Q'\} \to P' \mid new[x]\{Q'\}$ *if* $x \notin fn(P') \wedge x \in fn(Q')$,
intrude : $P' \mid new[x]\{Q'\} \to new[x]\{P'|Q'\}$ *if* $x \notin fn(P') \wedge x \in fn(Q')$,
$repl : !P' \to P'|!P' \qquad repl^{-1} : P'|!P' \to !P'$,

which are perfectly executable by equational rewriting modulo B_π.

As usual in many calculi, including the λ-calculus, E_α is not used in the execution itself: it is instead added as a *non-executable* set of equations (in this case two such equations).

Therefore, after importing $NEXT\{NAME\}$ and transforming the equations (iv)–(vii) into either executable equations or bi-directional rules, to show that $\mathcal{R}_\pi\{NAME\}$ is an *executable* rewrite theory it suffices to:

(1) Show ground convergence of equations (iv.2), (vi') and E_π^{aux} modulo B_π.
(2) Take as rules in R_π: *react*, *extrude*, *intrude*, *repl* and $repl^{-1}$.
(3) Show the rules in R_π are ground coherent with the oriented equations in (1) modulo B_π.

The checking of properties (1), (3) can be broken into: Task (i) checking that the oriented equations mentioned in (1) are terminating modulo B_π; Task (ii) checking that the equations in (1) are sort-decreasing and locally ground confluent modulo B_π; and Task (iii) checking that the rules in (2) are ground coherent with the oriented equations in (1) modulo B_π. These tasks can be assisted by tools such as those in [34, 35, 78]. To illustrate the techniques, in Appendix A we provide a specification of the π-calculus in Maude using swapping, which is made executable using NEXT (see § 4.1) and the techniques described in this section.

Task (i) (termination proof) turned out to be the most difficult, since it seems to be beyond the current capabilities of automated termination tools. We start with tasks (ii) and (iii).

Under the assumption that all equations in (1) terminate modulo B_π, Lemma 4.1 below proves that the equations in (1) are ground convergent, and Lemma 4.2 proves ground coherence of the rules in (2) with the oriented equations in (1) modulo B_π.

Lemma 4.1. *The oriented equations in (1) are sort-decreasing and locally ground confluent modulo B_π.*

Lemma 4.2. *Under the assumption that the equations in (1) are terminating modulo B_π, the rules in (2) are ground coherent with the equations in (1) modulo B_π.*

We now turn our attention to task (i) (termination proof). First, all equations in (1), except three of them, were proved terminating by Maude's termination assistant MTA [55] using an RPO (recursive path ordering) modulo B_π [91, 92]. The input for MTA is shown in Appendix A. MTA failed to prove the termination of equations 47, 48, 50, listed below:

$$[47]: \quad [x := y](in\ x'\ [y].P) = in\ ([x := y]\ x')[next(y, fn(P))].[x := y]((y\ next(y, fn(P))) \cdot P) \quad if\ x \neq y$$

$$[48]: \quad [x := y](new[y]\{P\}) = new[next(y, fn(P))]\{[x := y]((y\ next(y, fn(P))) \cdot P)\} \quad if\ x \neq y$$

$$[50]: \quad new[x]\{new[y]\{P'\}\} = new[y]\{new[x]\{P'\}\} \quad if\ x \gg y\ \wedge\ y \in fn(P') \wedge x \in fn(P') - \{y\}\ with\ P'\ of\ sort\ NeProcess$$

Equation 50, being permutative in nature, cannot be proved RPO-terminating automatically; but its ground RPO-termination follows from the fact that for any ground instances of x, y the order $x \gg y$ coincides with the RPO order $x > y$, because it is isomorphic to the order > on natural numbers, and this forces $new[x]\{new[y]\{P\}\} > new[y]\{new[x]\{P\}\}$ for any ground instance of P. The real difficulty resides in equations 47 and 48 (which specify renaming for binding operators *in* and *new*, swapping a bound name with a newly generated one to avoid capture), for which no precedence on the operators yields a suitable RPO orientation. To complete the proof we transform the system, replacing equations 47 and 48 with the semantically equivalent equations:

$$[47']: \quad [x := y](in\ x'\ [y].P) = in\ x''[y'].P' \quad if\ x \neq y\ \wedge\ x'' := [x := y]\ x'\ \wedge\ y' := next(y, fn(P)) \wedge P' := [x := y]((y\ y') \cdot P)$$

$$[48']: \quad [x := y](new[y]\{P\}) = new[y']\{P'\} \quad if\ x \neq y\ \wedge\ y' := next(y, fn(P)) \wedge P' := [x := y]((y\ y') \cdot P)$$

Let E_0 be the system without equations 47 and 48, and call R_0 its left-to-right oriented version, which has been proved to be RPO-terminating modulo B_π by MTA. The proof of the lemma below uses the termination of R_0 together with a well-funded induction.

Lemma 4.3. *The equations in (1) are ground terminating modulo B_π.*

Lemmas 4.1, 4.2 and 4.3 are new results, which give insights regarding the executability of the π-calculus. To the best of our knowledge this is the first time that the full calculus has been shown to be executable by rewriting (including the structural congruence, which as originally given makes it non-executable by rewriting modulo it). Ground coherence implies that the semantics does not depend on choices such as late vs early instantiation. Although not addressed in this paper, strategy expressions in Maude's strategy language can be used to specify the order in which rules should be applied [37].

5 GENERAL METHODOLOGY

In the previous sections we have described a general methodology to specify languages with binders in rewriting logic (including their structural congruences and operational semantics). Summarizing:

(1) To specify a language $\mathscr{C}$, the first step is to specify the kinds of names that are used in the language, selecting the appropriate instance of the *NAME* theory presented in § 3.1. Then, the syntax of the language can be specified directly as an order-sorted signature in rewriting logic, using the one-to-one correspondence between the context-free grammar rules of the language and the order-sorted algebraic signature, as described in § 3.2. If there are binding operators, a function β needs to be included in the specification, which provides, for each binding operator f, the information about the binding behaviour of the operator, i.e., which arguments (names or lists of names) are bound in other arguments (Definition 3.2).
(2) After specifying the binding signature, the next step is to define $\mathscr{C}$'s structural congruence, that is, the set of equations $E_\mathscr{C} = E^\alpha_\mathscr{C} \cup E^{cs}_\mathscr{C} \cup E^{aux}_\mathscr{C}$ (see § 3.3). Notice the modular specification of the congruence: $E^\alpha_\mathscr{C}$ defines the α-equivalence relation induced by the binding operators using auxiliary functions specified in $E^{aux}_\mathscr{C}$, such as name swapping, renaming, capture-avoiding substitution, etc. On the other hand, $E^{cs}_\mathscr{C}$ are calculus-specific equations and axioms (e.g., extrusion, associativity, commutativity, etc.).
(3) The steps above complete the specification of the statics of the language. The next step is to specify $\mathscr{C}$'s dynamics. This is done by defining a set R of rewrite rules, including information about frozen positions in operators, ϕ (see § 3.4). The rewrite rules induce a rewrite relation on the equivalence classes of terms generated by $E_\mathscr{C}$, that is, we obtain a transition system working on the previously defined congruence (which includes α) as expected.
(4) The final step consists of checking the executability of the equations and rules, to ensure that we have a correct-by-construction implementation of $\mathscr{C}$. The π-calculus poses non-trivial executability problems; other calculi are simpler. The general methodology presented in § 4 suggests the following steps to develop a correct-by-construction implementation of $\mathscr{C}$:
(i) if necessary, use *NEXT* to make $E^{aux}_\mathscr{C}$ ground convergent;
(ii) transform as many equations in $E^{cs}_\mathscr{C}$ as possible into ground convergent ones modulo structural axioms $B_\mathscr{C}$; if successful, then check coherence of the rules $R_\mathscr{C}$ with convergent equations modulo $B_\mathscr{C}$; and
(iii) try to transform any remaining non-ground-convergent equations in $E^{cs}_\mathscr{C}$ into bidirectional rules to get an always sound and, perhaps transforming $R_\mathscr{C}$, hopefully complete executable version of $\mathscr{C}$.
Regarding $E^\alpha_\mathscr{C}$ there are several options: we can include them in the non-orientable axioms and use a matching algorithm that takes α into account (e.g., a nominal matching algorithm [21, 102]); or, if the rewrite rules work uniformly on α-equivalence classes (i.e., they match any representative of an α-equivalence class) we can simply ignore $E^\alpha_\mathscr{C}$, as in the case of the executable rules for the π-calculus given in the Appendix; or, we could specify rewrite rules to compute a canonical form for each α-equivalence class and use R to rewrite canonical forms (as in De Bruijn implementations of the λ-calculus).

This methodology is general; in this paper we illustrated it with the π-calculus as an example, but it applies to any calculus with binders. It is highly modular: once the binding signature is specified, it induces an α-equivalence relation axiomatized by $E^{\alpha}_{\mathscr{C}}$ based on the auxiliary functions defined in $E^{aux}_{\mathscr{C}}$ (e.g., swapping and freshness, or renaming and free names); $E^{cs}_{\mathscr{C}}$ can be specified independently, by providing the axioms that describe the properties of the operators. We showed in § 3.3.3 the format of the equations for a binding operator f, with binding behaviour specified by $\beta(f)$. The notion of binding function β given in Definition 3.2 is sufficiently general to express binders such as those found in ubiquitous languages (typed λ-calculus, π-calculus, first and higher-order logic, etc.). Morever, the generation of the equations $E^{\alpha}_{\mathscr{C}}$ can be automated within the framework of rewriting logic using a *formal pattern* [71]. The latter is out of the scope of this paper; however, we can illustrate the advantages of this modular methodology by considering more general formats of binders, such as those defined by Urban et al. [101], where in addition to operators that bind a list of names in an argument (which we can already define), there are operators that bind a *set* of names, or that have a nested binding behaviour.

Generalized Binders. To formalize binding operators that bind sets of names we can use a binding function with the same format as in Definition 3.2, but specify that we have an argument which is of sort *name set* instead of name list. More precisely, if f's i-th argument is a set of names that are bound in the arguments in positions $j_1, \ldots, j_k$, all we need to do is specify that the sort of the i-th argument is a set rather than a list. Sets are lists that satisfy additional commutativity and idempotence axioms, directly specifiable in rewriting logic, as illustrated in the already used *SET* parameterized module.

The case of operators that have nested binding behaviour requires a generalization of the type of β such that $\beta(f) \in (\mathbb{N}^+ \times (\mathbb{N}^+)^+)^*$ instead of $\beta(f) \in (\mathbb{N} \times \mathbb{N}^+)^*$, that is, instead of an argument (which can be a list or set of names) having binding power on a list of argument positions, we can now specify a list of name arguments which each binds a list of arguments (so we have a list of lists in the second component of the pair). An example is in order: consider a standard *letrec* operator, which is added to the λ-calculus to simplify the definition of recursive functions, such as factorial:

letrec factorial (n) = *if* n = 0 *then* 1 *else* n * factorial (n-1)
in factorial (3)

In the case of *letrec*, the first argument is the name of the recursively defined function, the second argument is its list of arguments, the third argument is the definition of the recursive function and the fourth argument is the expression where the defined function is used. We can declare the *letrec* operator as follows:

$$letrec : Name\ List\ Exp\ Exp \to Exp$$

with binding function $\beta(letrec) = ((1\,2), (3\,4)(3))$, indicating that the first and second arguments bind respectively arguments 3, 4 and argument 3. Note the repetition of argument 3, which was not permitted in our previous definition of binding function due to the *disjointness condition* (we required that each argument appears at most once). However, in the example of *letrec* it is crucial to permit the function name (factorial in the example above) to be bound in the definition of the function (third argument) and in the expression that uses it (fourth argument); the second argument of the *letrec* operator is a list of names that is bound in the third argument only (i.e., only in the definition of the function).

Some well-formedness conditions are required on $\beta(f)$.

Definition 5.1 (Generalized Binding Function). Let $f : s_1 \ldots s_n \to s$ be a binder. $\beta(f) \in (\mathbb{N}^+ \times (\mathbb{N}^+)^+)^*$ such that for each $(\bar{i}, \bar{l}) \in \beta(f)$:

- all the positions s_j for $j \in \bar{i}$ are name positions (either a name sort, or a name tuple or name list);
- $|\bar{i}| = |\bar{l}|$;
- $\bar{l}$ satisfies the *telescope* condition: if $\bar{l} = l_1, \ldots, l_k$ then $l_1 \sqsupseteq l_2 \sqsupseteq \ldots \sqsupseteq l_k$, where $\sqsubseteq$ is the sublist relation;
- if $l_i = j_1, \ldots, j_m$ then for $1 \le p < q \le m$, $j_p < j_q$;
- and in addition a *disjointness condition* holds: the argument positions in $\bar{i}$ cannot occur in $\bar{l}$ (i.e., the first component in each pair in $\beta(f)$ specifies names that will be bound in other arguments, but cannot themselves be in the scope of another bound name) and moreover names in different pairs in $\beta(f)$ must be disjoint. More precisely, if $(\overline{i_j}, \overline{l_j}) \in \beta(f)$ and $(\overline{i_k}, \overline{l_k}) \in \beta(f)$, where $j \neq k$ then $\overline{i_j} \cap \overline{l_j} = \emptyset$, and $Pos(\overline{i_j}, \overline{l_j}) \cap Pos(\overline{i_k}, \overline{l_k}) = \emptyset$ where $Pos(\overline{l_1}, \ldots, \overline{l_k})$ denotes the set of positions mentioned in the list (or list of lists) $l_1, \ldots, l_k$.

For example, $\beta(letrec) = ((1\,2), (3\,4)(3))$ satisfies the telescope condition since $(3\,4) \sqsupseteq (3)$, and the disjointness condition since $\{1, 2\} \cap \{3, 4\} = \emptyset$.

Note that Definition 5.1 is a conservative extension of Definition 3.2: if $|\bar{i}| = 1$ for all pairs, we obtain the previous definition.

As before, from the function declaration we can automatically generate the equations $E^{\alpha}_{\mathscr{C}}$. To show this, we need to introduce notation. Assume without loss of generality that

$$f : \overline{s_{i_1}}\,\overline{s'_{l_1}} \ldots \overline{s_{i_q}}\,\overline{s'_{l_q}}\,\bar{s} \to s$$

and $\beta(f) = (\overline{i_1}, \overline{l_1}) \ldots (\overline{i_q}, \overline{l_q})$, where $\overline{s_{i_1}}, \ldots, \overline{s_{i_q}}$ are name sorts (or tuples of name sorts or lists of name sorts), and occurrences of names in those positions are bound in the arguments of sorts $\overline{s'_{l_1}}, \ldots, \overline{s'_{l_q}}$, respectively, which are arbitrary sorts, and finally the arguments of sorts $\bar{s}$ do not have bound names. More precisely, in a term $f(\bar{s}, \bar{L}, LaL', \overline{L'}, \overline{u_1} u_{j_{1_1}} \overline{u_2} u_{j_{1_2}}, \ldots, \overline{u_{n_1}} u_{j_{1_{n_1}}}, \bar{u}, \bar{v})$ where $(\bar{i}, \bar{l}) \in \beta(f)$, the names $\bar{L}, LaL', \overline{L'}$ in positions $\bar{i}$ are bound in the arguments $\overline{u_1} u_{j_{1_1}} \overline{u_2} u_{j_{1_2}}, \ldots, \overline{u_{n_1}} u_{j_{1_{n_1}}}\,\bar{u}$ in positions $\bar{l}$, and specifically the names in $L_r = LaL'$ bind the positions in $l_r = j_{1\,1} \ldots j_{1_{n_1}}$.

The generalised $E^{\alpha}_{\mathscr{C}}$ equations have the form:

$$\begin{aligned}(\alpha)\quad & f(\bar{s}, \bar{L}, LaL', \overline{L'}, \overline{u_1} u_{j_{1_1}} \overline{u_2} u_{j_{1_2}}, \ldots, \overline{u_{n_1}} u_{j_{1_{n_1}}}, \bar{u}, \bar{v}) = \\ & f(\bar{s}, \bar{L}, LbL', \overline{L'}, \overline{u_1}\,(a\ b) \cdot u_{j_{1_1}} \overline{u_2}\,(a\ b) \cdot u_{j_{1_2}} \ldots \overline{u_{n_1}}(a\ b) \cdot u_{j_{1_{n_1}}} \bar{u}, \bar{v}) \\ & \qquad \texttt{if}\ \ a \# L', \overline{L'} \wedge \\ & \qquad b \# L, L', \bar{L}, \overline{L'}, \overline{u_1}\,(a\ b) \cdot u_{j_{1_1}} \overline{u_2}\,(a\ b) \cdot u_{j_{1_2}} \\ & \qquad\qquad \ldots \overline{u_{n_1}}(a\ b) \cdot u_{j_{1_{n_1}}} \bar{u}\end{aligned}$$

Again, in the particular case in which β satisfies Definition 3.2, the equation above boils down to the equation (α) given in § 3.3.3.

At this point we may ask how general this methodology actually is. Since we are using equational logic, we inherit its power: by Bergstra-Tucker's Theorem any computable algebra can be specified with a finite number of convergent equations; but the real

question is how close the representation is to the informal notation (what we earlier called representational distance). The latter can be illustrated with examples: we have specified the π-calculus with 0 representational distance for its syntax, structural axioms, rule and operational semantics; and we have given a full executable specification for it.

6 RELATED WORK

Binding operators have been studied in a variety of contexts: to define models of computation such as the λ-calculus [14, 24], in higher-order rewriting frameworks (e.g., [64, 68]) and logical frameworks based on higher-order abstract syntax (e.g., [57, 84, 85]), etc. Standard programming languages and proof assistants do not offer built-in support for the formalisation of binding, structural congruences and transition relations. For example, a Rocq[7] formalization of the π-calculus cannot achieve 0-representational distance because: (i) the subtype inclusions used in the π-calculus would require explicit coercion operators in Rocq, changing the original syntax; (ii) the only non-beta equality available in Rocq is alpha-equality, thus requiring the formalization of π's structural equality as an equivalence relation; (iii) Rocq would require formalizing the π-calculus's transition rule and its SOS as a transition relation modulo the equivalence relation in (i). Instead, rewriting logic solves issues (i)-(iii) with 0 representational distance, as shown in the Appendix. Similar remarks apply to other proof assistants. Using rewriting logic, the notion of equivalence relation generated by a set of conditional equational axioms E is pre-defined (we do not have to prove that the equivalence relation we define is a congruence, since the equality relation $=_E$ is so by definition). Similarly, to specify a transition relation it suffices to provide a set of rewrite rules and a frozenness map ϕ. This greatly simplifies the formalization of structural congruences (such as the one in the π-calculus) and transition relations (not just for process algebras: in many formalizations of logics and type systems, it is useful to represent proof simplification rules, or to see inferences as transitions to analyse the cost of deductions for example). Indeed, describing a logic as a rewrite theory is entirely natural: the logic's inference rules are exactly the associated rewrite theory's rewrite rules: see [15, 66, 67, 82]. Rewriting logic supports many operational semantics styles: big or small step, reduction semantics, continuation semantics, CHAM semantics, etc., (see [93]).

Reasoning about languages with binding is challenging and requires specialized tools (one of the aims of the POPLMark challenge [9] was to explore and compare methods for the formalization of binding operators). Many different techniques were proposed to address the challenge. Below we discuss the works closer to ours.

A distinctive aspect of our proposal is the use of rewriting logic to provide a generic first-order definition of syntax with binding that is expressive enough to accommodate structural congruences and computation rules, and which can be executed. To specify the α-equivalence relation (possibly in the context of other equivalences), we took inspiration from the nominal approach [28, 44, 46, 89], which is also first-order. Nominal logic uses the well-understood concept of *permutation groups acting on sets* to provide a rigorous treatment of common informal practice to do with fresh and bound names. This is one of the ways in which we can represent binders in rewriting logic, we could use renaming or non-capturing substitution instead of permutations. The notion of binding operator introduced in this paper generalizes the notion of abstraction from nominal logic: As stated in Theorem 3.5, the abstraction operator $[\_]\_$ can be represented using $\beta([\_]\_) = (1, 2)$. In the other direction, our binding signatures can be encoded in a signature where the only binder is the abstraction operator (as in nominal logic) as follows: Consider a binding operator

$$f : List_1\ B_1 \dots List_j\ B_j\ \overline{C} \to D$$

where $\beta(f) = (1, 2)(3, 4)..(2j - 1, 2j)$, that is, the argument l_i of sort $List_i$ (list of names) binds the next argument (of sort B_i). This operator can be encoded using a non-binding function symbol f and abstractions in the nominal logic style. For example, $f(ab, t_1, cde, t_2, t_3)$ can be written as $f([a][b]t_1, [c][d][e]t_2, t_3)$. We aim to reduce the representational distance by avoiding encodings.

Nominal matching and nominal unification [102] are decidable and efficient algorithms exist [20–22, 65], which are the basis for efficient implementations of nominal rewriting [39–41, 98]. An important difference between nominal rewriting and the notion of rewriting modulo axioms defined in this paper is that nominal rewriting has nominal set semantics (more precisely, nominal universal algebra [45]) whereas here we interpret terms in the standard term algebra quotiented by the equational axioms. Our first-order approach inherits the advantages of the nominal approach, including the availability of decidable unification algorithms (when the equational theory is empty or satisfies known sanity properties) and the availability of induction principles that respect the variable convention. The study of matching and unification in the context of structural axioms is left for future work (some promising results are already available for nominal α-equivalence modulo associativity and commutativity [6, 7], and nominal narrowing [8]).

Nominal Isabelle [100] is an extension of Isabelle-HOL to facilitate formalization of languages with binders. Binding operators abstract one name at a time, as in nominal logic. Urban and Kaliszky [101] observe that this leads to clumsy formalizations in certain cases and propose an extension of Nominal Isabelle to deal with multiple variables bound at once. They consider three general binding mechanisms: with order, without order and including vacuous binding, and define also a notion of *deep binding* exemplified by the letrec construct in functional languages. Our binding signatures can also handle binding operators with or without order and including vacuous binding; however, the notion of deep binder requires the use of more involved β functions to provide information about the nesting of binding. To achieve this, we introduced generalized binding functions (Definition 5.1), where nested binding is specified by using "telescopes" as exemplified by the letrec operator.

Blanchette et al [17] present a category-theoretic approach to binding inspired by nominal logic but going beyond since there is no restriction to finite support. Binding operators are defined using a restricted class of bounded natural functors (BNFs) that include a "binding dispatcher" indicating which types of variables bind in which types of potential terms. This is similar to our function β, but it is defined at the level of types, not argument positions, which forces the introduction of multiple types if for example we have a binder that works on a pair of terms but binds only in the first

[7] previously known as Coq [16]

term: two different types of terms are required in this case. This general framework supports the definition of complex classes of binders and is equipped with (co)induction principles in the style of nominal logic. However, no structural equivalences are supported in this framework. Abbott et al. [2] define quotients for containers, which are closely related to the BNF-based framework. Quotient containers permit to define equivalences of labellings, which is useful to define permutative equivalences (e.g., define bags as a quotient of lists); however, the shape cannot be changed. This kind of quotient is not sufficient to define the structural equivalence relation for the π-calculus, since equivalent processes may have different shapes, not just different content.

Gheri and Popescu [48] present an Isabelle-HOL formalization of many-sorted syntax with binding using single abstractions in the nominal style and discuss recursion principles. Our formalization can be executed and thanks to the use of order-sorted signatures we avoid the embeddings required in many-sorted frameworks to map name sorts into data sorts, achieving a specification that is closer to the informal pen and paper specifications.

In the general theories of binders proposed by Fiore, Plotkin and Turi [43], binding is expressed using second-order types with a (categorical) algebraic semantics: every binding signature has an associated category of models consisting of variable sets endowed with compatible algebra and substitution structures. Our treatment is instead purely first-order, substitution is not needed (although it can be defined) and the associated category of models is the standard one. It supports inductive reasoning in calculi with binders using inductive theorem proving methods in initial algebras, e.g., [10, 50, 60, 62, 74, 80] and associated tools [18, 25, 29, 47, 51, 56, 58, 59, 63, 77], and, likewise, methods and tools for model checking and theorem proving verification of properties in initial models of rewrite theories [12, 13, 26, 38, 73, 94]. As in the nominal case, induction principles that respect the variable convention can be defined using the notion of fresh name (the study of inductive theorem proving methods is left for future work). The study of translations between higher-order abstract syntax (specifically higher-order patterns) and our syntax is also left for future work: we conjecture that the existing translations between higher-order patterns and nominal syntax (e.g. [23]) can be directly applied here.

This paper extends previous work on the formalization of the π-calculus in rewriting logic [96, 99, 103], which neither addressed π's non-executable structural equivalences nor made them coherently executable. We provided a full formalization of the π-calculus including structural equivalence, and proved that the formalization satisfies the required properties for executability.

7 CONCLUSIONS AND FUTURE WORK

We have presented a methodology for the specification of calculi with binders in rewriting logic that can express general binding mechanisms (including binding multiple names at the same time and binding different names in different arguments) and yields initial models. We have shown how calculi with binders can be specified in a modular way, using a generic notion of name and α-equivalence, which can be combined with calculus-specific equivalences and can be made executable. The α-equivalence relation is an equivalence relation by construction, and the equational axioms that define it can be automatically generated from the binding signature. Furthermore, since rewrite theories are elements of a computable data type [27] and meta-level transformations of theories can be directly represented using formal patterns (see, e.g., [71]), it would be possible to define formal patterns to support the automatic generation of the full specification of any calculus of interest. In future work we will study the use of this technique and explore connections with type-theoretic works (e.g., [4]) that also aim at facilitating the definition of generic binding calculi.

Acknowledgements. This work was partially funded by the Royal Society (Int. Exchanges IES\R2\212106) and the Leverhulme Trust (award VP2-2021-019). We thank Paco Durán, Daniele Nantes-Sobrinho and Andy Pitts for helpful comments.

A APPENDIX: EXECUTABLE π-CALCULUS

```
fmod NAT-FVP is
  sorts Nat NzNat Bool .  subsorts NzNat < Nat .
  op true : -> Bool [ctor metadata "0"] .
  op false : -> Bool [ctor metadata "0"] .
  op 0 : -> Nat [ctor metadata "0"] .
  op 1 : -> NzNat [ctor metadata "1"] .
  op _+_ : Nat Nat -> Nat [assoc comm id: 0  metadata "2"] .
  op _+_ : NzNat NzNat -> NzNat
                    [ctor assoc comm id: 0 metadata "2"] .
  op _~_ : Nat Nat -> Bool [comm metadata "3"] .
  op _>_ : Nat Nat -> Bool [metadata "3"] .
  op _or_ : Bool Bool -> Bool [assoc comm metadata "3"] .
  vars n m : Nat .  vars n' m' : NzNat .  var B : Bool .
  eq [1]: n ~ n = true [variant] .
  eq [2]: (n + m') ~ n = false [variant] .
  eq [3]: n + m' > n = true [variant] .
  eq [4]: n > n + m = false [variant] .
  eq [5]: B or true = true [variant] .
  eq [6]: B or false = B [variant] .
  eq [7]: B or B = B [variant] .
endfm

fmod NAME is protecting NAT-FVP .
 sort Name .
 op i : Name -> Nat [metadata "4"] .
 op j : Nat -> Name [metadata "4"] .
 op _.=._ : Name Name -> Bool [comm metadata "5"] .
 vars n m : Nat .  vars X Y : Name .
 eq [8]: X .=. Y = i(X) ~ i(Y) .
 eq [9]: j(i(X)) = X .
 eq [10]: i(j(n)) = n .
endfm

fmod NAME-SET is protecting NAME .
  sorts   Set NeSet .  subsorts Name < NeSet < Set .
  op mt : -> Set [ctor metadata "1"] .
  op _,_ : Set Set -> Set
                     [assoc comm id: mt metadata "2"] .
  op _,_ : NeSet NeSet -> NeSet
                    [ctor assoc comm id: mt metadata "2"] .
 op _,_ : NeSet Set -> NeSet [ctor assoc comm id: mt metadata "2"] .
 op _,_ : Set NeSet -> NeSet [ctor assoc comm id: mt metadata "2"] .
  op _\_ : Set Name -> Set [metadata "6"] .
  op _in_ :  Name Set -> Bool [metadata "7"] .
```

```
  vars S U : Set .   vars X Y : Name .  vars P Q : NeSet .
  eq [11]: (X,X) = X .
  eq [12]: mt \ X = mt .
  eq [13]: X \ X = mt .
  ceq [14]: X \ Y = X if X .=. Y = false .
  eq [15] : (X,Q) \ X = (Q \ X) .
  ceq [16]: (X,Q) \ Y = X,(Q \ Y) if X .=. Y = false .
  eq [17]: X in mt = false .
  eq [18]: X in Y = (X .=. Y) .
  eq [19]: X in (Y,Q) = (X .=. Y) or (X in Q) .
endfm

fmod SWAP is protecting NAME .
  sort Swap .
  op [_ _] : Name Name -> Swap [comm metadata "2"] .
 op _._ : Swap Name -> Name [metadata "15"] .    *** name swapping
  vars a a' b c : Name .
  eq [i]: [a a'] . a = a' .  eq [ii]: a a'] . a' = a .
 ceq [iii]: [a a'] . b = b if a .=. b = false /\ a' .=. b = false .
endfm

fmod NEXT is protecting NAME-SET .
  op _>>_ :  Name Name -> Bool [metadata "8"] .
  op max :  NeSet -> Name [metadata "9"] .
  op inc : Name -> Name [metadata "9"] .
  op next : NeSet -> Name [metadata "10"] .
 vars S U : Set .  var Q : NeSet .  vars X Y : Name .  vars n m : Nat .
  eq [20]: X >> Y = i(X) > i(Y) .
  eq [21]: max(X) = X .
  eq [22]: max(X,X) = X .
  ceq [23]: max(X,Y) = X if X >> Y = true .
  eq [24]: max(X,X,Q) = max(X,Q) .
  ceq [25]: max(X,Y,Q) = max(X,Q) if X >> Y = true .
  eq [26]: inc(X) = j(i(X) + 1)  .
  eq [27]: next(X) = inc(X) .
  eq [28]: next(X,Q) = inc(max(X,Q)) .
endfm

fmod PI-NOMINAL is protecting NEXT .  protecting SWAP .
  sorts Pi NeSum Sum NeProc Proc .
  subsorts Pi < NeSum < NeProc Sum <  Proc .
  op 0 : -> Sum [ctor metadata "0"] .
 op _|_ : Proc Proc -> Proc  [assoc comm id: 0 metadata "13"] .
  op _|_ : NeProc NeProc -> NeProc
                      [ctor assoc comm id: 0 metadata "13"] .
 op _+_ : Sum Sum -> Sum [frozen assoc comm id: 0 metadata "2"] .
  op _+_ : NeSum NeSum -> NeSum
                [ctor frozen assoc comm id: 0 metadata "2"] .
  op !_ : Proc -> NeProc [ctor frozen metadata "13"] .
  op new[_]{_} : Name Proc -> Proc [ctor metadata "14"] .
  op new[_]{_} : Name NeProc -> NeProc [ctor metadata "14"] .
                       *** new-abstraction [name restriction]
 op in_[_]._ : Name Name Proc -> Pi [ctor frozen metadata "13"] .
                *** synchronous reception with in-abstraction
 op out_<_>._ : Name Name Proc -> Pi [ctor frozen metadata "13"] .
                 *** synchronous message send along a channel
 op fn : Proc -> Set [frozen metadata "12"] .     *** free names
 op _._ : Swap Proc -> Proc  [metadata "15"] .    *** name swapping
  sort Ren .  *** renaming sort
 op [_:=_] : Name Name -> Ren [ctor metadata "2"] .   *** renaming
  op __ : Ren Name -> Name [frozen metadata "16"] .
                               *** renaming applied to names
  op __ : Ren Proc -> Proc [frozen metadata "16"] .
  op __ : Ren NeSum -> NeSum [frozen metadata "16"] .
                          *** renaming applied to processes
  vars a b c d a' b' : Name .    vars X Y Z X' Y' : Name .
 vars P Q R T : Proc .  vars P' R' Q' T' : NeProc . vars N M : Sum .
 vars N' M' : NeSum .  vars n m : Nat .  vars n' m' : NzNat .

 *** swapping homomorphic equations
  eq [iv] : [a b] . 0 = 0 .
  eq [v] : [a b] . (P' | Q') = ([a b] . P') | ([a b] . Q') .
  eq [vi] : [a b] . (P' + Q') = ([a b] . P') + ([a b] . Q') .
  eq [vii] : [a b] . ! P = ! ([a b] . P) .
 eq [viii] : [a b] . (new [c]{P}) = new [([a b] . c)]{([a b] . P)} .
  eq [ix] : [a b] . (in c [d]. P) =
                   in ([a b] . c) [([a b] . d)]. ([a b] . P) .
  eq [x] : [a b] . (out c < d >. P) =
                   out ([a b] . c) < ([a b] . d) >. ([a b] . P) .

 *** free names
 eq [29] : fn(0) = mt .
 eq [30] : fn(P' | Q') = fn(P'),fn(Q') .
 eq [31] : fn(N' + M') = fn(N'),fn(M') .
 eq [32] : fn(out X < Y >. P) = X,Y,fn(P) .
 eq [33] : fn(! P) = fn(P) .
 eq [34] : fn(in X [ Y ]. P) = X,(fn(P) \ Y) .
 eq [35] : fn(new [X]{P'}) = fn(P') \ X .

 *** renaming application for names:
 eq [36]: [X := Y] X = Y .
 ceq [37]: [X := Y] Z = Z if X .=. Z = false .

 *** renaming application for non-binders
 eq [38]: [X := Y] 0 = 0 .
 eq [39]: [X := Y] (P' | R') = ([X := Y] P') | ([X := Y] R') .
 eq [40]: [X := Y] (M' + N') = ([X := Y] M') + ([X := Y] N') .
 eq [41]: [X := Y] (out X' < Y' >. P) =
           out ([X := Y] X') < [X := Y] Y' >. ([X := Y] P) .
 eq [42]: [X := Y] (! P) = ! ([X := Y] P) .

 *** renaming application for in- and new-binders
 eq [43]: [X := Y] (in X' [ X ]. P) =
                     in ([X := Y] X') [ X ]. P .
 ceq [44]: [X := Y] (in X' [ Y' ]. P) =
               in ([X := Y] X') [ Y' ]. ([X := Y] P)
                     if X .=. Y' = false /\ Y .=. Y' = false .
 eq [45]: [X := Y] (new [X]{P}) = new [X]{P} .
 ceq [46]: [X := Y] (new [X']{P}) = new [X']{[X := Y] P}
                      if X .=. X' = false /\ Y .=. X' = false .
ceq [47]: [X := Y] (in X' [Y]. P) =
  in ([X := Y] X') [next(Y,fn(P))].([X := Y]([Y next(Y,fn(P))] . P))
                        if X .=. Y = false .
 ceq [48]: [X := Y] (new [Y]{P}) =
 new [next(Y,fn(P))]{[X := Y] ([Y next(Y,fn(P))] . P)}
                                          if X .=. Y = false .
 ceq [49]: new [X]{P} = P if X in fn(P) = false .
 ceq [50]: new [X]{new[Y]{P'}} = new [Y]{new[X] {P'}}
  if (X >> Y) = true /\ Y in fn(P') = true /\ X in (fn(P')\ Y) = true .
endfm
```

REFERENCES

[1] Martín Abadi and Luca Cardelli. 1996. *A Theory of Objects.* Springer.

[2] Michael Gordon Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. 2004. Constructing Polymorphic Programs with Quotient Types. In *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3125)*, Dexter Kozen and Carron Shankland (Eds.). Springer, 2–15. https://doi.org/10.1007/978-3-540-27764-4_2

[3] Peter Aczel. [n. d.]. A general Church-Rosser theorem. ([n. d.]). Manuscript, University of Manchester, 1978.

[4] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proc. ACM Program. Lang.* 2, ICFP, Article 90 (jul 2018), 30 pages. https://doi.org/10.1145/3236785

[5] Musab A. AlTurki and José Meseguer. 2015. Executable rewriting logic semantics of Orc and formal analysis of Orc programs. *J. Log. Algebr. Meth. Program.* 84, 4 (2015), 505–533.

[6] Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Daniele Nantes-Sobrinho, and Ana Oliveira. 2019. A formalisation of nominal α-equivalence with A, C, and AC symbols. *Theor. Comput. Sci.* 781 (2019), 3–23. https://doi.org/10.1016/j.tcs.2019.02.020

[7] Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes-Sobrinho. 2021. Formalising nominal C-unification generalised with protected variables. *Math. Struct. Comput. Sci.* 31, 3 (2021), 286–311. https://doi.org/10.1017/S0960129521000050

[8] Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. 2016. Nominal Narrowing. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11.

[9] B. E. Aydemier, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, , and S. Zdancewic. 2005. Mechanized metatheory for the masses: The POPLmark challenge. In *TPHOL 18 (LNCS)*. Springer.

[10] Leo Bachmair. 1988. Proof by Consistency in Equational Theories. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 228–233.

[11] Leo Bachmair and Nachum Dershowitz. 1989. Completion for Rewriting Modulo a Congruence. *Theor. Comput. Sci.* 67, 2&3 (1989), 173–201.

[12] Kyungmin Bae, Santiago Escobar, and José Meseguer. 2013. Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. In *Rewriting Techniques and Applications (RTA'13) (LIPIcs, Vol. 21)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 81–96.

[13] Kyungmin Bae and José Meseguer. 2015. Model checking linear temporal logic of rewriting formulas under localized fairness. *Sci. Comput. Program.* 99 (2015), 193–234.

[14] H. P. Barendregt. 1984. *The Lambda Calculus, its Syntax and Semantics*. North Holland, Amsterdam, 2nd ed.

[15] D. Basin, M. Clavel, and J. Meseguer. 2004. Rewriting Logic as a Metalogical Framework. *ACM Transactions on Computational Logic* 5 (2004), 528–576. Issue 3.

[16] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer.

[17] Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. 2019. Bindings as bounded natural functors. *Proc. ACM Program. Lang.* 3, POPL (2019), 22:1–22:34. https://doi.org/10.1145/3290335

[18] Adel Bouhoula and Michaël Rusinowitch. 1995. Implicit Induction in Conditional Theories. *J. Autom. Reason.* 14, 2 (1995), 189–235. https://doi.org/10.1007/BF00881856

[19] Roberto Bruni and José Meseguer. 2006. Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.* 360, 1-3 (2006), 386–414.

[20] Christophe Calvès and Maribel Fernández. 2008. A Polynomial Nominal Unification Algorithm. *Theor. Comp. Sci.* 403 (Aug. 2008), 285–306. Issue 2–3.

[21] Christophe Calvès and Maribel Fernández. 2010. Matching and Alpha-equivalence Check for Nominal Terms. *Journal of Comp. Syst. Sci.* 76, 5 (2010), 283–301.

[22] Christophe Calvès and Maribel Fernández. 2011. The First-Order Nominal Link. In *Logic-Based Program Synthesis and Transformation - 20th International Symposium, LOPSTR 2010, Hagenberg, Austria, July 23-25, 2010, Revised Selected Papers (LNCS, Vol. 6564)*. Springer, 234–248.

[23] James Cheney. 2005. Relating Nominal and Higher-Order Pattern Unification. In *Proceedings of the 19th International Workshop on Unification (UNIF 2005)*. 104–119.

[24] Alonzo Church. 1941. The Calculi of Lambda-Conversion. *Annals of Mathematics Studies* 6 (1941). Second edition 1951.

[25] Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. 1998. Building equational proving tools by reflection in rewriting logic. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project. http://maude.cs.uiuc.edu.

[26] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. 2007. *All About Maude – A High-Performance Logical Framework*. Springer LNCS Vol. 4350.

[27] Manuel Clavel, José Meseguer, and Miguel Palomino. 2007. Reflection in Membership Equational Logic, Many-Sorted Equational Logic, Horn Logic with Equality, and Rewriting Logic. *Theoretical Computer Science* 373 (2007), 70–91.

[28] Ranald A. Clouston and Andrew M. Pitts. 2007. Nominal Equational Logic. *ENTCS* 172 (2007), 223–257. https://doi.org/10.1016/j.entcs.2007.02.009

[29] Hubert Comon and Robert Nieuwenhuis. 2000. Induction=I-Axiomatization+First-Order Consistency. *Inf. Comput.* 159, 1-2 (2000), 151–186.

[30] Thierry Coquand and Gerard Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2–3 (Feb. 1988), 95–120.

[31] N. Dershowitz and J.-P. Jouannaud. 1990. Rewrite Systems. In *Handbook of Theoretical Computer Science, Vol. B*, J. van Leeuwen (Ed.). North-Holland, 243–320.

[32] Francisco Durán and José Meseguer. 2003. Structured theories and institutions. *Theor. Comput. Sci.* 309, 1-3 (2003), 357–380.

[33] Francisco Durán and José Meseguer. 2007. Maude's module algebra. *Sci. Comput. Program.* 66, 2 (2007), 125–153.

[34] Francisco Durán and José Meseguer. 2012. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *J. Algebraic and Logic Programming* 81 (2012), 816–850.

[35] Francisco Durán, José Meseguer, and Camilo Rocha. 2020. Ground confluence of order-sorted conditional specifications modulo axioms. *J. Log. Algebraic Methods Program.* 111 (2020), 100513.

[36] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. 2015. Semantics, distributed implementation, and formal analysis of KLAIM models in Maude. *Sci. Comput. Program.* 99 (2015), 24–74.

[37] Steven Eker, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Alberto Verdejo. 2023. The Maude strategy language. *J. Log. Algebraic Methods Program.* 134 (2023), 100887.

[38] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. 2002. The Maude LTL model checker. In *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*, F. Gadducci and U. Montanari (Eds.). ENTCS, Elsevier.

[39] Maribel Fernández and Murdoch J. Gabbay. 2007. Nominal Rewriting. *Inf. Comput.* 205, 6 (June 2007), 917–965.

[40] Maribel Fernández and Murdoch J. Gabbay. 2010. Closed Nominal Rewriting and Efficiently Computable Nominal Algebra Equality. In *LFMTP*.

[41] Maribel Fernández, Murdoch J. Gabbay, and Ian Mackie. 2004. Nominal Rewriting Systems. In *PPDP*. ACM Press, 108–119.

[42] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science (LICS 1999)*. IEEE Computer Society Press, 193–202.

[43] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 193–202.

[44] Murdoch J. Gabbay. 2011. Foundations of nominal techniques: logic and semantics of variables in abstract syntax. *Bulletin of Symbolic Logic* (2011).

[45] Murdoch J. Gabbay and Aad Mathijssen. 2009. Nominal Universal Algebra: Equational Logic with Names and Binding. *J. Log. Comput.* 19, 6 (Dec. 2009), 1455–1508.

[46] Murdoch J. Gabbay and Andrew M. Pitts. 2001. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing* 13, 3–5 (July 2001), 341–363.

[47] Daniel Găină, Dorel Lucanu, Kazuhiro Ogata, and Kokichi Futatsugi. 2014. On Automation of OTS/CafeOBJ Method. In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi (Lecture Notes in Computer Science, Vol. 8373)*. Springer, 578–602.

[48] Lorenzo Gheri and Andrei Popescu. 2020. A Formalized General Theory of Syntax with Bindings: Extended Version. *J. Autom. Reason.* 64, 4 (2020), 641–675.

[49] Jürgen Giesl and Deepak Kapur. 2001. Dependency Pairs for Equational Rewriting. In *RTA 2001 (Lecture Notes in Computer Science, Vol. 2051)*. Springer, 93–108.

[50] Joseph Goguen. 1980. How to Prove Algebraic Inductive Hypotheses without Induction: with Applications to the Correctness of Data Type Representations. In *Proc. Fifth Conference on Automated Deduction*. LNCS, Vol. 87. Springer, 356–373.

[51] Joseph Goguen. 1990. Proving and Rewriting. In *Algebraic and Logic Programming, Second International Conference, Nancy, France, October 1-3, 1990, Proceedings (Lecture Notes in Computer Science, Vol. 463)*. Springer, 1–24.

[52] Joseph Goguen and Rod Burstall. 1992. Institutions: Abstract Model Theory for Specification and Programming. *J. ACM* 39, 1 (1992), 95–146.

[53] Joseph Goguen and José Meseguer. 1992. Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theoretical Computer Science* 105 (1992), 217–273.

[54] J. Goguen, J. W. Thatcher, and E. G. Wagner. 1978. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology*, R. Yeh (Ed.). Prentice-Hall, 80–149.

[55] R. Gutiérrez, J. Meseguer, and S. Skeirik. [n. d.]. The Maude termination assistant. In Pre-Proceedings of WRLA 2018.

[56] John V. Guttag, James J. Horning, Stephen J. Garland, Kevin D. Jones, A. Modet, and Jeannette M. Wing. 1993. *Larch: Languages and Tools for Formal Specification*. Springer. https://doi.org/10.1007/978-1-4612-2704-5

[57] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A framework for defining logics. *JACM* 40, 1 (1993), 143–184.

[58] Joe Hendrix, Deepak Kapur, and José Meseguer. 2010. Coverset Induction with Partiality and Subsorts: A Powerlist Case Study. In *ITP 2010*, Vol. 6172. Springer LNCS, 275–290.

[59] Joseph D. Hendrix. 2008. *Decision Procedures for Equationally Based Reasoning*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign. `http://hdl.handle.net/2142/10967`.

[60] Gérard P. Huet and Jean-Marie Hullot. 1982. Proofs by Induction in Equational Theories with Constructors. *J. Comput. Syst. Sci.* 25, 2 (1982), 239–266. https://doi.org/10.1016/0022-0000(82)90006-X

[61] Jean-Pierre Jouannaud and Hélène Kirchner. 1986. Completion of a Set of Rules Modulo a Set of Equations. *SIAM Journal of Computing* 15 (November 1986), 1155–1194.

[62] Deepak Kapur and David R. Musser. 1987. Proof by Consistency. *Artif. Intell.* 31, 2 (1987), 125–157. https://doi.org/10.1016/0004-3702(87)90017-8

[63] Deepak Kapur and Hantao Zhang. 1989. An Overview of Rewrite Rule Laboratory (RRL). In *Proc. RTA-89 (Lecture Notes in Computer Science, Vol. 355)*. Springer, 559–563.

[64] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. 1993. Combinatory Reduction Systems: Introduction and Survey. *Theor. Comput. Sci.* 121, 1&2 (1993), 279–308.

[65] Jordi Levy and Mateu Villaret. 2010. An Efficient Nominal Unification Algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 6)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 209–226.

[66] Narciso Martí-Oliet and José Meseguer. 1994. General Logics and Logical Frameworks. In *What is a Logical System?*, D. Gabbay (Ed.). Oxford University Press, 355–392.

[67] Narciso Martí-Oliet and José Meseguer. 2002. Rewriting Logic as a Logical and Semantic Framework. In *Handbook of Philosophical Logic, 2nd. Edition*, D. Gabbay and F. Guenthner (Eds.). Kluwer Academic Publishers, 1–87. First published as SRI Tech. Report SRI-CSL-93-05, August 1993.

[68] Richard Mayr and Tobias Nipkow. 1998. Higher-order Rewrite Systems and their Confluence. *Theor. Comp. Sci.* 192 (1998), 3–29.

[69] José Meseguer. 1992. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science* 96, 1 (1992), 73–155.

[70] José Meseguer. 2012. Twenty years of rewriting logic. *J. Algebraic and Logic Programming* 81 (2012), 721–781.

[71] José Meseguer. 2014. Taming distributed system complexity through formal patterns. *Sci. Comput. Program.* 83 (2014), 3–34.

[72] José Meseguer. 2017. Strict coherence of conditional rewriting modulo axioms. *Theor. Comput. Sci.* 672 (2017), 1–35.

[73] José Meseguer. 2020. Generalized rewrite theories, coherence completion, and symbolic methods. *J. Log. Algebraic Methods Program.* 110 (2020).

[74] José Meseguer and Joseph Goguen. 1985. Initiality, Induction and Computability. In *Algebraic Methods in Semantics*, Maurice Nivat and John Reynolds (Eds.). Cambridge University Press, 459–541.

[75] J. Meseguer and G. Roşu. 2007. The Rewriting Logic Semantics Project. *Theoretical Computer Science* 373 (2007), 213–237.

[76] José Meseguer and Grigore Rosu. 2011. The Rewriting Logic Semantics Project: A Progress Report. In *Fundamentals of Computation Theory, FCT 2011 (Lecture Notes in Computer Science, Vol. 6914)*. Springer, 1–37.

[77] J. Meseguer and S. Skeirik. 2020. Inductive Reasoning with Equality Predicates, Contextual Rewriting and Variant-Based Simplification. In *Proc. WRLA 2020 (LNCS, Vol. 12328)*. Springer, 114–135.

[78] José Meseguer and Stephen Skeirik. 2022. On Ground Convergence and Completeness of Conditional Equational Program Hierarchies. In *Rewriting Logic and Its Applications - 14th International Workshop, WRLA@ETAPS 2022, Munich, Germany, April 2-3, 2022 (Lecture Notes in Computer Science, Vol. 13252)*. Springer, 191–211.

[79] Robin Milner. 1999. *Communicating and mobile systems: the π-calculus*. Cambridge University Press.

[80] David R. Musser. 1980. On Proving Inductive Properties of Abstract Data Types. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*. ACM Press, 154–162.

[81] Nipkow, Paulson, and Wenzel. 2001. *Isabelle's Logics: HOL*. Cambridge University Computer Laboratory.

[82] Carlos Olarte, Elaine Pimentel, and Camilo Rocha. 2023. A rewriting logic approach to specification, proof-search, and meta-proofs in sequent systems. *J. Log. Algebraic Methods Program.* 130 (2023), 100827.

[83] G. E. Peterson and M. E. Stickel. 1981. Complete Sets of Reductions for Some Equational Theories. *Journal of the Association Computing Machinery* 28, 2 (1981), 233–264.

[84] Frank Pfenning and Carsten Schürmann. 1999. System description: Twelf — A Meta-logical Framework for Deductive Systems. In *CADE'16 (LNCS, Vol. 1632)*. Springer, 202–206.

[85] Brigitte Pientka. 2010. Beluga: Programming with Dependent Types, Contextual Data, and Contexts. In *FLOPS*. 1–12. https://doi.org/10.1007/978-3-642-12251-4_1

[86] Andrew M. Pitts. 2001. Nominal Logic: A First Order Theory of Names and Binding. In *TACS (LNCS, Vol. 2215)*. Springer, 219–242.

[87] Andrew M. Pitts. 2003. Nominal Logic, A First Order Theory of Names and Binding. *Inf. Comput.* 186, 2 (2003), 165–193.

[88] Andrew M. Pitts. 2013. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, USA.

[89] Andrew M. Pitts and Murdoch J. Gabbay. 2000. A Metalanguage for Programming with Bound Names Modulo Renaming. In *Proceedings of the 5th international conference on the mathematics of program construction (MPC 2000) (LNCS, Vol. 1837)*. Springer, 230–255.

[90] Andrei Popescu. 2024. Nominal Recursors as Epi-Recursors. *Proc. ACM Program. Lang.* 8, POPL, Article 15 (jan 2024), 32 pages. https://doi.org/10.1145/3632857

[91] Albert Rubio. 1994. *Automated Deduction with Constrained Clauses*. Ph. D. Dissertation. Universitat Politècnica de Catalunya.

[92] Albert Rubio. 2002. A Fully Syntactic AC-RPO. *Inf. Comput.* 178, 2 (2002), 515–533.

[93] T. Serbanuta, G. Roşu, and J. Meseguer. 2009. A Rewriting Logic Approach to Operational Semantics. *Inf. Comput.* 207, 2 (2009), 305–340.

[94] Stephen Skeirik, Andrei Stefanescu, and José Meseguer. 2020. A Constructor-Based Reachability Logic for Rewrite Theories. *Fundam. Inform.* 173, 4 (2020), 315–382.

[95] M.-O. Stehr. 2000. CINNI - A Generic Calculus of Explicit Substitutions and its Application to λ-, ς- and π-calculi, K. Futatsugi (Ed.). ENTCS 36, Elsevier. Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications.

[96] M.-O. Stehr. 2000. A Rewriting Semantics for Algebraic Nets. In *Petri Nets for System Engineering – A Guide to Modelling, Verification, and Applications*, C. Girault and R. Valk (Eds.). Springer-Verlag.

[97] M.-O. Stehr and J. Meseguer. 2004. Pure Type Systems in Rewriting Logic: Specifying Typed Higher-Order Languages in a First-Order Logical Framework. In *Essays in Memory of Ole-Johan Dahl*. Springer LNCS Vol. 2635, 334–375.

[98] Takaki Suzuki, Kentaro Kikuchi, Takahito Aoto, and Yoshihito Toyama. 2015. Confluence of Orthogonal Nominal Rewriting Systems Revisited. In *26th International Conference on Rewriting Techniques and Applications (RTA 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 36)*, Maribel Fernández (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 301–317. https://doi.org/10.4230/LIPIcs.RTA.2015.301

[99] Prasanna Thati, Koushik Sen, and Narciso Martí-Oliet. 2002. An Executable Specification of Asynchronous Pi-Calculus Semantics and May Testing in Maude 2.0. In *Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications*, F. Gadducci and U. Montanari (Eds.). ENTCS, Elsevier.

[100] Christian Urban. 2008. Nominal Techniques in Isabelle/HOL. *J. Autom. Reason.* 40, 4 (May 2008), 327–356.

[101] Christian Urban and Cezary Kaliszyk. 2012. General Bindings and Alpha-Equivalence in Nominal Isabelle. *Log. Methods Comput. Sci.* 8, 2 (2012).

[102] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. 2004. Nominal Unification. *Theor. Comp. Sci.* 323, 1–3 (2004), 473–497.

[103] P. Viry. 1996. Input/Output for ELAN. In *Proc. First Intl. Workshop on Rewriting Logic and its Applications (Electronic Notes in Theoretical Computer Science, Vol. 4)*, J. Meseguer (Ed.). Elsevier. http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4.htm.

[104] P. Viry. 2002. Equational rules for rewriting logic. *Theoretical Computer Science* 285 (2002), 487–517. Issue 2.

Closure Conversion, Flat Environments, and the Complexity of Abstract Machines

Beniamino Accattoli
Inria & LIX, École Polytechnique
France
beniamino.accattoli@inria.fr

Cláudio Belo Lourenço
Huawei Central Software Institute
UK
claudio.lourenco@huawei.com

Dan Ghica
Huawei Central Software Institute,
University of Birmingham
UK
dan.ghica@huawei.com

Giulio Guerrieri
University of Sussex
UK
g.guerrieri@sussex.ac.uk

Claudio Sacerdoti Coen
Università di Bologna
Italy
claudio.sacerdoticoen@unibo.it

Abstract

Closure conversion is a program transformation at work in compilers for functional languages to turn inner functions into global ones, by building *closures* pairing the transformed functions with the *environment* of their free variables. Abstract machines rely on similar and yet different concepts of *closures* and *environments*.

We study the relationship between the two approaches. We adopt a simple λ-calculus with tuples as source language and study abstract machines for both the source language and the target of closure conversion. Moreover, we focus on the simple case of flat closures/environments (no sharing of environments). We provide three contributions. Firstly, a new simple proof technique for the correctness of closure conversion, inspired by abstract machines. Secondly, we show how the closure invariants of the target language allow us to design a new way of handling environments in abstract machines, not suffering the shortcomings of other styles.

Thirdly, we study the machines from the point of view of time complexity. We show that closure conversion decreases various dynamic costs while increasing the size of the initial code. Despite these changes, the overall complexity of the machines before and after closure conversion turns out to be the same.

CCS Concepts

• **Theory of computation → Lambda calculus**; **Abstract machines**; **Operational semantics**; **Invariants**; • **Software and its engineering → Compilers**.

Keywords

Lambda calculus, abstract machine, program transformation

ACM Reference Format:

Beniamino Accattoli, Cláudio Belo Lourenço, Dan Ghica, Giulio Guerrieri, and Claudio Sacerdoti Coen. 2025. Closure Conversion, Flat Environments, and the Complexity of Abstract Machines. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

A feature of functional languages, as well as of the λ-calculi on which they are based, is the possibility of defining *inner functions*, that is, functions defined inside the definition of other functions. The tricky point is that inner functions can also use the external variables of their enveloping functions. For instance, the Church numeral $\underline{3} := \lambda x.\lambda y.x(x(xy))$ contains the inner function $\lambda y.x(x(xy))$ that uses the externally defined variable x.

The result of functional programs can be a function. In particular, it can be the instantiation of an inner function. For instance, the result of applying $\underline{3}$ above to a value v is the instantiated function $\lambda y.v(v(vy))$, as one can easily see by doing one β-reduction step.

Closure Conversion. In practice, however, compiled functional programs do not follow β-reduction literally, nor do they produce the code of the instantiated function itself, because the potential duplications of the substitution process would be too costly. The idea is to decompose β-reduction in smaller, micro steps, delaying substitution as much as possible, and computing representations of instantiated functions called *closures*. A closure, roughly, is the pair of the defined inner function (that is, before instantiation) plus the tuple of instantiations for its free variables, called its *environment*.

This is achieved via a program transformation called *closure conversion*, that re-structures the code turning beforehand all inner functions into closures. At compile time, closures pair inner functions with *initial environments* that simply contain the free variables of the functions. Execution shall then dynamically fill up the environments with the actual instantiations.

Compilation and Abstract Machines. Abstract machines are often seen as a technique alternative to compilation and related to the interpretation of programming languages. This is because abstract machines tend to be developed for the source language, before the pipeline of transformations and optimizations of the compiler.

A *first aim* of our paper is to take a step toward closing the gap between compilation and abstract machines, by studying how transformations used by compilers—here closure conversion—induce

invariants exploitable for the design of machines working at further stages of the compilation pipeline, while still being abstract.

Same Terminology, Different Concepts. Similarly to compilation, abstract machines do not follow β-reduction literally, nor do they produce the code of instantiated functions. In particular, some abstract machines use data structures called again *closures* and *environments*. These notions, however, are similar and yet *different*.

In closure conversion, an inner function with free variables is transformed at compile time into a closed function paired with the environment of its free variables. The environment is in general *open* (it is closed by some enveloping converted function, yet locally it is open), and shall be filled/closed only during execution, which provides the instantiations. The aim is to hoist the closed function up to global scope while its environment stays at the call site.

In abstract machines, every piece of code receives an environment, not just functions, and environments and closures have a mutually recursive structure. Additionally, environments are built dynamically, during execution, not in advance, and they are always closed: the pair of a piece of code and its environment is called a *closure* because they read back (or decode) to a closed λ-term. Moreover, there is no hoisting of code.

A *second aim* of this paper is to establish and clarify the relationship between these two approaches. To disambiguate, we keep *closures* for the pairs *(closed function, possibly open environment)* of closure conversion and use *m-closures* for the pairs *(code, closed environment)* used in abstract machines; we further disambiguate calling *bags* the environments of closures, while the environments of m-closures keep their usual names. With respect to this terminology, we study *closure conversion for m-closures*. One of the outcomes of closure conversion turns out to be the elimination of m-closures.

Complexity of Abstract Machines. Finally, a last influence on our study comes from the recent development by Accattoli and coauthors of both a complexity-based theory of abstract machines [2–5, 8, 12, 13, 20] and time and space reasonable cost models for the λ-calculus [6, 7, 9]. Such a line of work has developed fine analyses of tools and techniques for abstract machines, studying how different kinds of environments and forms of sharing impact on the cost of execution of abstract machines. The *third aim* of the paper is to understand how tuples and closure conversions affect environments, forms of sharing, and the cost of execution.

To avoid misunderstandings, our aim is *not* the study of optimized/shared notions of (compiler) closures, their efficient representations, their minimization, or the trade-off between access time and allocation time—in fact, we adopt the basic form of *flat* closure conversion that we apply to *all* functions for *all* their free variables.

Flat Environments. Different data structures for (m-)closures and bags/environments and various closure conversion algorithms can be used. The design space, in particular, is due to chains of nested functions—say, f is nested inside g, in turn, nested inside h—where two consecutive nested functions, f and g, can both use variables, say x, of h. Therefore, one might want the closures for f and g to share the environment entry for x. Perhaps surprisingly, sharing bags between closures can easily break *safety for space* of closure conversion, i.e. might not preserve the space required by the source program, if parts of shared environments survive the lifespan of

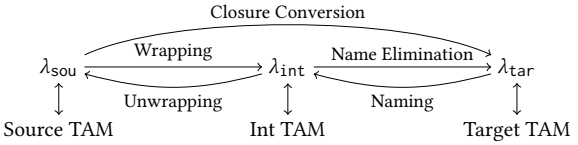


Figure 1: λ-Calculi and abstract machines in the paper.

the associated closures. The simplest approach is using *flat bags* (and flat closure conversion), where no sharing between bags is used; with flat bags, there are two distinct but identical entries for x in the bags of f and g. Flat bags are safe for space [31, 35, 36].

Machine environments can also either be shared or flat (i.e. with no sharing between environments). The distinction, however, is not often made since it does not show up in the abstract specification of the machine, but only when one concretely implements it or studies the complexity of the overhead of the machine (since the two techniques have different costs). For a meaningful comparison with flat closure conversion, for source programs we adopt an abstract machine meant to be implemented using flat environments.

Our aim is to understand how closure conversion compares *asymptotically* with m-closures of abstract machines, in particular with respect to one of the key parameters for time analyses, the size of the initial term. Flat environments are chosen for their simplicity and their safeness for space. More elaborated forms are future work.

Our Setting. We decompose closure conversion in two phases, first going from a source calculus $\lambda_{\mathtt{sou}}$ to an intermediate one $\lambda_{\mathtt{int}}$, and then to a target calculus $\lambda_{\mathtt{tar}}$. We provide each of the three calculi with its own abstract machine. Our setting is summed up in Fig. 1, where *TAM* stands for *tupled abstract machine*.

Our source λ-calculus $\lambda_{\mathtt{sou}}$ is the simplest possible setting accommodating closure conversion, that is, Plotkin's effect-free call-by-value λ-calculus extended with tuples, because tuples are needed to define the conversion. Our setting is untyped, because it is how abstract machines are usually studied, and for the sake of minimality.

The first transformation $\underline{\cdot} : \lambda_{\mathtt{sou}} \to \lambda_{\mathtt{int}}$, dubbed *wrapping*, replaces abstractions $\lambda\widetilde{x}.t$ (where $\widetilde{x}$ is a sequence of variables) with the dedicated construct of *abstract closures* $\underline{\lambda\widetilde{x}.t} := [\![\lambda\widetilde{y}.\lambda\widetilde{x}.\underline{t}\,|\,\vec{\widetilde{y}}]\!]$ where $\widetilde{y}$ is the sequence of free variables of $\lambda\widetilde{x}.t$, and $\vec{\widetilde{y}}$ is the tuple of these same variables, forming the *initial bag* of the closure.

Roughly, the second transformation $\lambda_{\mathtt{int}} \to \lambda_{\mathtt{tar}}$, deemed *name elimination*, turns the abstraction $\lambda\widetilde{y}$ of many variables into the abstraction of a single variable representing the bag/environment. It is similar to a translation from named variables to de Bruijn indices. In fact, name elimination is only outlined in the paper, the details are given in [10], because they are mostly routine.

The target calculus $\lambda_{\mathtt{tar}}$ is not the low-level target language of a compiler. It is indeed still high-level, because we study only flat closure conversion, not the whole compilation pipeline. We do not model the hoisting of closed functions up to global scope; it is an easy aspect of closure conversion and is usually avoided in its study.

We give three contributions, shaping the paper into three parts.

Contribution 1: A Simple Proof of Correctness. The correctness of closure conversion is often showed by endowing both the source and the target calculus of the transformation with big-step operational semantics and establishing a logical relation between the two

[16, 28, 31, 37]. The reason is that adopting a small-step semantics does not seem to work: closure conversion does not commute with meta-level substitution (that is, the substitution of converted terms is not the conversion of the substituted terms), and thus it does not map β-steps from the source to the target calculus.

Bowman and Ahmed [18] are to our knowledge the only ones adopting a small-step semantics. They neatly get around the non-commutation issue by noticing that the substitution of converted terms is *η-equivalent to* the conversion of the substituted terms. Our proof technique looks at the same issue in a different way, inspired by correctness proofs for abstract machines and independently of η-equivalence (which we do not consider for our calculi, for the sake of minimality). It might be of interest for languages where η-equivalence is not sound (e.g. because of observable effects, as in OCaml, or when η-equivalence does not preserve typability, like in languages with mutability and the value restriction), since in these cases Bowman and Ahmed's proof might not scale up.

Contribution 2: New Kinds of Machine Environments. According to Fernández and Siafakas [23], there are two kinds of abstract machines, those using many (shared or flat) *local environments*, which are defined by mutual induction with m-closures, and those using a single (necessarily flat) *global environment* or *heap* and no m-closures. Each kind has pros and cons, there is no absolute better style of machine environments; see also Accattoli and Barras [4].

For our machine for the source calculus $\lambda_{\mathtt{sou}}$, the Source TAM, we adopt flat local environments and m-closures, as to allow us to compare closures (with flat bags) and m-closures. The contribution here is that the invariants enforced by flat closure conversion (more precisely, by wrapping) enable a new management of environments, what we dub *stackable environments* and plug into the Int TAM, our machine for the intermediate calculus $\lambda_{\mathtt{int}}$. Stackable environments have the pros of *both* global and local environments, and *none* of their cons, as explained in Sect. 8. They are called *stackable* because the current one can be put on hold—on the stack—when entering a closed function with its new environment, and re-activated when the evaluation of the function is over. But be careful: their stackability is not necessarily an advantage, it is just the way they work; the advantage is the lack of the cons of local and global environments.

Moving to the target calculus $\lambda_{\mathtt{tar}}$ enables a further tweak of environments, adopted by the Target TAM: environments—which usually are *maps* associating variables to values—become *tuples* of values, with no association to variables. This is enabled by *name elimination*, which turns variables into indices referring to the tuple, in a way reminiscent of de Bruijn indices.

Contribution 3: Time Complexity. We study how tuples and flat closure conversion impact the forms of sharing and the time complexity of the machines. Our analyses produce four insights:

(1) *Tuples raise the overhead*: we give a theoretical analysis of the cost of adding tuples to the pure λ-calculus, which, to our knowledge, does not appear anywhere in the literature. We show why tuples require their own form of sharing and that the *creation of tuples* at runtime is *unavoidable*. This is done by adapting *size exploding families* from the study of reasonable time cost models for the λ-calculus [1]. Moreover, tuples raise the dependency on the size $|t|$ of the initial code t of the overhead of the machine. Namely, let the *height* $\mathsf{hg}(t)$ be the maximum number of bound variables of t in the scope of which a sub-term of t is contained: the dependency for flat environments raises from $O(\mathsf{hg}(t))$ to $O(|t| \cdot \mathsf{hg}(t))$.

(2) *Name elimination brings a logarithmic speed-up*: with variable names, flat environments have at best $O(\log(\mathsf{hg}(t)))$ access time, while de Bruijn indices (or our name elimination) enable $O(1)$ access time—this is true both before and after closure conversion. Before conversion, however, the improvement does not lower the overall asymptotic overhead of the machine with respect to the size of the initial term, which is dominated by the other flat environment operations. After closure conversion, instead, it *does* lower the overall dependency of the machine from $O(|t^{\mathsf{cc}}| \cdot \mathsf{hg}(t^{\mathsf{cc}}))$ to $O(|t^{\mathsf{cc}}|)$, where t^{cc} is t after closure conversion.

(3) *Amortized constant cost of transitions*: in any abstract machine, independently of their implementation, the number of transitions of an execution and the cost of some single transitions depend on the size of the initial term. This is related to the higher-order nature of λ-calculi. Closure conversion impacts on the cost of single transitions (but not on their number): their *amortized* cost becomes constant. The insight is that the non-constant cost of transitions in ordinary abstract machines is related to inner functions.

(4) *Dynamically faster, statically bigger, overall the same*: the previous two points show that closure conversion decreases the dependency of machines on the size of the initial term during execution. The dynamic improvement however is counter-balanced by the fact that $|t^{\mathsf{cc}}|$ is possibly *bigger* than $|t|$, namely $|t^{\mathsf{cc}}| \in O(|t| \cdot \mathsf{hg}(t))$. Therefore, the overall complexity is $O(|t| \cdot \mathsf{hg}(t))$ also after closure conversion.

OCaml Code and Proofs. As additional material on GitHub [33], we provide an OCaml implementation of the Target TAM, the machine for closure converted terms, described in [10, Appendix K].

All proofs are in [10], the long version of this paper.

Terms	t, u, s, r	$::=$	$x \mid \lambda x.t \mid tu$	$(\lambda x.t)v \mapsto_{\beta_v} t\{x \leftarrow v\}$
Values	v, v'	$::=$	$\lambda x.t$	$\dfrac{t \mapsto_{\beta_v} u}{C\langle t\rangle \to_{\beta_v} C\langle u\rangle}$
Ev. Ctxs	C, C'	$::=$	$\langle\cdot\rangle \mid tC \mid Cv$	

Figure 2: The untyped pure call-by-value calculus λ_{cbv}

2 Preliminaries: λ_{cbv}, a Call-by-Value λ-Calculus

In Fig. 2 we present λ_{cbv}, a variant of Plotkin's call-by-value λ-calculus [32] with its β-reduction by value $\to_{\beta_v}$, adopting two specific choices. Firstly, the only values are λ-abstractions. Excluding variables from values differs from [32] but is common in the machine-oriented literature. It does not change the result of evaluation while inducing a faster substitution process, see [14].

Secondly, we adopt a small-step operational semantics, defined via evaluation contexts. Evaluation contexts C are special terms with exactly one occurrence of the *hole* constant $\langle\cdot\rangle$. We write $C\langle t\rangle$ for the term obtained from the evaluation context C by replacing its hole with the term t (possibly capturing some free variables of t).

The small-step rule of β_v-reduction is *weak*, that is, it does not evaluate abstraction bodies (indeed, the production $\lambda x.C$ is *absent*

$$\begin{array}{rrcl} \textsc{Terms} & t,u,s & ::= & x \mid tu \mid \pi_i t \mid \overbrace{\lambda\widetilde{x}.t}^{\lambda\widetilde{x}.t} \mid \overbrace{\lambda x_1,..,x_n.t}^{\lambda\widetilde{x}.t}\ n\geq 0 \mid \overbrace{(\!|t_1,..,t_n|\!)}^{\overrightarrow{t}}\ n\geq 0 \\ \textsc{Values} & v,v' & ::= & \lambda\widetilde{x}.t \mid \overrightarrow{v} \\ \textsc{Ev. Ctxs} & C,C' & ::= & \langle\cdot\rangle \mid tC \mid Cv \mid \pi_i C \mid \underbrace{(\!|t_1,..,t_k,C,v_1,..,v_h|\!)}_{(\!|\widetilde{t},C,\widetilde{v}|\!)}\ k,h\geq 0 \end{array}$$

$$\begin{array}{l|l} (\lambda\widetilde{x}.t)\,\overrightarrow{v} \mapsto_{\beta_v} t\{\widetilde{x}\leftarrow\overrightarrow{v}\} \text{ if } \|\widetilde{x}\| = \|\overrightarrow{v}\| & \dfrac{t\mapsto_a u}{C\langle t\rangle \to_a C\langle u\rangle}\ a\in\{\beta_v,\pi\} \\ \pi_i\,\overrightarrow{v} \mapsto_{\pi} v_i \quad \text{if } 1\leq i\leq\|\overrightarrow{v}\| & \to_{\mathsf{sou}} := \to_{\beta_v}\cup\to_{\pi} \end{array}$$

Figure 3: The source calculus λ_{sou} extending λ_{cbv} with tuples.

in the definition of evaluation context C in Fig. 2), as it is common in functional programming languages, and *deterministic*, namely proceeding from right to left (as forced by the production Cv)[1].

We identify terms up to α-renaming; $t\{x\leftarrow u\}$ stands for metalevel capture-avoiding substitution of u for the free occurrences of x in t.

The lemma below rests on the closed hypothesis and will be used as a design check for next sections' calculi. It is an untyped instantiation of Wright and Felleisen's uniform evaluation property [41]. The term "harmony" is borrowed from [11].

Lemma 2.1 (λ_{cbv} harmony). *If $t\in\lambda_{\mathsf{cbv}}$ is closed, then either t is a value or $t\to_{\beta_v} u$ for some closed $u\in\lambda_{\mathsf{cbv}}$.*

Notations. We set some notations for both calculi and machines. Let $\to$ be a reduction relation. An evaluation sequence $e: t\to^* u$ is a possibly empty sequence of $\to$-steps the length of which is noted $|e|$. If a and b are sub-reduction (i.e., $\to_a\subseteq\to$ and $\to_b\subseteq\to$) then $\to_{a,b}:=\to_a\cup\to_b$ and $|e|_a$ is the number of a steps in e.

3 Part 1: The Source Calculus λ_{sou}

In this section, we extend λ_{cbv} with tuples, which are needed to define closure conversion, obtaining λ_{sou}, our source calculus.

Terms. The *source calculus* λ_{sou} defined in Fig. 3 adopts n-ary *tuples* $\overrightarrow{t} = (\!|t_1,..,t_n|\!)$, together with projections π_i on the i^{th} element[2]. Abstractions are now on *sequences* of variables $\widetilde{x} = x_1,..,x_n$. With a slight abuse, we also compact $(\!|t_1,..,t_n|\!)$ into $(\!|\widetilde{t}|\!)$, and write $(\!|\widetilde{t},u,\widetilde{s}|\!)$ (note that replacing sequences with tuples changes the meaning: $(\!|\widetilde{t},u,\widetilde{s}|\!)$ and $(\!|\overrightarrow{t},u,\overrightarrow{s}|\!)$ are different terms, and we shall need both notations). Both tuples and sequences of variables can be empty, that is, $\lambda.t$ and $(\!|\,|\!)$ are terms of λ_{sou}. *Values* now are abstractions *and* tuples *of values*—tuples of arbitrary terms are not values in general.

Notations and Conventions about Tuples and Sequences. We assume that in every sequence $\widetilde{x}$ all elements are distinct and, for brevity, we abuse notations and consider sequences of variables also as the sets of their elements, writing $x_i\in\widetilde{x}$, or $\mathtt{fv}(t)=\widetilde{x}$, or $\widetilde{x}\cup\widetilde{y}$. We set $\|\overrightarrow{t}\| := n$ if $\overrightarrow{t} = (\!|t_1,..,t_n|\!)$ and call it the *length* of the tuple $\overrightarrow{t}$ (so, $\|(\!|\,|\!)\| = 0$), and similarly for $\|\widetilde{x}\|$. Moreover, if $\widetilde{x} = x_1,..,x_n$ and $\overrightarrow{v} = (\!|v_1,..,v_n|\!)$ we then set $t\{\widetilde{x}\leftarrow\overrightarrow{v}\} := t\{x_1\leftarrow v_1,..,x_n\leftarrow v_n\}$ (the *simultaneous* substitution). We write $\widetilde{x}\#\widetilde{y}$ when $\widetilde{x}$ and $\widetilde{y}$ have no element in common, and $\mathtt{fv}(t)\subseteq\widetilde{x}\#\widetilde{y}$ when moreover $\mathtt{fv}(t)\subseteq(\widetilde{x}\cup\widetilde{y})$.

Small-Step Operational Semantics. The β_v-rule can fire only if the argument is a tuple of values of the right length, and similarly for the π-rule. For instance, $(\lambda x.x)(\lambda y.yy)\not\to_{\beta_v}\lambda y.yy$, because one needs a unary tuple around the argument, that is, $(\lambda x.x)(\!|\lambda y.yy|\!)\to_{\beta_v}\lambda y.yy$. Note that evaluation contexts now enter projections and tuples, proceeding right-to-left. As it shall be the case for all calculi in this paper, the operational semantics $\to_{\mathsf{sou}}$ of λ_{sou} is *deterministic*: if $t\to_{\mathsf{sou}} u$ and $t\to_{\mathsf{sou}} s$ then $u=s$ and t is not a value. The proof is a routine induction.

[1]The right-to-left order (adopted also in [24, 26]) induces a more natural presentation of the machines, but all our results could be restated using the left-to-right order.

[2]We do not define projections as tuple-unpacking abstractions because it would turn π-steps into β-steps and so blur the cost analysis (that counts β but not π-steps).

$$\begin{array}{rrcl} \textsc{Bags} & b,b' & ::= & \overrightarrow{x} \mid \overrightarrow{v} \\ \textsc{Terms} & t,u,s,r & ::= & x \mid tu \mid \pi_i t \mid \overrightarrow{t} \mid [\![\widetilde{y};\widetilde{x}.t|b]\!] \\ \textsc{Values} & v,v' & ::= & [\![\widetilde{y};\widetilde{x}.t|b]\!] \mid \overrightarrow{v} \\ \textsc{Eval ctxs} & C,C' & ::= & \langle\cdot\rangle \mid tC \mid Cv \mid \pi_i C \mid (\!|\widetilde{t},C,\widetilde{v}|\!) \end{array}$$

$$\begin{array}{rl|l} [\![\widetilde{y};\widetilde{x}.t|\overrightarrow{v_1}]\!]\,\overrightarrow{v_2} & \mapsto_{\mathtt{i}\beta_v} t\{\widetilde{y};\widetilde{x}\leftarrow\overrightarrow{v_1};\overrightarrow{v_2}\} & \text{if } \|\widetilde{y}\| = \|\overrightarrow{v_1}\| \text{ and } \|\widetilde{x}\| = \|\overrightarrow{v_2}\| \\ \pi_i\,\overrightarrow{v} & \mapsto_{\mathtt{i}\pi} v_i & \text{if } 1\leq i\leq\|\overrightarrow{v}\| \end{array}$$

$$\frac{t\mapsto_a u}{C\langle t\rangle\to_a C\langle u\rangle}\ a\in\{\mathtt{i}\beta_v,\mathtt{i}\pi\} \qquad \to_{\mathtt{int}} := \to_{\mathtt{i}\beta_v}\cup\to_{\mathtt{i}\pi}$$

Figure 4: The intermediate calculus $\lambda_{\mathtt{int}}$.

Clashes. In an untyped setting, there might be terms with *clashes*, that is, irreducible badly formed configurations such as $\pi_i(\lambda x.t)$. To exclude clashes without having to have types, we adopt a notion of clash-freeness, which would be ensured by any type system.

Definition 3.1 (Clashes, clash-free terms). A term t is a *clash* if it has shape $C\langle u\rangle$ where u has one of the following forms:

- *Clashing projection*: $u=\pi_i v$ and if $v=\overrightarrow{v}$ then $\|\overrightarrow{v}\|<i$;
- *Clashing abstraction*: $u=(\lambda\widetilde{x}.s)v$ and if $v=\overrightarrow{v}$ then $\|\widetilde{x}\|\neq\|\overrightarrow{v}\|$;
- *Clashing tuple*: $u=\overrightarrow{r}s$.

A term t is *clash-free* when, co-inductively, t is not a clash and if $t\to_{\mathsf{sou}} u$ then u is clash-free.

Note that clashes are normal forms. All the calculi and machines of the paper shall come with their notion of clash and clash-freeness, which shall be taken into account in statements and proofs but the definitions of which shall be omitted (they are in [10]).

Lemma 3.2 (λ_{sou} harmony). *If $t\in\lambda_{\mathsf{sou}}$ is closed and clash-free, then either t is a value or $t\to_{\mathsf{sou}} u$ for some closed and clash-free u.*

4 Part 1: the Intermediate Calculus $\lambda_{\mathtt{int}}$ and the Wrapping Transformation

In this section, we define the intermediate calculus $\lambda_{\mathtt{int}}$ and the wrapping translation from λ_{sou} to $\lambda_{\mathtt{int}}$. We discuss why the natural first attempt to show the correctness of the translation does not work, and solve the issue via a reverse translation from $\lambda_{\mathtt{int}}$ to λ_{sou}.

Terms of $\lambda_{\mathtt{int}}$. In $\lambda_{\mathtt{int}}$, defined in Fig. 4, abstractions $\lambda\widetilde{x}.t$ are replaced by *closures* $[\![\lambda\widetilde{y}.\lambda\widetilde{x}.t|b]\!]$, which are compactly noted $[\![\widetilde{y};\widetilde{x}.t|b]\!]$. The *bag* b of a closure can be of two forms $\overrightarrow{z}$ and $\overrightarrow{v}$, giving *variable closures* $[\![\widetilde{y};\widetilde{x}.t|\overrightarrow{z}]\!]$ and *evaluated closures* $[\![\widetilde{y};\widetilde{x}.t|\overrightarrow{v}]\!]$, which are both values. In a closure $[\![\widetilde{y};\widetilde{x}.t|b]\!]$, $\widetilde{y}$ and $\widetilde{x}$ verify $\widetilde{y}\#\widetilde{x}$ (i.e. no elements in common), and scope over the *body* t of the closure; $\widetilde{y}$ and $\widetilde{x}$ do not scope over b. The idea is that $\mathtt{fv}(\lambda\widetilde{x}.t)\subseteq\widetilde{y}$, so that $\lambda\widetilde{y}.\lambda\widetilde{x}.t$ is closed. The elements of b are meant to replace the variables $\widetilde{y}$ in t.

The rationale behind $\lambda_{\mathtt{int}}$ is understood by looking at the translation from λ_{sou} to $\lambda_{\mathtt{int}}$ in Fig. 5. Basically, every abstraction is closed and paired with the bag of its free variables. Evaluated closures $[\![\widetilde{y};\widetilde{x}.t|\overrightarrow{v}]\!]$ are not in the image of the translation, unless

Wrapping translation $\lambda_{\mathtt{sou}} \to \lambda_{\mathtt{int}}$

$$\begin{array}{rcll} \underline{x} & := & x & \underline{tu} := \underline{t}\,\underline{u} \\ \underline{\lambda\widetilde{x}.t} & := & [\![\widetilde{y}; \widetilde{x}.\underline{t} | \overrightarrow{y}]\!] & \text{if } \mathtt{fv}(\lambda\widetilde{x}.t) = \widetilde{y} \\ \underline{(\!|t_1,..,t_n|\!)} & := & (\!|\underline{t_1},..,\underline{t_n}|\!) & \underline{\pi_i t} := \pi_i \underline{t} \end{array}$$

Unwrapping (reverse) translation $\lambda_{\mathtt{int}} \to \lambda_{\mathtt{sou}}$

$$\begin{array}{rcll} \lceil x \rceil & := & x & \lceil tu \rceil := \lceil t \rceil \lceil u \rceil \\ \lceil [\![\widetilde{y}; \widetilde{x}.t | \overrightarrow{y}]\!] \rceil & := & \lambda\widetilde{x}.\lceil t \rceil & \lceil [\![\widetilde{y}; \widetilde{x}.t | \overrightarrow{v}]\!] \rceil := \lambda\widetilde{x}.\lceil t \rceil \{\widetilde{y} \leftarrow \overrightarrow{\lceil v \rceil}\} \\ \lceil (\!|t_1,..,t_n|\!) \rceil & := & (\!|\lceil t_1 \rceil,..,\lceil t_n \rceil|\!) & \lceil \pi_i t \rceil := \pi_i \lceil t \rceil \end{array}$$

Figure 5: The translations $\underline{\ }: \lambda_{\mathtt{sou}} \to \lambda_{\mathtt{int}}$ (wrapping) and $\lceil\cdot\rceil: \lambda_{\mathtt{int}} \to \lambda_{\mathtt{sou}}$ (unwrapping).

$\mathtt{fv}(\lambda\widetilde{x}.t) = \emptyset$, which gives the closure $[\![;\widetilde{x}.t|(\!|\,|\!)]\!]$ that is both a variable and an evaluated closure. Evaluated closures are generated by the reduction rules, discussed after defining well-formedness.

Definition 4.1. A closure $[\![\widetilde{y}; \widetilde{x}.t|b]\!]$ is *well-formed* if $\mathtt{fv}(t) \subseteq \widetilde{y}\#\widetilde{x}$, $\|b\| = \|\widetilde{y}\|$ and if b is a variable bag then $b = (\!|\widetilde{y}|\!)$. Terms $t \in \lambda_{\mathtt{int}}$ and evaluation contexts $C \in \lambda_{\mathtt{int}}$ are *well-formed* if all their closures are well-formed, and *prime* if moreover they are variable closures.

Operational Semantics. The intermediate variant $\to_{\mathtt{i}\beta_v}$ of the β_v-rule involves a well-formed evaluated closure $[\![\widetilde{y}; \widetilde{x}.t|\overrightarrow{v_1}]\!]$ and an argument $\overrightarrow{v_2}$ of the same length of $\widetilde{x}$, and amounts to substitute the bag $\overrightarrow{v_1}$ on $\widetilde{y}$ and the argument $\overrightarrow{v_2}$ on $\widetilde{x}$. Substitution $t\{\widetilde{y}; \widetilde{x} \leftarrow \overrightarrow{v_1}; \overrightarrow{v_2}\}$ is as defined expected (see [10, Appendix B]) by performing $\{\widetilde{y} \leftarrow \overrightarrow{v_1}\}$ and $\{\widetilde{x} \leftarrow \overrightarrow{v_2}\}$ *simultaneously* and requires that $\mathtt{fv}(t) \subseteq \widetilde{y}\#\widetilde{x}$. Well-formed terms are stable by substitution, when defined, and the reduct of $\to_{\mathtt{i}\beta_v}$ is a (well-formed) term of $\lambda_{\mathtt{int}}$ because (by well-formedness) closures close their bodies, so the closing substitution generated by the step turns variable bags into value bags. Tuple projection is as in $\lambda_{\mathtt{sou}}$. Note that evaluation contexts do not enter closures. See [10, Appendix B] for the definition of clash(-freeness) for $\lambda_{\mathtt{int}}$. A term $t \in \lambda_{\mathtt{int}}$ is *good* if it is well-formed and clash-free.

The intermediate calculus $\lambda_{\mathtt{int}}$ is deterministic (if $t \to_{\mathtt{int}} u$ and $t \to_{\mathtt{int}} s$ then $u = s$ and t is not a value) and harmonic.

Lemma 4.2 (Harmony of $\lambda_{\mathtt{int}}$). *Let $t \in \lambda_{\mathtt{int}}$ be closed and good. Then either t is a value or $t \to_{\mathtt{int}} u$ for some closed good $u \in \lambda_{\mathtt{int}}$.*

Translation From Source to Intermediate, or Wrapping. The wrapping translation from $\lambda_{\mathtt{sou}}$ to $\lambda_{\mathtt{int}}$ takes a (possibly open) term $t \in \lambda_{\mathtt{sou}}$ and returns a term $\underline{t} \in \lambda_{\mathtt{int}}$, and it is defined in Fig. 5. As already mentioned, it turns abstractions into closures by closing them and pairing them with the bag of their free variables. It is extended to evaluation contexts C as expected, setting $\underline{\langle\cdot\rangle} := \langle\cdot\rangle$.

Lemma 4.3 (Properties of the translation $\lambda_{\mathtt{sou}} \to \lambda_{\mathtt{int}}$).

(1) Values*: if $v \in \lambda_{\mathtt{sou}}$ then $\underline{v}$ is a value of $\lambda_{\mathtt{int}}$.*
(2) Terms*: if $t \in \lambda_{\mathtt{sou}}$ then $\underline{t} \in \lambda_{\mathtt{int}}$ is well-formed and prime.*
(3) Contexts*: if $C \in \lambda_{\mathtt{sou}}$ then $\underline{C}$ is an evaluation context of $\lambda_{\mathtt{int}}$.*

Problem: Wrapping and Substitution Do not Commute. As source values and contexts are translated to their intermediate analogous, one may think that the translation preserves reduction steps: if $t \to_{\mathtt{sou}} u$ then $\underline{t} \to_{\mathtt{int}} \underline{u}$. But this is *false*, because the translation does not commute with substitution, as also discussed in [18, 38]. Indeed, in general $\underline{t}\{x \leftarrow \underline{v}\} \neq \underline{t\{x \leftarrow v\}}$: take $t := \lambda y.yx$ and a closed value v, we get the terms below, where $[\![x; y.yx|(\!|\underline{v}|\!)]\!] \neq [\![; y.y\underline{v}|(\!|\,|\!)]\!]$.

$$\begin{array}{rcl} \underline{t}\{x \leftarrow \underline{v}\} & = & [\![x; y.yx|(\!|x|\!)]\!]\{x \leftarrow \underline{v}\} = [\![x; y.yx|(\!|\underline{v}|\!)]\!] \\ \underline{t\{x \leftarrow v\}} & = & \underline{\lambda y.yv} = [\![; y.y\underline{v}|(\!|\,|\!)]\!] \end{array}$$

The point being that $t\{x \leftarrow v\}$ is *closed* and so its wrapping is different from first wrapping t, which is instead *open*, and then closing the wrapped term $\underline{t}$ using $\{x \leftarrow \underline{v}\}$. Not only $\underline{t}\{x \leftarrow \underline{v}\} \neq \underline{t\{x \leftarrow v\}}$, they are not even related by $\to_{\mathtt{int}}$, or by the equational theory generated by $\to_{\mathtt{int}}$ (but they can be shown to be contextually equivalent).

The problem is that the translation targets prime terms (Lemma 4.3.2) but $\to_{\mathtt{int}}$ creates evaluated closures (which are not in the image of translation), i.e., *the reduct of a prime term (of $\lambda_{\mathtt{int}}$) may not be prime.* Consider again $t := \lambda y.yx$: then $(\lambda x.t)(\!|v|\!) \to_{\mathtt{sou}} t\{x \leftarrow v\} = \lambda y.yv$ and $\underline{(\lambda x.t)(\!|v|\!)} = [\![; x.[\![x; y.yx|(\!|x|\!)]\!]|(\!|\,|\!)]\!](\!|\underline{v}|\!)$ is prime, but $\underline{(\lambda x.t)(\!|v|\!)}$ $\to_{\mathtt{int}}$-reduces to the non-prime $[\![x; y.yx|(\!|\underline{v}|\!)]\!] = \underline{t}\{x \leftarrow \underline{v}\} \neq \underline{t\{x \leftarrow v\}}$.

Reverse Translation. The literature usually overcomes this problem by switching to a different approach, adopting a big-step semantics and a logical relation proof technique. One of our contributions is to show a direct solution, as done also by Bowman and Ahmed [18], but in a different way. The idea comes from the correctness of abstract machines, which is proved by projecting the machine on the calculus, rather than the calculus on the machine. Therefore, we define a reverse *unwrapping* translation from $\lambda_{\mathtt{int}}$ to $\lambda_{\mathtt{sou}}$ and show that it smoothly preserves reduction steps.

The *unwrapping translation* $\lceil t \rceil$ of a well formed term $t \in \lambda_{\mathtt{int}}$ to $\lambda_{\mathtt{sou}}$ is defined in Fig. 5. It amounts to substitute the bag b for the sequence $\widetilde{y}$ of variables of a closure $[\![\widetilde{y}; \widetilde{x}.t|b]\!]$. In contrast with wrapping, now unwrapping and substitution commute.

Proposition 4.4 (Commutation of substitution and the reverse translation). *If $t, \overrightarrow{v_1}, \overrightarrow{v_2} \in \lambda_{\mathtt{int}}$ and $\mathtt{fv}(t) \subseteq \widetilde{x}\#\widetilde{y}$, then $\lceil t\{\widetilde{x}; \widetilde{y} \leftarrow \overrightarrow{v_1}; \overrightarrow{v_2}\} \rceil = \lceil t \rceil \{\widetilde{x} \leftarrow \overrightarrow{\lceil v_1 \rceil}\}\{\widetilde{y} \leftarrow \overrightarrow{\lceil v_2 \rceil}\}$.*

Strong Bisimulation. From the commutation property (Prop. 4.4), it easily follows that the reverse translation projects and reflects rewrite steps and—if terms are closed—also normal forms. As a consequence, it is a termination-preserving strong bisimulation, possibly the strongest form of *correctness* for a program transformation.

Theorem 4.5 (Source-intermediate termination-preserving strong bisimulation). *Let $t \in \lambda_{\mathtt{int}}$ be closed and well-formed.*

(1) Projection*: for $a \in \{\beta_v, \pi\}$, if $t \to_{\mathtt{i}a} u$ then $\lceil t \rceil \to_a \lceil u \rceil$.*
(2) Halt*: t is $\to_{\mathtt{int}}$-normal if and only if $\lceil t \rceil$ is $\to_{\mathtt{sou}}$-normal.*
(3) Reflection*: for $a \in \{\beta_v, \pi\}$, if $\lceil t \rceil \to_a u$ then there exists $s \in \lambda_{\mathtt{int}}$ such that $t \to_{\mathtt{i}a} s$ and $\lceil s \rceil = u$.*
(4) Inverse*: if u is a source term then $\lceil \underline{u} \rceil = u$.*

Projection can be extended to sequences of steps. The reflection and inverse properties ensure that reflections of consecutive steps from the source can be composed, bypassing the problem with the wrapping translation, see the corollary below. Its proof only depends on the abstract properties of our notion of bisimulation, thus similar corollaries hold for all bisimulations in this paper.

Corollary 4.6 (Preservation of reduction steps). *If $t \to^k_{\mathtt{sou}} u$ then there exists $s \in \lambda_{\mathtt{int}}$ such that $\underline{t} \to^k_{\mathtt{int}} s$ and $\lceil s \rceil = u$.*

Proof. By induction on k. If $k = 0$ then $t = u$; taking $s := \underline{t}$, the inverse property gives $\lceil s \rceil = \lceil \underline{t} \rceil = t = u$. For $k > 0$, one has $t \to^{k-1}_{\mathtt{sou}} u' \to_{\mathtt{sou}} u$. By *i.h.*, $\underline{t} \to^{k-1}_{\mathtt{int}} s'$ with $\lceil s' \rceil = u'$, so $\lceil s' \rceil \to_{\mathtt{sou}} u$; by reflection $s' \to_{\mathtt{int}} s$ for some s such that $\lceil s \rceil = u$. □

$$
\begin{array}{rrcl}
\textsc{Projected vars} & p, p' & ::= & \pi_i \mathtt{w} \mid \pi_i \mathtt{s} \\
\textsc{Terms} & t, u, s, r & ::= & p \mid \pi_i t \mid \overrightarrow{t} \mid t\, u \mid [\![t|b]\!]_{n,m} \\
\textsc{Bags} & b, b' & ::= & \vec{p} \mid \vec{v} \\
\textsc{Values} & v, v' & ::= & [\![t|b]\!]_{n,m} \mid \overrightarrow{v} \\
\textsc{Eval Contexts} & C, C' & ::= & \langle\cdot\rangle \mid t\, C \mid C\, \overrightarrow{v} \mid \pi_i C \mid (\!|\widetilde{t}, C, \widetilde{v}|\!)
\end{array}
$$

$$\textsc{Name elimination translation } \lambda_{\mathtt{int}} \to \lambda_{\mathtt{tar}}$$

$$
\begin{array}{rclcrcl}
y_i^{\widetilde{y},\widetilde{x}} & := & \pi_i \mathtt{w} & \qquad & (\!|t_1,..,t_n|\!)^{\widetilde{y},\widetilde{x}} & := & (\!|\, t_1^{\widetilde{y},\widetilde{x}}, .., \, t_n^{\widetilde{y},\widetilde{x}}|\!) \\
(\pi_i t)^{\widetilde{y},\widetilde{x}} & := & \pi_i\, t^{\widetilde{y},\widetilde{x}} & & [\![\widetilde{z};\widetilde{w}.t|b]\!]^{\widetilde{y},\widetilde{x}} & := & [\![\, t^{\widetilde{z},\widetilde{w}} |\, b^{\widetilde{y},\widetilde{x}}]\!] \\
x_i^{\widetilde{y},\widetilde{x}} & := & \pi_i \mathtt{s} & & t\,u^{\widetilde{y},\widetilde{x}} & := & t^{\widetilde{y},\widetilde{x}}\, u^{\widetilde{y},\widetilde{x}}
\end{array}
$$

$$\textsc{Naming (reverse) translation } \lambda_{\mathtt{tar}} \to \lambda_{\mathtt{int}}$$

$$
\begin{array}{rclcrcl}
(\pi_i \mathtt{w})_{\widetilde{y},\widetilde{x}} & := & y_i & \qquad & (t\,u)_{\widetilde{y},\widetilde{x}} & := & t_{\widetilde{y},\widetilde{x}}\, u_{\widetilde{y},\widetilde{x}} \\
(\pi_i \mathtt{s})_{\widetilde{y},\widetilde{x}} & := & x_i & & (\!|t_1,..,t_n|\!)_{\widetilde{y},\widetilde{x}} & := & (\!|\, t_{1\,\widetilde{y},\widetilde{x}}, .., \, t_{n\,\widetilde{y},\widetilde{x}}|\!) \\
((\pi_i t))_{\widetilde{y},\widetilde{x}} & := & \pi_i\, t_{\widetilde{y},\widetilde{x}} & & [\![t|b]\!]_{n,m\,\widetilde{y},\widetilde{x}} & := & [\![\widetilde{z};\widetilde{w}.\, t_{\widetilde{z},\widetilde{w}} |\, b_{\widetilde{y},\widetilde{x}}]\!]
\end{array}
$$

with $\widetilde{z}\#\widetilde{w}$ fresh, $\|\widetilde{z}\| = n$, $\|\widetilde{w}\| = m$

Figure 6: The target calculus $\lambda_{\mathtt{tar}}$, and the translations $\lambda_{\mathtt{int}} \to \lambda_{\mathtt{tar}}$ (name elimination) and $\lambda_{\mathtt{tar}} \to \lambda_{\mathtt{int}}$ (naming).

About the proof of the various points of Theorem 4.5, the closed hypothesis is used for the left-to-right direction of the halt property, which in turn is used to prove the reflection property. Projection is used instead in the right-to-left direction of the halt property. The inverse property is straightforward.

5 Part 1: Outline of the Target Calculus $\lambda_{\mathtt{tar}}$ and of Name Elimination

In this section, we quickly outline the *target calculus* $\lambda_{\mathtt{tar}}$ and the translation from $\lambda_{\mathtt{int}}$ to $\lambda_{\mathtt{tar}}$, dubbed *name elimination*, which, when composed with the wrapping translation of Section 4, provides the *closure conversion* transformation. The unsurprising details (definitions, statements, and proofs) are given in [10, Appendix C]. The ideas behind $\lambda_{\mathtt{tar}}$ and name elimination defined in Fig. 6 are:

- *Variable binders*: replacing the sequences $\widetilde{y}$ and $\widetilde{x}$ of abstracted variables in closures $[\![\widetilde{y};\widetilde{x}.t|b]\!]$ with the special variables $\mathtt{w}$ and $\mathtt{s}$ (short for *wrapped* and *source*), standing for the tuples $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ meant to be substituted on $\widetilde{y}$ and $\widetilde{x}$;
- *Variable occurrences*: replacing every occurrence of a variable y_i or x_j in t with the *projected variables* $\pi_i\mathtt{w}$ or $\pi_j\mathtt{s}$, which are two special compound terms as $\mathtt{w}$ and $\mathtt{s}$ cannot appear without being paired with a projection.

We use p (for *projected variable*) to refer to either $\pi_i\mathtt{w}$ or $\pi_j\mathtt{s}$. The transformation is similar to switching to de Bruijn indices, except that—because of the already wrapped setting—the index simply refers to the composite binder of the closure, rather to the nested binders above the variable occurrence. It also slightly differs from standard closure conversion: the standard transformation would eliminate the $\widetilde{y}$ names but usually not the $\widetilde{x}$ ones. We eliminate both, as the total elimination of names shall induce a logarithmic speed-up for the abstract machine associated with $\lambda_{\mathtt{tar}}$, in Sect. 12.

The terminology *name elimination* refers to the fact that, after this transformation, a term uses only two variables, $\mathtt{w}$ and $\mathtt{s}$. As all closures would then have shape $[\![\mathtt{w};\mathtt{s}.t|b]\!]$, we simplify the notation and just write $[\![t|b]\!]$, with the implicit assumption that every closure now binds $\mathtt{w}$ and $\mathtt{s}$ in t. To define a reverse translation from $\lambda_{\mathtt{tar}}$ to $\lambda_{\mathtt{int}}$ dubbed *naming* in Fig. 6, closures $[\![t|b]\!]_{n,m}$ are annotated with two natural numbers $n, m \in \mathbb{N}$, which record the length of the replaced sequences of variables $\widetilde{y}$ and $\widetilde{x}$. To simplify the notation, however, we shall omit these annotations when not relevant. The *body* of a closure $[\![t|b]\!]$ is t, and—as in $\lambda_{\mathtt{int}}$—closures are either variable closures, if $b = \widetilde{p}$, or evaluated closures, if $b = \overrightarrow{v}$. A term $t \in \lambda_{\mathtt{tar}}$ is *closed* if $\pi_i\mathtt{w}$ and $\pi_i\mathtt{s}$ do not occur out of closure bodies.

Name elimination is parametric in two lists $\widetilde{y}$ and $\widetilde{x}$ of abstracted variables, which intuitively are those of the closest enclosing closure being translated. In particular the two parametric lists change when the translation crosses the boundary of a closure in the $[\![\widetilde{z};\widetilde{w}.t|b]\!]^{\widetilde{y},\widetilde{x}}$ clause in Fig. 6. On closed terms, the translation is meant to be applied with empty parameter lists (as $t^{\epsilon,\epsilon}$).

Results. In We prove that name elimination, its reverse naming translation (from $\lambda_{\mathtt{tar}}$ to $\lambda_{\mathtt{int}}$), and the composed naming-unwrapping reverse translation (from $\lambda_{\mathtt{tar}}$ to $\lambda_{\mathtt{sou}}$) are termination-preserving strong bisimulations as in Theorem 4.5. The technical development is rather smooth and follows the structure of Section 4 without the subtleties related to the commutation with substitution.

More precisely, in [10, Appendix C], we show that the name elimination translation is a termination-preserving strong bisimulation between $\lambda_{\mathtt{int}}$ and $\lambda_{\mathtt{tar}}$. The two obtained strong bisimulation theorems—namely Theorem 4.5 and the latter concerning name elimination—do not compose, because the first one uses the reverse translation from $\lambda_{\mathtt{int}}$ to $\lambda_{\mathtt{sou}}$, while the second one uses the direct translation from $\lambda_{\mathtt{int}}$ to $\lambda_{\mathtt{tar}}$. To solve the issue, we use the reverse translation from $\lambda_{\mathtt{tar}}$ to $\lambda_{\mathtt{int}}$ dubbed *naming* in Fig. 6.

Unsurprisingly, naming is a termination-preserved strong bisimulation from the target calculus $\lambda_{\mathtt{tar}}$ to the intermediate one $\lambda_{\mathtt{int}}$.

Theorem 5.1 (Target-intermediate termination-preserving strong bisimulation). *Let $t \in \lambda_{\mathtt{tar}}$ be closed.*

1. Projection*: if $t \to_{\mathtt{t}\beta_v} u$ then $t_{\epsilon,\epsilon} \to_{\mathtt{i}\beta_v} u_{\epsilon,\epsilon}$ and if $t \to_{\mathtt{t}\pi} u$ then $t_{\epsilon,\epsilon} \to_{\mathtt{i}\pi} u_{\epsilon,\epsilon}$.*
2. Halt*: t is $\to_{\mathtt{tar}}$-normal (resp. a value, resp. a clash) if and only if $t_{\epsilon,\epsilon}$ is $\to_{\mathtt{tar}}$-normal (resp. a value, resp. a clash).*
3. Reflection*: if $t_{\epsilon,\epsilon} \to_{\mathtt{i}\beta_v} u$ then there exists s such that $t \to_{\mathtt{t}\beta_v} s$ and $s_{\epsilon,\epsilon} = u$, and if $t_{\epsilon,\epsilon} \to_{\mathtt{i}\pi} u$ then there exists s such that $t \to_{\mathtt{t}\pi} s$ and $s_{\epsilon,\epsilon} = u$.*
4. Inverse*: let $u \in \lambda_{\mathtt{int}}$, if $\mathtt{fv}(u) \subseteq \widetilde{y}\#\widetilde{x}$ then $u^{\widetilde{y},\widetilde{x}}_{\widetilde{y},\widetilde{x}} =_\alpha u$.*

The following theorem states that the reverse transformation $\lceil t_{\epsilon,\epsilon} \rceil$ of closure conversion (obtained by composing naming and unwrapping) is a termination-preserving strong bisimulation. It is our final *correctness* result for closure conversion. The proof is obtained by simply composing the two bisimulations results for unwrapping (Theorem 4.5) and naming (Theorem 5.1).

Theorem 5.2 (Target-source termination-preserving strong bisimulation). *Let $t \in \lambda_{\mathtt{tar}}$ be closed.*

1. Projection*: if $t \to_{\mathtt{t}\beta_v} u$ then $\lceil t_{\epsilon,\epsilon} \rceil \to_{\beta_v} \lceil u_{\epsilon,\epsilon} \rceil$ and if $t \to_{\mathtt{t}\pi} u$ then $\lceil t_{\epsilon,\epsilon} \rceil \to_{\pi} \lceil u_{\epsilon,\epsilon} \rceil$.*
2. Halt*: t is $\to_{\mathtt{tar}}$-normal if and only if $\lceil t_{\epsilon,\epsilon} \rceil$ is $\to_{\mathtt{sou}}$-normal.*

(3) Reflection: *if* $\lceil t \rceil_{\epsilon,\epsilon} \to_{\beta_v} u$ *then there exists* s *such that* $t \to_{\mathtt{t}\beta_v} s$ *and* $\lceil s \rceil_{\epsilon,\epsilon} = u$, *and if* $\lceil t \rceil_{\epsilon,\epsilon} \to_{\pi} u$ *then there exists* s *such that* $t \to_{\mathtt{t}\pi} s$ *and* $\lceil s \rceil_{\epsilon,\epsilon} = u$.

(4) Inverse: *if* $u \in \lambda_{\mathtt{sou}}$ *then* $\lceil \underline{u}^{\epsilon,\epsilon} \rceil_{\epsilon,\epsilon} = u$.

6 Part 2 Preliminaries: Abstract Machines

This section starts the second part of the paper. Here, we introduce the terminology and the form of implementation theorem that we adopt for our abstract machines, along the lines of Accattoli and co-authors [2, 5, 13], here adapted to handle clashes.

Abstract Machines Glossary. An *abstract machine* for a strategy $\to_{\mathtt{str}}$ of a calculus λ_{cal} is a quadruple $\mathtt{M} = (\mathtt{States}, \rightsquigarrow, \cdot^\circ, \underline{\cdot})$ where $(\mathtt{States}, \rightsquigarrow)$ is a labeled transition system with transitions $\rightsquigarrow$ partitioned into *principal transitions* $\rightsquigarrow_{pr}$, corresponding to the steps of the strategy and labeled with the labels of the rewrite rules in λ_{cal} (here variants of β_v and π steps), and *overhead transitions* $\rightsquigarrow_{oh}$, that take care of the various tasks of the machine (searching, substituting, and α-renaming), together with two functions:

- *Initialization* $\cdot^\circ : \lambda_{cal} \to \mathtt{States}$ turns λ_{cal}-terms into states;
- *Read-back* $\underline{\cdot} : \mathtt{States} \to \lambda_{cal}$ turns states into λ_{cal}-terms and satisfies the constraint $\underline{t^\circ} = t$ for every λ_{cal}-term t.

A state $q \in \mathtt{States}$ is composed by the *active term* t, and some data structures. A state q is *initial* for t if $t^\circ = q$. A state is *final* if no transitions apply; final states are partitioned into *successful* and *clash* states. A *run* r is a possibly empty sequence of transitions. For runs, we use notations such as $|r|$ as for evaluation sequences (Sect. 2). An *initial run* (*from* t) is a run from an initial state t°. A state q is *reachable* if it is the target state of an initial run.

Abstract machines manipulate *pre-terms*, that is, terms without implicit α-renaming, even if for simplicity we keep calling them terms. In that setting, we write t^α in a state q for a *fresh renaming* of t, i.e. t^α is α-equivalent to t but all of its bound variables are fresh (with respect to those in t and in the other components of q).

Implementation Theorem, Abstractly. We now define when a machine *implements* the strategy $\to_{\mathtt{str}}$ of a calculus λ_{cal}, abstracting and generalizing the setting of the previous sections.

Definition 6.1 (Machine implementation). A machine $\mathtt{M} = (\mathtt{States}, \rightsquigarrow, \cdot^\circ, \underline{\cdot})$ *implements the strategy* $\to_{\mathtt{str}}$ of a calculus λ_{cal} when given a $t \in \lambda_{cal}$ the following holds:

(1) *Runs to evaluations*: for any $\mathtt{M}$-run $r : t^\circ \rightsquigarrow^* q$ there exists a $\to_{\mathtt{str}}$-evaluation $e : t \to^*_{\mathtt{str}} \underline{q}$. Additionally, if q is a successful state then $\underline{q}$ is a clash-free $\to_{\mathtt{str}}$-normal form.

(2) *Evaluations to runs*: for every $\to_{\mathtt{str}}$-evaluation $e : t \to^*_{\mathtt{str}} u$ there exists a $\mathtt{M}$-run $r : t^\circ \rightsquigarrow^* q$ such that $\underline{q} = u$. Additionally, if u is a clash-free $\to_{\mathtt{str}}$-normal form then there exists a successful state q' such that $q \rightsquigarrow^*_{oh} q'$.

(3) *Principal matching*: in both previous points the number $|e|_l$ of steps of of the evaluation e of label l are exactly the number $|r|_l$ of principal l transitions in r, i.e. $|e|_l = |r|_l$.

Next, we give sufficient conditions that a machine and a deterministic strategy have to satisfy in order for the former to implement the latter, what we call *an implementation system.*

M-Closures		Local Envs	Stacks
$c ::= (t, E)$		$E ::= \epsilon \mid [x \leftarrow c] : E'$	$S ::= \epsilon \mid \bullet c : S \mid \circ c : S$

M-Clos.		Stack	Trans.	M-Clos.		Stack
$t\,u$	E	S	$\rightsquigarrow_{sea_1}$	u	E	$\circ(t, E) : S$
$\lambda x.t$	E	$\circ c : S$	$\rightsquigarrow_{sea_2}$	c		$\bullet(\lambda x.t, E) : S$
$\lambda x.t$	E	$\bullet c : S$	$\rightsquigarrow_{\beta_v}$	t	$[x \leftarrow c] : E$	S
x	E	S	$\rightsquigarrow_{sub}$	$E(x)$		S

Figure 7: The local abstract machine (LAM) for $\lambda_{\mathtt{cbv}}$

Definition 6.2 (Implementation system). A machine $\mathtt{M} = (\mathtt{States}, \rightsquigarrow, \cdot^\circ, \underline{\cdot})$ and a strategy $\to_{\mathtt{str}}$ form an *implementation system* if:

(1) *Overhead transparency*: $q \rightsquigarrow_{oh} q'$ implies $\underline{q} = \underline{q'}$;

(2) *Principal projection*: $q \rightsquigarrow_{pr} q'$ implies $\underline{q} \to_{\mathtt{str}} \underline{q'}$ and the two have the same label;

(3) *Overhead termination*: $\rightsquigarrow_{oh}$ terminates;

(4) *Halt*: $\mathtt{M}$ successful states read back to $\to_{\mathtt{str}}$-normal forms, and clash states to clashes of λ_{cal}.

Via a simple lemma for the *evaluation to runs* part (in [10, Appendix D]), we obtain the following abstract implementation theorem.

Theorem 6.3 (Sufficient condition for implementations). *Let* $\mathtt{M}$ *be a machine and* $\to_{\mathtt{str}}$ *be a strategy forming an implementation system. Then,* $\mathtt{M}$ *implements* $\to_{\mathtt{str}}$ *(in the sense of Definition 6.1).*

Local Environments. We overview one the two main forms of environments for abstract machines, the *local* one (as opposed to *global*), for the archetypal call-by-value calculus $\lambda_{\mathtt{cbv}}$ of Sect. 2, to compare it later with the Source TAM for $\lambda_{\mathtt{sou}}$. The terminology local/global is due to [23], and the two techniques are analyzed e.g. (for call-by-name and call-by-need) in [4] but they are folklore.

The *local abstract machine* (LAM), a machine with local environments for $\lambda_{\mathtt{cbv}}$, is defined in Fig. 7. It is a right-to-left variant of the CEK machine [22]. It uses a *stack* to search for β_v-redexes, filling it with entries that encode an evaluation context of $\lambda_{\mathtt{cbv}}$. The substitutions triggered by the encountered β_v-redexes are delayed and stored in environments. There is one principal transition $\rightsquigarrow_{\beta_v}$, of label β_v, and three overhead transitions $\rightsquigarrow_{sea_1}$, $\rightsquigarrow_{sea_2}$, and $\rightsquigarrow_{sub}$ realizing the search for β_v-redexes and substitution. Contrary to the global approach, the LAM has many environments E, paired with terms as to form *m-closures* c. In fact, a local environment E is itself a list of pairs of variables and m-closures: m-closures and environments are defined by mutual induction. A *state* is a triple (t, E, S) where (t, E) is the m-closure of an active term t and a local environment E, and S is a stack. Initial states have shape (t, ϵ, ϵ).

M-Closures are called in this way because when evaluating a closed term an invariant ensures that $\mathtt{fv}(t) \subseteq \mathtt{dom}(E)$ for any m-closure (t, E) in a reachable state (including the active one), where the domain $\mathtt{dom}(E)$ of an environment E is the set of variables on which it has a substitution. The use of m-closures allows one to avoid α-renaming (and copying code) in transition $\rightsquigarrow_{sub}$, at the price of using many environments, thus using space anyway. The duplication of E in transition $\rightsquigarrow_{sea_1}$ is where different implementation approaches (shared *vs* flat) play a main role. With shared environments (whose simplest implementation is as linked lists), the duplication only duplicates the pointer to the environment, not the whole environment. With flat environments (whose simplest

implementation is as arrays), duplication is an actual duplication of the array; we shall further discuss the duplication of flat environments when discussing the complexity of abstract machines.

7 Part 2: the Source TAM for λ_{sou}

Here we present a machine with local environments, the *source tupled abstract machine* (Source TAM) for the source calculus λ_{sou}. We adopt local rather than global environments as to have m-closures, to then show that wrapping removes their need, in the next section.

The Source TAM is defined in Fig. 8. M-Closures carry a flag $\blacktriangleleft \in \{\circ, \bullet\}$, where $\circ$ stands for *non-(completely)-evaluated* and $\bullet$ for *evaluated*. In λ_{sou}, values have a tree structure, the leaves of which are abstractions. The evaluated m-closures $\bullet c$ of the Source TAM have a similar tree structure, plus local environments. A *state* is a couple $(\blacktriangleleft c \mid S)$, where $\blacktriangleleft c$ is a m-closure and S is a stack. The active m-closure is also flagged, naturally inducing a partition of transitions in two blocks. Stack entries are flagged m-closures, π_i and *partially evaluated tuples* $((\widetilde{t}, \downarrow, \widetilde{\bullet c}), E)$ ($\widetilde{t}$ are non-evaluated terms).

The *initialization* of t is the *initial state* $t^\circ := (\circ(t, \epsilon) \mid \epsilon)$. *Successful states* are $(\bullet c \mid \epsilon)$, that is, an evaluated m-closure and an empty stack. Clash and clash-free states are defined in [10, Appendix E].

Transitions. The union of all the transitions of the Source TAM is noted $\leadsto_{\text{STAM}}$. The principal transitions are $\leadsto_{\bullet\beta_v}$ and $\leadsto_{\bullet\pi}$, of label β_v and π, all the other transitions are overhead ones. If $\widetilde{x} = x_1, .., x_n$ and $\overrightarrow{\bullet c} = (\!|\bullet c_1, .., \bullet c_n|\!)$, we set $[\widetilde{x} \leftarrow \overrightarrow{\bullet c}] := [x_1 \leftarrow \bullet c_1] .. [x_n \leftarrow \bullet c_n]$; this notation is used in transition $\leadsto_{\bullet\beta_v}$. In presence of tuples, there are additional overhead transitions. If the active m-closure is:

- $\circ(\pi_i t, E)$ then $\leadsto_{\circ sea_2}$ triggers the evaluation of t and the projection π_i goes on the stack (to trigger transition $\leadsto_{\bullet\pi}$);
- $\circ(\lambda\widetilde{x}.t, E)$ then $\leadsto_{\circ sea_5}$ flips the flag to $\bullet$ (weak evaluation);
- $\circ((\!|\!|), E)$ then $\leadsto_{\circ sea_4}$ changes the flag to $\bullet$, discarding E;
- $\circ((\!|., t_n|\!|), E)$ then $\leadsto_{\circ sea_3}$ evaluates its elements right-to-left, adding a partially evaluated tuple to the stack $((\!|.., \downarrow|\!|), E)$.
- $\bullet c$ then the behavior depends on the first element of the stack. If it is a partially evaluated tuple $((\!|., t, \downarrow, ..|\!|), E)$ then $\leadsto_{\bullet sea_6}$ swaps $\bullet c$ and the next element t in the tuple (duplicating E), similarly to $\leadsto_{\bullet sea_1}$. If the tuple on the stack is $((\!|\downarrow, ..|\!|), E)$ then $\leadsto_{\bullet sea_3}$ plugs $\bullet c$ on $\downarrow$ forming a new evaluated m-closure.

The name of transitions $\leadsto_{\bullet sea_1}$ and $\leadsto_{\bullet sea_3}$ stresses that they do the dual job of $\leadsto_{\circ sea_1}$ and $\leadsto_{\circ sea_3}$—such a duality shall be exploited in the complexity analysis of Sect. 11. Transition $\leadsto_{\bullet sea_6}$ has no dual (it is not named $\leadsto_{\bullet sea_2}$ because it is not the dual of $\leadsto_{\circ sea_2}$).

Invariant and Read Back. Here is an invariant of the Source TAM.

Lemma 7.1 (m-closure invariant). *Let q be a Source TAM reachable state and $\blacktriangleleft c = \blacktriangleleft(t, E)$ be a m-closure or $((\!|., t, .., \downarrow, \widetilde{\bullet c}|\!|), E)$ be a stack entry in q. Then $\mathsf{fv}(t) \subseteq \mathsf{dom}(E)$.*

The read-back $\underline{\cdot}$ of the Source TAM to λ_{sou} is defined in Fig. 8. M-Closures and states read back to terms, stacks read back to evaluation contexts. In the read back of partially evaluated m-closures $((\!|\widetilde{t}, \downarrow, \widetilde{\bullet c}|\!|), E)$ on the stack, E spreads on the non-evaluated terms $\widetilde{t}$ (the m-closures in $\widetilde{\bullet c}$ have their own environments on their leaves).

Lemma 7.2 (Read-back properties).

(1) $\underline{\bullet c}$ *is a value of λ_{sou} for every $\bullet c$ of the Source TAM.*
(2) $\underline{S}$ *is an evaluation context of λ_{sou} for every Source TAM stack S.*

Implementation Theorem. According to the recipe in Sect. 6, we now prove the properties inducing the implementation theorem. The read-back to evaluation contexts (Lemma 7.2) is used for principal projection. The m-closure invariant (Lemma 7.1) is used in the proof of the halt property, to prove that the machine is never stuck on the left-hand side of a $\leadsto_{\circ sub}$ transition (see [10, Appendix E]). Overhead termination is proved via a measure, developed in Sect. 11, which gives a bound on the number of overhead transitions.

Theorem 7.3. *The Source TAM and λ_{sou} form an implementation system (as in Def. 6.2), hence the Source TAM implements $\to_{\mathsf{sou}}$.*

8 Part 2: the Int TAM for $\lambda_{\mathtt{int}}$

Here we present the *intermediate tupled abstract machine* (Int TAM) for the intermediate calculus $\lambda_{\mathtt{int}}$. Its feature is a new way of handling environments, resting on the strong properties of $\lambda_{\mathtt{int}}$.

The Int TAM is defined in Fig. 9. The flags will be explained after an overview of the new aspects of the machine. Mostly, the Int TAM behaves as the Source TAM by just having removed the structure of m-closures. The principal transitions are $\leadsto_{\bullet\beta_v}$ and $\leadsto_{\bullet\pi}$, of label β_v and π, all the other transitions are overhead. There are two new aspects: transition $\leadsto_{\circ sub_c}$ that evaluates variable bags and the use of *stackable environments* rather than *local* environments and m-closures. The machine is akin to those with global environments, except that no α-renaming is needed, as in the local approach.

Evaluating Bags. In the Int TAM, abstractions $\lambda\widetilde{x}.t$ are replaced by non-evaluated prime closures $\circ[\![\widetilde{y}; \widetilde{x}.t | (\!|\widetilde{y}|\!)]\!]$. The new transition $\leadsto_{\circ sub_c}$ (replacing $\leadsto_{\circ sea_5}$) substitutes on all variables in $\widetilde{y}$ in one shot, producing the evaluated closure $\bullet[\![\widetilde{y}; \widetilde{x}.t | E(\widetilde{y})]\!]$ where $E(\widetilde{y}) = (\!|E(y_1), .., E(y_n)|\!)$ if $\widetilde{y} = y_1, .., y_n$, and E is the (global) environment.

Stackable Environments. Transitions $\leadsto_{\bullet\beta_v}$ and the new $\leadsto_{\bullet sea_7}$ encapsulate a second new aspect, *stackable environments*. Indeed:

- *Closure bodies and environments*: when the machine encounters the analogous of a $\to_{\mathtt{i}\beta_v}$-redex $[\![\widetilde{y}; \widetilde{x}.t | \overrightarrow{v_1}]\!]\ \overrightarrow{v_2}$, the new entries $[\widetilde{y} \leftarrow \overrightarrow{\bullet v_1}]\ [\widetilde{x} \leftarrow \overrightarrow{\bullet v_2}]$ of the environment created by transition $\leadsto_{\bullet\beta_v}$ in the machine are all that is needed to evaluate t, because the free variables of t are all among $\widetilde{y}$ and $\widetilde{x}$. Thus, the environment E that is active before firing the redex is useless to evaluate t, and can be removed after transition $\leadsto_{\bullet\beta_v}$.
- *Stackability*: E is not garbage collected, it is pushed on the new *activation stack*, along with the ordinary stack S (now called *constructor stack*) which contains non-evaluated terms with variables in $\mathsf{dom}(E)$. This is still done by transition $\leadsto_{\bullet\beta_v}$.
- *Popping*: when the body t of the closure has been evaluated, the focus is on a value $\bullet v$ and the constructor stack is empty. The activation stack has the pair (S, E) that was active before firing the β_v-redex. The machine throws away the current environment $E' := [\widetilde{y} \leftarrow \overrightarrow{\bullet v_1}]\ [\widetilde{x} \leftarrow \overrightarrow{\bullet v_2}]$, since $\widetilde{y}$ and $\widetilde{x}$ have no occurrences out of t, and reactivates the pair (S, E), to keep evaluating terms in S. This is done by the new transition $\leadsto_{\bullet sea_7}$.

Flags. The Int TAM evaluates (well-formed) terms of $\lambda_{\mathtt{int}}$ decorated (only on top) with a flag $\blacktriangleleft \in \{\circ, \bullet\}$: $\circ t$ denotes that t has not been evaluated yet, while $\bullet t$ denotes that t has been evaluated. The results of evaluation are values, thus the $\bullet$ flag shall be associated to values only, and an invariant shall ensure that every evaluated value

Eval. m-clos.
$$\bullet c ::= \bullet(\lambda\widetilde{x}.t, E) \mid \overbrace{(\!|\bullet c_1, .., \bullet c_n|\!)}^{\vec{\bullet c}} \ n \geq 0$$
Flagged m-clos.
$$\circledcirc c ::= \bullet c \mid \circ(t, E)$$
Local envs
$$E, E' ::= \epsilon \mid [x \leftarrow \bullet c] : E$$
St. entries
$$S_{en} ::= \circledcirc c \mid \pi_i \mid (\!|\widetilde{t}, \downarrow, \widetilde{\bullet c}|\!), E)$$
Stacks
$$S, S' ::= \epsilon \mid S_{en} : S$$
States
$$q, q' ::= (\circledcirc c \mid S)$$

m-closure			Stack	Trans.	m-closure			Stack	
$\circ$	tu	E	S	$\rightsquigarrow_{\circ sea_1}$	$\circ$	u	E	$\circ(t, E) : S$	
$\circ$	$\pi_i t$	E	S	$\rightsquigarrow_{\circ sea_2}$	$\circ$	t	E	$\pi_i : S$	
$\circ$	$(\!|., t_n|\!)$	E	S	$\rightsquigarrow_{\circ sea_3}$	$\circ$	t_n	E	$((\!|., \downarrow|\!), E) : S$	
$\circ$	$(\!|\,|\!)$	E	S	$\rightsquigarrow_{\circ sea_4}$	$\bullet$	$(\!|\,|\!)$	ϵ	S	
$\circ$	$\lambda\widetilde{x}.t$	E	S	$\rightsquigarrow_{\circ sea_5}$	$\bullet$	$\lambda\widetilde{x}.t$	E	S	
$\circ$	x	E	S	$\rightsquigarrow_{\circ sub}$		$E(x)$		S	
	$\bullet c$		$\circ(t, E) : S$	$\rightsquigarrow_{\bullet sea_1}$	$\circ$	t	E	$\bullet c : S$	
	$\bullet c$		$((\!|., t, \downarrow, ..|\!), E) : S$	$\rightsquigarrow_{\bullet sea_6}$	$\circ$	t	E	$((\!|., \downarrow, \bullet c, ..|\!), E) : S$	
	$\bullet c$		$((\!|\downarrow, ..|\!), E) : S$	$\rightsquigarrow_{\bullet sea_3}$		$(\!|\bullet c, ..|\!)$		S	
$\bullet$	$\lambda\widetilde{x}.t$	E	$\vec{\bullet c} : S$	$\rightsquigarrow_{\bullet \beta_v}$	$\circ$	t	$[\widetilde{x} \leftarrow \vec{\bullet c}] : E$	S	(∗)
	$\vec{\bullet c}$		$\pi_i : S$	$\rightsquigarrow_{\bullet \pi}$		$\bullet c_i$		S	(#)

Side conditions: (∗) if $\|\widetilde{x}\| = \|\vec{\bullet c}\|$ (#) if $1 \leq i \leq \|\vec{\bullet c}\|$

States read back $\underline{\cdot}$ (to terms of λ_{sou})
$$\underline{(\circledcirc c \mid S)} := \underline{S}\langle \underline{\circledcirc c} \rangle$$

M-Closures read-back $\underline{\cdot}$ (to terms of λ_{sou})
$$\underline{\circledcirc(t, \epsilon)} := t \qquad \underline{(\!|\bullet c_1, .., \bullet c_n|\!)} := (\!|\underline{\bullet c_1}, .., \underline{\bullet c_n}|\!)$$
$$\underline{\circledcirc(t, [x \leftarrow \bullet c] : E)} := \underline{(t\{x \leftarrow \bullet c\}, E)}$$

Stacks read-back $\underline{\cdot}$ (to evaluation contexts of λ_{sou})
$$\underline{\epsilon} := \langle\cdot\rangle \qquad \underline{\circ(t, E) : S} := \underline{S}\langle \underline{\circ(t, E)}\langle\cdot\rangle\rangle \qquad \underline{\bullet c : S} := \underline{S}\langle\langle\cdot\rangle \underline{\bullet c}\rangle$$
$$\underline{\pi_i : S} := \underline{S}\langle\pi_i\langle\cdot\rangle\rangle \qquad \underline{((\!|t_1, .., t_n, \downarrow, \vec{\bullet c}|\!), E) : S} := \underline{S}\langle(\!|\underline{(\!|t_1, E|\!)}, .., \underline{(\!|t_n, E|\!)}, \langle\cdot\rangle, \underline{\vec{\bullet c}}|\!)\rangle$$

Figure 8: The source tupled abstract machine (Source TAM).

Stackable envs $E, E' ::= \epsilon \mid [x \leftarrow \bullet v] : E$ Activation stacks $A, A' ::= \epsilon \mid (S, E) : A$
Constructor stacks $S, S' ::= \epsilon \mid \circ t : S \mid \bullet v : S \mid \pi_i : S \mid (\!|\widetilde{\circ t}, \downarrow, \widetilde{\bullet v}|\!) : S$ States $q, q' ::= (\circledcirc t \mid S \mid E \mid A)$

Focus	Cons. stack	Env	Act. stack	trans.	Focus	Cons. stack	Env	Act. stack	
$\circ(tu)$	S	E	A	$\rightsquigarrow_{\circ sea_1}$	$\circ u$	$\circ t : S$	E	A	
$\circ\pi_i t$	S	E	A	$\rightsquigarrow_{\circ sea_2}$	$\circ t$	$\pi_i : S$	E	A	
$\circ(\!|t_1, .., t_n|\!)$	S	E	A	$\rightsquigarrow_{\circ sea_3}$	$\circ t_n$	$(\!|\circ t_1, .., \downarrow|\!) : S$	E	A	
$\circ(\!|\,|\!)$	S	E	A	$\rightsquigarrow_{\circ sea_4}$	$\bullet(\!|\,|\!)$	S	E	A	
$\circ[\![\widetilde{y}; \widetilde{x}.t \mid (\!|\widetilde{y}|\!)]\!]$	S	E	A	$\rightsquigarrow_{\circ sub_c}$	$\bullet[\![\widetilde{y}; \widetilde{x}.\circ t \mid E(\widetilde{y})]\!]$	S	E	A	
$\circ x$	S	E	A	$\rightsquigarrow_{\circ sub_v}$	$E(x)$	S	E	A	
$\bullet v$	$\circ t : S$	E	A	$\rightsquigarrow_{\bullet sea_1}$	$\circ t$	$\bullet v : S$	E	A	
$\bullet v$	$(\!|., \circ t, \downarrow, ..|\!) : S$	E	A	$\rightsquigarrow_{\bullet sea_6}$	$\circ t$	$(\!|., \downarrow, \bullet v, ..|\!) : S$	E	A	
$\bullet v$	$(\!|\downarrow, ..|\!) : S$	E	A	$\rightsquigarrow_{\bullet sea_3}$	$\bullet(\!|\bullet v, ..|\!)$	S	E	A	
$\bullet \vec{v}$	$\pi_i : S$	E	A	$\rightsquigarrow_{\bullet \pi}$	$\bullet v_i$	S	E	A	if $1 \leq i \leq \|\vec{v}\|$
$\bullet[\![\widetilde{y}; \widetilde{x}.\circ t \mid \vec{\bullet v_1}]\!]$	$\vec{\bullet v_2} : S$	E	A	$\rightsquigarrow_{\bullet \beta_v}$	$\circ t$	ϵ	$[\widetilde{y} \leftarrow \vec{\bullet v_1}][\widetilde{x} \leftarrow \vec{\bullet v_2}]$	$(S, E) : A$	if $\|\widetilde{y}\| = \|\vec{\bullet v_2}\|$, $\|\widetilde{x}\| = \|\vec{\bullet v_1}\|$
$\bullet v$	ϵ	E	$(S, E') : A$	$\rightsquigarrow_{\bullet sea_7}$	$\bullet v$	S	E'	A	

Read back $\underline{\cdot}$ to $\lambda_{\mathtt{int}}$

Flagged terms $\underline{\circ t} := t \qquad \underline{\bullet[\![\widetilde{y}; \widetilde{x}.\circ t \mid \vec{\bullet v}]\!]} := [\![\widetilde{y}; \widetilde{x}.t \mid \underline{\vec{\bullet v}}]\!] \qquad \underline{\bullet(\!|\bullet v_1, \ldots, \bullet v_n|\!)} := (\!|\underline{\bullet v_1}, \ldots, \underline{\bullet v_n}|\!)$

Constructor stacks $\underline{\epsilon} := \langle\cdot\rangle \qquad \underline{\pi_i : S} := \underline{S}\langle\pi_i\langle\cdot\rangle\rangle \qquad \underline{\bullet v : S} := \underline{S}\langle\langle\cdot\rangle \underline{\bullet v}\rangle \qquad \underline{\circ t : S} := \underline{S}\langle t\langle\cdot\rangle\rangle \qquad \underline{(\!|\widetilde{\circ t}, \downarrow, \widetilde{\bullet v}|\!) : S} := \underline{S}\langle(\!|\widetilde{t}, \langle\cdot\rangle, \widetilde{\bullet v}|\!)\rangle$

xAct. stacks $\underline{\epsilon} := \langle\cdot\rangle \quad \underline{(S, E) : A} := \underline{A}\langle \underline{S}\sigma_E\rangle$ | States $\underline{(\circledcirc t \mid S \mid E \mid A)} := \underline{A}\langle \underline{S}\sigma_E\langle \underline{\circledcirc t}\sigma_E\rangle\rangle$ | Env-induced subst. $\sigma_{[\widetilde{x} \leftarrow \vec{\bullet v}]} := \{\widetilde{x}; \epsilon \leftarrow \underline{\vec{\bullet v}}; (\!|\,|\!)\}$

Figure 9: The intermediate tupled abstract machine (Int TAM).

$\bullet v$ in a reachable state is closed. In $\circ t$, t has no flags, and non-flagged sub-terms are implicitly considered as flagged with $\circ$. Every evaluated value carries a $\bullet$ flag, so in $\bullet v$ there are in general other $\bullet$ flags (when it is a tuple or the bag of the closure). Moreover, evaluated closures shall have shape $\bullet[\![\widetilde{y}; \widetilde{x}.\circ t \mid \vec{\bullet v}]\!]$, that is, they have an additional (redundant) $\circ$ flag on their body (it shall be used in section Sect. 12 for the complexity analysis). The machine is started on *prime* terms of $\lambda_{\mathtt{int}}$ (because wrapped terms of λ_{sou} are prime, Lemma 4.3), thus all closures in an initial state have shape $[\![\widetilde{y}; \widetilde{x}.t \mid (\!|\widetilde{y}|\!)]\!]$. In fact, all non-evaluated closures $\circ[\![\widetilde{y}; \widetilde{x}.t \mid b]\!]$ in reachable states shall always be prime (that is, such that $b = (\!|\widetilde{y}|\!)$): we prove this invariant and we avoid assuming the general shape of non-evaluated closures, which would require additional and never used transitions.

The *initialization* t° of $t \in \lambda_{\mathtt{int}}$ is given by the *initial state* $(\circ t \mid \epsilon \mid \epsilon \mid \epsilon)$ where t is closed and prime. *Successful states* have shape $(\bullet v \mid \epsilon \mid E \mid \epsilon)$. Clash states are defined in [10, Appendix F].

Invariants and Read Back. Here are the invariants of the Int TAM.

Lemma 8.1 (Invariants). *Let $q = (\circledcirc t \mid S \mid E \mid A)$ be a Int TAM reachable state.*

1. Well-formedness*: all closures in q are well-formed.*
2. Closed values*: every value $\bullet v$ in q is closed.*
3. Closure*: $\mathtt{fv}(t) \cup \mathtt{fv}(S) \subseteq \mathtt{dom}(E)$ and $\mathtt{fv}(S') \subseteq \mathtt{dom}(E')$ for every entry (S', E') of the activation stack A.*

The read back $\underline{\cdot}$ is defined in Fig. 9. Flagged terms $\circledcirc t$ and states q read back to terms of $\lambda_{\mathtt{int}}$, the constructor and activation stacks S and A read back to evaluation contexts of $\lambda_{\mathtt{int}}$. The read back of A and q is based on a notion of meta-level *environment-induced (simultaneous) substitution* σ_E, defined in Fig. 9 and applied to terms (as read back of flagged terms) and evaluation contexts (as read back of stacks); meta-level substitutions are extended to evaluation contexts as expected, the definition is in [10, Appendix F].

The implementation theorem is proved following the schema used for the Source TAM, see [10, Appendix F]. Overhead transparency for the additional transition $\rightsquigarrow_{\bullet sea_7}$ relies on the closed values invariant (Lemma 8.1).

Tupled envs $E, E' ::= \overrightarrow{\bullet v_{\mathtt{w}}}; \overrightarrow{\bullet v_{\mathtt{s}}}$

Tupled envs lookup $(\mathtt{v} \in \{\mathtt{w}, \mathtt{s}\})$ $(\overrightarrow{\bullet v_{\mathtt{w}}}; \overrightarrow{\bullet v_{\mathtt{s}}})(\pi_i \mathtt{v}) := (\overrightarrow{\bullet v_{\mathtt{v}}})_i$ if $i \leq \|\overrightarrow{\bullet v_{\mathtt{v}}}\|$

Focus	Co.	En	Ac.	trans.	Focus	Co.	En	Act. st.
$\circ[\![t \mid \vec{p}]\!]$	S	E	A	$\rightsquigarrow_{\circ sub_c}$	$\bullet[\![\circ t \mid E(\vec{p})]\!]$	S	E	A
$\circ p$	S	E	A	$\rightsquigarrow_{\circ sub_v}$	$E(p)$	S	E	A
$\bullet[\![\circ t \mid \overrightarrow{\bullet v_1}]\!]$	$\overrightarrow{\bullet v_2} : S$	E	A	$\rightsquigarrow_{\bullet \beta_v}$	$\circ t$	ϵ	$\overrightarrow{\bullet v_1}; \overrightarrow{\bullet v_2}$	$(S, E) : A$

Side conditions: if look-up is defined in $\rightsquigarrow_{\circ sub_c}$ and $\rightsquigarrow_{\circ sub_v}$.

Read back $\underline{\cdot}$ to $\lambda_{\mathtt{tar}}$

Flagged terms $\underline{\bullet[\![\circ t \mid \overrightarrow{\bullet v}]\!]} := [\![t \mid \overrightarrow{\underline{\bullet v}}]\!]$

Env-induced subst. $\sigma_{\overrightarrow{\bullet v_{\mathtt{w}}}; \overrightarrow{\bullet v_{\mathtt{s}}}} := \{\!\{\mathtt{w}; \mathtt{s} \leftarrow \overrightarrow{\bullet v_{\mathtt{w}}}; \overrightarrow{\bullet v_{\mathtt{s}}}\}\!\}$

Figure 10: The target tupled abstract machine (Target TAM).

Lemma 8.2 (Read back properties).

(1) Values: *$\underline{\bullet v}$ is a value of $\lambda_{\mathtt{int}}$ for every $\bullet v$ of the Int TAM.*
(2) Evaluation contexts: *$\underline{S}$ and $\underline{A}$ are evaluation contexts of $\lambda_{\mathtt{int}}$ for every Int TAM constructor and activation stacks S and A.*

Theorem 8.3. *The Int TAM and $\lambda_{\mathtt{int}}$ form an implementation system (as in Def. 6.2), thus the Int TAM implements $\rightarrow_{\mathtt{int}}$ on prime terms.*

9 Part 2: the Target TAM for $\lambda_{\mathtt{tar}}$

Here we present the *target tupled abstract machine* (Target TAM) for the target calculus $\lambda_{\mathtt{int}}$, a minor variant of the Int TAM. Beyond the elimination of variable names, its key feature is the use of *tupled environment*, that is, a pair of tuples as data structures for environments, instead of a list of explicit substitution entries $[x \leftarrow \bullet v]$.

The Target TAM is defined in Fig. 10, by giving the only ingredients of the Int TAM that are redefined for the Target TAM, leaving everything else unchanged but for the fact that, when considering the omitted transitions of the Int TAM as transitions of the Target TAM, the symbol E for environment refers to the new notion of environment adopted here (the omitted transitions do not touch the environment). Here the closures $[\![t|b]\!]_{n,m}$ of $\lambda_{\mathtt{tar}}$ are written $[\![t|b]\!]$ (and are then decorated with flags as for the Int TAM) because n and m play a role only for the reverse translation from $\lambda_{\mathtt{tar}}$ to $\lambda_{\mathtt{int}}$ studied in [10, Appendix C], while here they are irrelevant.

The elimination of names enables the use of a *tupled (stackable) environment*: the environment is now a pair of tuples $\overrightarrow{\bullet v_{\mathtt{w}}}; \overrightarrow{\bullet v_{\mathtt{s}}}$, where $\overrightarrow{\bullet v_{\mathtt{w}}}$ provides values for the wrapped projected variables $\pi_i \mathtt{w}$, and $\overrightarrow{\bullet v_{\mathtt{s}}}$ for the source ones $\pi_i \mathtt{s}$, with no need to associate the values of these tuples to variable names via entries of the form $[x \leftarrow \bullet v]$ (as it was the case for the Int TAM). The change is relevant, as the data structure for environments changes from a *map* to a *tuple*, removing the need (and the cost) of creating a map in transition $\rightsquigarrow_{\bullet \beta_v}$ and inducing a logarithmic speed-up, as we shall see.

The look up into tupled environments is defined in Fig. 10 and is the only new notion needed in the new transitions. The implementation theorem is proved following the same schema used for the Source TAM and the Int TAM, the details are in [10, Appendix G].

Theorem 9.1. *The Target TAM implements $\rightarrow_{\mathtt{tar}}$ on prime terms.*

Actual Implementation of the Target TAM. We provide an OCaml implementation of the Target TAM on GitHub [33], described in [10, Appendix K]. The textual interface asks for a term of the source calculus $\lambda_{\mathtt{sou}}$, which is translated to the target calculus $\lambda_{\mathtt{tar}}$ by applying first wrapping and then name elimination, thus passing through the intermediate calculus $\lambda_{\mathtt{int}}$, as described above. The obtained $\lambda_{\mathtt{tar}}$-term is then reduced by the Target TAM until a normal form is reached, if any, and the final $\lambda_{\mathtt{tar}}$-term is extracted. The machine state is printed after every step, in ASCII art.

The implementation is not particularly optimized and it does not have a graphical user interface. It is designed to stay as close as possible to the definitions given in the paper, and to provide evidence supporting the assumptions of the cost analysis of Sect. 12. In particular, we use OCaml arrays for tuples, variables in abstractions, and bags in closures, for achieving $O(1)$ access times.

10 Part 3 Preliminaries: Sharing, Size Explosion, and the Complexity of Abstract Machines

This section starts the third part of the paper, about the time complexity analysis of abstract machines. Here, we quickly overview the size explosion problem of the λ-calculus as the theoretical motivation for the use of sharing in implementations, as well as the structure of the study of the overhead of abstract machines.

Size Explosion. A well-known issue of the λ-calculus is the existence of families of terms whose size grows *exponentially* with the number of β-steps. They are usually built exploiting some variant of the duplicator $\delta := \lambda x.xx$. We give an example in $\lambda_{\mathtt{cbv}}$. Define:

Variant of δ	Size explod. family		Exploded results	
$\pi := \lambda x.\lambda y.yxx$	$t_0 := \mathtt{I}$	$t_{n+1} := \pi t_n$	$u_0 := \mathtt{I}$	$u_{n+1} := \lambda y.yu_n u_n$

Proposition 10.1 (Size Explosion in $\lambda_{\mathtt{cbv}}$). *Let $n \in \mathbb{N}$. Then $t_n \rightarrow_{\beta_v}^n u_n$, moreover $|t_n| = O(n)$, $|u_n| = \Omega(2^n)$, and u_n is a value.*

The proof is in [10, Appendix H]. Size explosion has been extensively analyzed in the study of reasonable cost models—see [1] for an introduction—because it suggests that the number n of $\rightarrow_{\beta_v}$ steps is not a reasonable time measure for the execution of λ-terms: for size exploding families, indeed, it does not even account for the time to write down the normal form, which is of size $\Omega(2^n)$.

One is tempted to circumvent the problem by tweaking the calculus, with types, by changing the evaluation strategy, restricting to CPS, and so on. None of these tweaks works, size explosion can always be adapted: it is an inherent feature of higher-order computations [1].

Sharing for Functions. A solution nonetheless exists: it amounts to add a way to *share* sub-terms to avoid their blind duplication during evaluation. For size explosion in $\lambda_{\mathtt{cbv}}$, it is enough to add a simple form of *sub-term sharing* by delaying meta-level substitution and avoiding substituting under abstractions. For instance, evaluating the size exploding term t_n above in a variant of $\lambda_{\mathtt{cbv}}$ where the mentioned sub-term sharing is implemented via explicit substitutions, gives the following normal form, of size *linear* (rather than exponential) in n:

$$\lambda y_1.y_1x_1x_1[x_1 \leftarrow \lambda y_2.y_2x_2x_2] \dots [x_{n-1} \leftarrow \lambda y_n.y_nx_nx_n][x_n \leftarrow \mathtt{I}].$$

The explosion re-appears if one unfolds that normal form to an ordinary λ-term, but it is now encapsulated in the unfolding.

Abstract machines of the previous sections have environments to implement sub-term sharing and avoid the size explosion due to β_v, giving hope for a time complexity lower than exponential.

Parameters for the Time Complexity Analysis of Abstract Machines. Given a strategy $\rightarrow_{\mathtt{str}}$ of a calculus λ_{cal} and an abstract machine $\mathtt{M}$

implementing $\rightarrow_{\mathtt{str}}$ (see Definition 6.1), the time complexity of M is obtained by estimating the cost—when concretely implemented on random access machines (RAMs)—of a run $r_e : t_0^\circ \rightsquigarrow^* q$ implementing an arbitrary evaluation sequence $e : t_0 \rightarrow^n_{\mathtt{str}} t_n$ (thus having $\underline{q} = t_n$) as a function of two parameters:

(1) *Code size*: the size $|t_0|$ of the initial term t_0;
(2) *Number of $\rightarrow_{\mathtt{str}}$-steps/$\beta$-steps*: the number n of $\rightarrow_{\mathtt{str}}$-steps in e. If λ_{cal} has other rules other than β/β_v, the parameter is often just the number of β/β_v steps, which is usually considered the relevant time cost model.

Recipe for Time Complexity. The way the time complexity of an abstract machine is established tends to follow the same schema:

(1) *Number of overhead transitions*: bounding the number of overhead transitions as a function of $|t_0|$ and n (which by the principal matching property of implementations—see Definition 6.1—is enough to bound the length of r_e);
(2) *Cost of single transitions*: bounding the cost of single transitions, which is typically constant or depends only on $|t_0|$;
(3) *Total cost*: inferring the total cost of a run r by multiplying the number of steps of each kind of transition for their cost, and summing all the obtained costs.

The key tool for such an analysis is the *sub-term property*, an *invariant* of abstract machines stating that some of the terms in a reachable state are sub-terms of the initial term. This allows one to develop bounds with respect to the size $|t_0|$ of the initial term t_0.

Time Complexity of the LAM. The LAM verifies a sub-term invariant, and its time complexity follows a well-known schema in the literature, closely inspected by Accattoli and Barras [4] for call-by-name and call-by-need, and that smoothly adapts to CbV. Consider an evaluation $e : t_0 \rightarrow^n_{\beta_v} t_n$ in $\lambda_{\mathtt{cbv}}$. About the first point of the recipe, the bound on overhead transitions is $O((n+1)\cdot|t_0|)$, that is, bilinear. Additionally, if one takes *complete* evaluations, that is, for which $t_n = v$ is a value, then the bound lowers to $O(n)$. Such an independence from the initial term is due to the fact that whether a term is a value can be checked in $O(1)$ in $\lambda_{\mathtt{cbv}}$, by simply checking whether the top-most constructor is an abstraction.

For the second point of the recipe, the cost of single transitions of the LAM depends on the data structures used for local environments, as discussed by Accattoli and Barras [4]. With flat environments, the cost of manipulating them is $O(|t_0|)$ (because of the duplication of E in $\rightsquigarrow_{sea_1}$), giving a total cost of $O(n\cdot|t_0|)$ for complete runs. With shared environments, the best structures manipulate them in $O(\log|t_0|)$, giving a total cost of $O(n\cdot\log|t_0|)$ for complete runs. Thus, shared local environments are faster, but they are optimized for time and inefficient with respect to space, as they prevent some garbage collection to take place. Here, we take as reference *flat* local environment, which induce the same overall $O(n\cdot|t_0|)$ overhead as global environments and enable a better management of space (not discussed here, see [9] instead).

Notions of Flatness for Local Environment. Let us be precise on a subtle point about local environments and their flatness. Local environments are defined by mutual recursion with m-closures, and various notions of sharing and flatness are possible, as one can share environments, or m-closures, both, or none of them. Sharing both is essentially the same as sharing only environments.

Sharing m-closures rather than environments is what we above called *flat environments*. For instance, if $E = [x_1 \leftarrow \bullet c_1] : [x_2 \leftarrow \bullet c_2] : \epsilon$ then $\bullet c_1$ and $\bullet c_2$ are shared so that the concrete representation of E is $E = [x_1 \leftarrow p_1] : [x_2 \leftarrow p_2] : \epsilon$ where p_1 and p_2 are pointers to $\bullet c_1$ and $\bullet c_2$. Copying E then means copying $[x_1 \leftarrow p_1] : [x_2 \leftarrow p_2] : \epsilon$, *without* recursively copying the structure of $\bullet c_1$ and $\bullet c_2$.

The *super flat environments* obtained by removing sharing for *both* environments and m-closures are studied in [9], where it is shown that the overhead of abstract machines becomes *exponential*.

No Sharing for Stackable Environments. An interesting aspect of the new stackable environments is that they need no sharing. Indeed, the Int TAM and the Target TAM never duplicate their stackable environments. In particular, when they discard the current environment E in transition $\rightsquigarrow_{\bullet sea_7}$, it can be collected.

De Bruijn Indices Do Not Change the Overhead. It is well-known that, representing flat environments as arrays and variables with de Bruijn indices, one can look-up environments in $O(1)$ rather than in $O(\log|t_0|)$, which is the best that one can do with (some ordered domain of) names. However, the $O(|t_0|)$ cost of copying flat environments in transition $\rightsquigarrow_{sea_1}$ dominates, so that turning to de Bruijn indices does not change the overall $O(n\cdot|t_0|)$ overhead.

Size Exploding Tuples. In $\lambda_{\mathtt{sou}}$, there is a new form of size explosion, due to tuples, requiring another form of sharing. To our knowledge, the view provided here is novel. Set $\mathtt{I} := \lambda z.z$ and:

Variant of δ	Size explod. family	Exploded results
$\tau := \lambda x.(\!|x,x|\!)$	$s_0 := \mathtt{I} \quad s_{n+1} := \tau(\!|s_n|\!)$	$r_0 := \mathtt{I} \quad r_{n+1} := (\!|r_n, r_n|\!)$

Proposition 10.2 (Size explosion of tuples). *Let $n \in \mathbb{N}$. Then $s_n \rightarrow^n_{\beta_v} r_n$, moreover $|s_n| = O(n)$, $|r_n| = \Omega(2^n)$, and r_n is a value.*

Sharing for Tuples. To avoid the size explosion of tuples, another form of sharing is used. The idea is the same as for functions: forms of size explosion are circumvented by forms of sharing designed to limit the substitution process. The key point is that tuples should never be copied, only *pointers* to them should be copied, thus representing r_n above using *linear* (rather than exponential) space in n, as follows (where the p_i are *pointers* and $[p_i \leftarrow v]$ are *heap entries*):

$$(\!|p_1, p_1|\!)\,[p_1 \leftarrow (\!|p_2, p_2|\!)]\ldots[p_n \leftarrow (\!|p_n, p_n|\!)]\,[p_n \leftarrow \mathtt{I}]$$

The abstract machines of the previous sections have environments for sub-term sharing (needed for β_v) but they do *not explicitly* handle tuple sharing. The reason is practical: explicitly handling tuple sharing would require a treatment of *pointers* and a *heap* (i.e., a further global environment) and more technicalities. Our machines, however, are meant to be concretely implemented with tuple sharing, as in our OCaml implementation of the Target TAM [33].

11 Part 3: Complexity of the Source TAM, or, Tuples Raise the Overhead

Here, we develop the time complexity analysis of the Source TAM, stressing the novelty of tuples (see [10, Appendix I] for proofs).

The Source TAM verifies the following *sub-term invariant*.

Lemma 11.1 (Sub-term invariant). *Let q be a Source TAM reachable state from the initial state t°.*

Size of λ_{sou} terms

$$|x| := 1 \qquad |\lambda\widetilde{x}.t| := |t| + \|\widetilde{x}\| + 1 \qquad |tu| := |t| + |u| + 1$$
$$|\pi_i t| := |t| + 1 \qquad |(\!|t_1, \ldots, t_n|\!)| := n + \textstyle\sum_{i=1}^{n} |t_i|$$

Overhead measure for the Source TAM

$$\text{Con. stack entries } S_{en} \quad |\bullet c|_{oh} := 0 \quad |\circ(t,E)|_{oh} := |t| \quad |\pi_i|_{oh} := 0$$
$$|(\!|\circ t_1, .., \circ t_n, \downarrow, \widetilde{\bullet c}|\!), E)|_{oh} := n + \textstyle\sum_{i=1}^{n} |t_i|_{oh}$$
$$\text{Stacks} \quad |\epsilon|_{oh} := 0 \quad |S_{en} : S|_{oh} := |S_{en}|_{oh} + |S|_{oh}$$
$$\text{States} \quad |(\!|c \mid S|\!)|_{oh} := |\!|c|_{oh} + |S|_{oh}$$

Figure 11: Size $|\cdot|$ of λ_{sou} terms and overhead measure $|\cdot|_{oh}$.

(1) *u is a sub-term of t for every m-closure of shape $\circ(u, E)$ or $((\!|.., u, .., \downarrow, \vec{\bullet c}|\!), E)$ in q.*
(2) *$\lambda\widetilde{x}.u$ is a sub-term of t for every m-closure $\bullet(\lambda\widetilde{x}.u, E)$ in q.*

Tuples are Not Sub-Terms. Point 2 only concerns evaluated m-closures containing abstractions, and *not* evaluated tuples. Consider $t := (\lambda x.\lambda y.(\!|x, y|\!))(\!|\mathtt{I}|\!)(\!|\delta|\!) \to_{\beta_v} (\lambda y.(\!|\mathtt{I}, y|\!))(\!|\delta|\!) \to_{\beta_v} (\!|\mathtt{I}, \delta|\!)$ in λ_{sou} and note that $(\!|\mathtt{I}, \delta|\!)$ is *not* a sub-term of t. The run of the Source TAM on t produces a m-closure $((\!|\bullet\mathtt{I}, \bullet\delta|\!), E)$ for some E. The leaves of the tree-structure of an evaluated tuple are abstractions ($\mathtt{I}$ and δ in the example), which are initial sub-terms, but they might be arranged in ways that were not present in the initial term.

A consequence of this fact is that when the Source TAM starts evaluating a non-empty tuple $(\!|.., t_n|\!)$ with transition $\rightsquigarrow_{\circ sea_3}$, by adding $(\!|.., \downarrow|\!)$ to the stack, it has to *allocate a new tuple* on the heap for $(\!|.., \downarrow|\!)$ (which has a cost, discussed below). This never happens in absence of tuples, that is, in the LAM of Fig. 7. More precisely, the LAM does not copy any code, but it has to allocate new pointers to local environments, when they are extended by $\rightsquigarrow_{\beta_v}$.

Step 1 of the Recipe: Number of Transitions and Overhead Measure. For establishing a bound on overhead transitions, we first factor some of them ($\rightsquigarrow_{\bullet sea_{1,3}}$) out by simply noticing that they are enabled and thus bound by some others ($\rightsquigarrow_{\circ sea_{1,3}}$). Actually, the same is true also for the principal transition $\rightsquigarrow_{\bullet\pi}$, bounded by $\rightsquigarrow_{\circ sea_2}$.

Lemma 11.2 (Transition match). *Let r be a Source TAM run. Then $|r|_{\pi,\bullet sea_{1,3}} \leq |r|_{\circ sea_{1,2,3}}$.*

For the other transitions, we use a measure $|\cdot|_{oh}$, defined in Fig. 11 together with the size $|t|$ of λ_{sou} terms, which we use to derive a bilinear bound on overhead/projection transitions (Prop. 11.4).

Lemma 11.3 (Overhead measure properties). *Let $t^\circ \rightsquigarrow^*_{\text{STAM}} q$ a Source TAM run and $q \rightsquigarrow_a q'$.*

(1) *if $a = \bullet\beta_v$ then $|q'|_{oh} \leq |q|_{oh} + |t|$;*
(2) *if $a \in \{\circ sub, \circ sea_{1-5}, \bullet sea_6\}$ then $|q'|_{oh} < |q|_{oh}$;*
(3) *if $a \in \{\bullet sea_{1,3}, \bullet\pi\}$ then $|q'|_{oh} = |q|_{oh}$.*

Proposition 11.4 (Bilinear bound on the number of transitions). *Let $t \in \lambda_{\mathsf{sou}}$ be closed. If $r : t^\circ \rightsquigarrow^*_{\text{STAM}} q$ then $|r| \in O\big((|r|_{\beta_v} + 1) \cdot |t|\big)$.*

Tuples Raise the Overhead. Prop. 11.4 shows that projection transitions are also bi-linear. In Section 10, we mentioned that, without tuples, the bound improves to $O(|r|_{\beta_v})$ if one considers complete runs (that is, runs ending on values). With tuples, there is no such improvement. Indeed, even just checking that the initial term is actually a value v takes time $O(|v|)$ with tuples: if v is a tree of tuples, the Source TAM has to visit the tree and check that all the leaves are abstractions; in absence of tuples the check instead costs $O(1)$.

Step 2 of the Recipe: Cost of Single Transitions. To obtain fine bounds with respect to the initial term, we introduce the notions of width and height of a term t, both bounded by the size $|t|$ of t.

Definition 11.5 (Width, height). The *width* $\mathsf{wd}(t) \in \mathbb{N}$ of $t \in \lambda_{\mathsf{sou}}$ is the maximum length of a tuple or of a sequence of variables in t. The *height* $\mathsf{hg}(t) \in \mathbb{N}$ of $t \in \lambda_{\mathsf{sou}}$ is the maximum number of bound variables of t in the scope of which a sub-term of t is contained.

As discussed after Lemma 11.1, $\rightsquigarrow_{\circ sea_3}$ has to allocate a new tuple, thus its cost seems to depend on $\mathsf{wd}(t)$, and similarly for $\rightsquigarrow_{\bullet\beta_v}$. However, the price related to tuples can be considered as absorbed by the cost of search (by changing the multiplicative constant), since the new tuple of $\rightsquigarrow_{\circ sea_3}$ is then traversed, if the run is long enough, and the one of $\rightsquigarrow_{\bullet\beta_v}$ was traversed before the transition. Therefore, if we consider *complete* runs (i.e. ending on final states), $\rightsquigarrow_{\circ sea_3}$ and $\rightsquigarrow_{\bullet\beta_v}$ have *amortized* cost independent of $\mathsf{wd}(t)$.

Transitions $\bullet sea_1$, $\bullet sea_3$, and $\bullet sea_6$ duplicate E, the length of which is bound by $\mathsf{hg}(t)$. With flat environments, this costs $O(\mathsf{hg}(t))$. Transition $\rightsquigarrow_{\circ sub}$ has to look-up the environment, the cost of which is $O(\mathsf{hg}(t))$. De Bruijn indices or an ordered domain of names might improve the cost of look-up, but at no overall advantage, because of the dominating cost of duplicating environments for $\bullet sea_1$, $\bullet sea_3$, and $\bullet sea_6$. All other transitions have constant cost, assuming that accessing the i-th component of a tuple (needed for $\rightsquigarrow_{\bullet\pi}$) takes constant time. The next proposition sums it up.

Proposition 11.6 (Cost of single transitions). *Let $r : t^\circ \rightsquigarrow^*_{\text{STAM}} q$ be a complete Source TAM run. A transition $\rightsquigarrow_a$ of r has cost $O(\mathsf{hg}(t))$ if $a \in \{\bullet sea_1, \bullet sea_3, \bullet sea_6, \circ sub\}$, and $O(1)$ otherwise.*

Step 3 of the Recipe: Total Complexity. By simply multiplying the number of single transitions (Proposition 11.4) for their cost (Proposition 11.6), we obtain the complexity of the Source TAM.

Theorem 11.7. *Let $t \in \lambda_{\mathsf{sou}}$ be closed and $r : t^\circ \rightsquigarrow^*_{\text{STAM}} q$ be a complete Source TAM run. Then, r can be implemented on RAMs in time $O\big((|r|_{\beta_v} + 1) \cdot |t| \cdot \mathsf{hg}(t)\big)$.*

If one flattens $\mathsf{hg}(t)$ as $|t|$, the complexity of the Source TAM is $O\big((|r|_{\beta_v} + 1) \cdot |t|^2\big)$, that is, quadratic in $|t|$, while in absence of tuples—that is, for the LAM—it is linear in $|t|$.

12 Part 3: Complexity of the Target TAM, or, Closure Conversion Preserves the Overhead

In this section, we adapt the time complexity analysis of the Source TAM to an analysis of the Target TAM (skipping the less efficient Int TAM), and then connect source and target by considering the impact of closure conversion on the given analysis.

The Target TAM has a sub-term invariant, expressed compactly thanks to our flags. In particular the part about abstractions of the sub-term invariant for the Source TAM (Lemma 11.1) is here captured by having flagged the body of unevaluated closures with $\circ$.

Lemma 12.1 (Sub-term invariant). *Let q be a Target TAM reachable state from the initial state t°. Then u is a sub-term of t for every non-evaluated term $\circ u$ in q.*

Step 1: Number of (Overhead) Transitions. The bound on the number of (overhead) transitions is obtained following the same reasoning used for the Source TAM. The new transition $\rightsquigarrow_{\bullet sea_7}$ is part of

the transitions that are factored out, since each $\leadsto_{\bullet sea_7}$ transition is enabled by a $\leadsto_{\bullet\beta_v}$ transition, which adds an entry to the activation stack. We also use an overhead measure (in [10, Appendix J]) which is a direct adaptation to the Target TAM of the one given for the Source TAM. Note indeed that the measure ignores environments, which are the main difference between the two machines.

Proposition 12.2. *Let $t \in \lambda_{\mathtt{tar}}$ be closed. If $r : t^\circ \leadsto^*_{\mathrm{TTAM}} q$ then $|r| \in O\big((|r|_{\beta_v} + 1) \cdot |t|\big)$.*

Step 2: Cost of Single Transitions. For all transitions of the Target TAM but $\leadsto_{\circ sub_v}$, $\leadsto_{\circ sub_c}$, and $\leadsto_{\bullet sea_7}$ the cost is the same. For $\leadsto_{\circ sub_v}$, the cost is now $O(1)$ since tupled environments have $O(1)$ access time via indices (in the Int TAM its cost is instead $O(\mathtt{hg}(t))$).

For the new transition $\leadsto_{\circ sub_c}$, $O(\mathtt{wd}(t))$ look-ups in the environment are needed. Because of tupled environments, each look-up costs $O(1)$. Thus, the cost seems to depend on $\mathtt{wd}(t)$, but—reasoning as for $\leadsto_{\bullet\beta_v}$ and $\leadsto_{\circ sea_3}$ in Section 11—one can amortize it with the cost of search in complete runs. That is, we shall consider $\leadsto_{\circ sub_c}$ to have constant cost. The new transition $\leadsto_{\bullet sea_7}$ has constant cost as well. Summing up, we get one of the insights mentioned in the introduction: the amortized cost of all single transitions is $O(1)$.

Proposition 12.3 (Cost of single transitions). *Let $r : t^\circ \leadsto^*_{\mathrm{TTAM}} q$ be a complete Target TAM run. Every transition of r costs $O(1)$.*

Step 3: Total Complexity. Multiplying the number of single transitions (Prop. 12.2) for their cost (Prop. 12.3), we obtain the time complexity of the Target TAM, which seem better than for the Source TAM. After the theorem we discuss why it is not necessarily so.

Theorem 12.4. *Let $t \in \lambda_{\mathtt{tar}}$ be closed and $r : t^\circ \leadsto^*_{\mathrm{TTAM}} q$ be a complete Target TAM run. Then, r can be implemented on RAMs in time $O\big((|r|_{\beta_v} + 1) \cdot |t|\big)$.*

Factoring in the Size Growth of Wrapping. To complete the analysis, consider that the term $t \in \lambda_{\mathtt{tar}}$ on which the Target TAM is run is meant to be the closure conversion (that is, wrapping + name elimination) of a term $u \in \lambda_{\mathtt{sou}}$. While name elimination does not affect the size of terms, wrapping $\underline{\cdot}$ does (proof in [10, Appendix J]).

Lemma 12.5 (Wrapping size growth bound).

1. *If $t \in \lambda_{\mathtt{sou}}$ then $|\underline{t}| \in O(\mathtt{hg}(t) \cdot |t|)$.*
2. *There are families of terms $\{t_n\}_{n\in\mathbb{N}}$ for which $\mathtt{hg}(t_n) = \Theta(|t_n|)$, so that $|\underline{t_n}| \in \Theta(|t_n|^2)$.*

Proof. Point 1: the size increment in $\underline{t}$ is due to the bags introduced by wrapping abstractions, and it is proportional to the number of free variables in the body of the abstraction, bounded by $\mathtt{hg}(t)$; so $|\underline{t}| \in O(\mathtt{hg}(t)\cdot|t|)$. Point 2: take $t_n := \lambda x_1..\lambda x_n.x_1x_2\ldots x_n \neq \lambda x_1,..,x_n.x_1x_2\ldots x_n$ (note n abstractions in t_n, not just 1). □

We can now instantiate the bounds for running the Target TAM on the closure conversion of a $\lambda_{\mathtt{sou}}$ term, obtained by substituting the bounds in Lemma 12.5.2 in Theorem 12.4. We end up obtaining the same complexity as for the Source TAM (Theorem 11.7), despite the bound here being the outcome of a different reasoning.

Theorem 12.6. *Let $t \in \lambda_{\mathtt{sou}}$ be closed and $r : (\underline{t}^{\epsilon,\epsilon})^\circ \leadsto^*_{\mathrm{TTAM}} q$ be a complete Target TAM run. Then, r can be implemented on RAMs in time $O\big((|r|_{\beta_v} + 1) \cdot |t| \cdot \mathtt{hg}(t)\big)$.*

13 Related Work and Conclusions

Related work. The seminal work of Minamide et al. [28] uses existential types to type closure conversion, and other works study the effect of closure conversion on types, as well as conversion towards typed target languages [16, 18, 29]. This line of work explores the use of types in compilation which can span the entire compiler stack, from λ-calculus to (typed) assembly [30].

Appel and co-authors have studied closure conversion, its efficient variants, and its *space safety* [17, 31, 35, 36]. Unlike them, we address a direct style λ-calculus, without relying on a CPS. We are however inspired by [31] in studying flat environments, also known as *flat closures*. Optimizations of flat closures are studied in [25].

Closure conversion of only some abstractions, and of only some of their free variables is studied by Wand and Steckler [39]. Closure conversion has also been studied in relation to graphical languages [34], non-strict languages [37], formalizations [19, 40], extended to mutable state by Mates et al. [27] and to a type-preserving transformation for dependent types by Bowman and Ahmed [18].

Many of the cited works study various aspects of the efficiency of closure conversion. As said in the introduction, our concerns here are orthogonal, as we are rather interested in the asymptotic overhead of the machines with respect to flat closure conversion.

Sullivan et al. [38] study a call-by-push-value λ-calculus where converted and non-converted functions live together, considering also an abstract machine. They do not study the complexity of the machine, nor the new notions of environments studied here.

For abstract machines, we follow Accattoli and co-authors, see for instance [2, 5, 13]. Another framework is [21].

Work orthogonal to our concerns is the *derivation* of abstract machines using closure conversion as a step in the process, along with CPS transformation and defunctionalization [15].

Conclusions. We study the relationship between closure conversion and abstract machines in probably the simplest possible setting, an extension with tuples of Plotkin's untyped call-by-value λ-calculus, and with respect to the simple notion of flat environments. Our starting point is to decompose closure conversion in two subtransformations, dubbed *wrapping* and *name elimination*, turning the source calculus into a target calculus, via an intermediate one.

Each calculus is then paired with a variant of the *tupled abstract machine* (TAM). The Source TAM has machine-closures and local environments, while the Int TAM and the Target TAM have forms of converted closures. Moreover, they exploit the invariants enforced by the transformations, adopting new, better behaved forms of environments, namely *stackable* and *tupled environments*.

We give proof of correctness—under the form of termination-preserving strong bisimulations—for wrapping and name elimination, as well as implementation theorems for every machine with respect to its associated calculus. In particular, the proof technique for the correctness of closure conversion is new and simple.

Lastly, we study the time complexity of the abstract machines, showing that flat closure conversion reshuffles the costs, lowering the dependency on the initial term, while at the same time increasing the size of the initial term, ending up with the same complexity.

Various directions for future work are possible, e.g. extending the source calculus with recursive let and pattern matching; or refining the abstract machines by making explicit the sharing for tuples.

References

[1] Beniamino Accattoli. 2019. A Fresh Look at the lambda-Calculus (Invited Talk). In *Proceedings of the 4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. 1:1–1:20. doi:10.4230/LIPIcs.FSCD.2019.1

[2] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2014. Distilling abstract machines. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 363–376. doi:10.1145/2628136.2628154

[3] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2015. A Strong Distillery. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9458)*, Xinyu Feng and Sungwoo Park (Eds.). Springer, 231–250. doi:10.1007/978-3-319-26529-2_13

[4] Beniamino Accattoli and Bruno Barras. 2017. Environments and the complexity of abstract machines. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP 2017)*. 4–16. http://doi.acm.org/10.1145/3131851.3131855

[5] Beniamino Accattoli, Andrea Condoluci, Giulio Guerrieri, and Claudio Sacerdoti Coen. 2019. Crumbling Abstract Machines. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 4:1–4:15. doi:10.1145/3354166.3354169

[6] Beniamino Accattoli, Andrea Condoluci, and Claudio Sacerdoti Coen. 2021. Strong Call-by-Value is Reasonable, Implosively. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–14. doi:10.1109/LICS52264.2021.9470630

[7] Beniamino Accattoli and Ugo Dal Lago. 2016. (Leftmost-Outermost) Beta-Reduction is Invariant, Indeed. *Logical Methods in Computer Science* 12, 1 (2016). doi:10.2168/LMCS-12(1:4)2016

[8] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. 2021. The (In)Efficiency of Interaction. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–33. doi:10.1145/3434332

[9] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. 2022. Reasonable Space for the λ-Calculus, Logarithmically. In *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, Christel Baier and Dana Fisman (Eds.). ACM, 47:1–47:13. doi:10.1145/3531130.3533362

[10] Beniamino Accattoli, Dan Ghica, Giulio Guerrieri, Cláudio Belo Lourenço, and Claudio Sacerdoti Coen. 2025. Closure Conversion, Flat Environments, and the Complexity of Abstract Machines. arXiv:2507.15843 [cs.PL] https://arxiv.org/abs/2507.15843

[11] Beniamino Accattoli and Giulio Guerrieri. 2016. Open Call-by-Value. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 206–226. doi:10.1007/978-3-319-47958-3_12

[12] Beniamino Accattoli and Giulio Guerrieri. 2017. Implementing Open Call-by-Value. In *Proceedings of the 7th IPM International Conference on Fundamentals of Software Engineering (FSEN 2017), Tehran, Iran, April 26-28, 2017, Revised Selected Papers*. 1–19. doi:10.1007/978-3-319-68972-2_1

[13] Beniamino Accattoli and Giulio Guerrieri. 2019. Abstract machines for Open Call-by-Value. *Sci. Comput. Program.* 184 (2019). doi:10.1016/j.scico.2019.03.002

[14] Beniamino Accattoli and Claudio Sacerdoti Coen. 2017. On the value of variables. *Inf. Comput.* 255 (2017), 224–242. doi:10.1016/j.ic.2017.01.003

[15] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*. ACM, 8–19. doi:10.1145/888251.888254

[16] Amal Ahmed and Matthias Blume. 2008. Typed closure conversion preserves observational equivalence. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 157–168. doi:10.1145/1411204.1411227

[17] Andrew W. Appel and Trevor Jim. 1989. Continuation-Passing, Closure-Passing Style. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 293–302. doi:10.1145/75277.75303

[18] William J. Bowman and Amal Ahmed. 2018. Typed closure conversion for the calculus of constructions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 797–811. doi:10.1145/3192366.3192372

[19] Adam Chlipala. 2010. A verified compiler for an impure functional language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 93–106. doi:10.1145/1706299.1706312

[20] Andrea Condoluci, Beniamino Accattoli, and Claudio Sacerdoti Coen. 2019. Sharing Equality is Linear. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (PPDP 2019)*. 9:1–9:14. doi:10.1145/3354166.3354174

[21] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press. https://mitpress.mit.edu/9780262062756/semantics-engineering-with-plt-redex/

[22] Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, Martin Wirsing (Ed.). North-Holland, 193–222.

[23] Maribel Fernández and Nikolaos Siafakas. 2008. New Developments in Environment Machines. In *Proceedings of the 8th International Workshop on Reduction Strategies in Rewriting and Programming, WRS@RTA 2008, Hagenberg, Austria, July 14, 2008 (Electronic Notes in Theoretical Computer Science, Vol. 237)*, Aart Middeldorp (Ed.). Elsevier, 57–73. doi:10.1016/J.ENTCS.2009.03.035

[24] Benjamin Grégoire and Xavier Leroy. 2002. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 235–246. doi:10.1145/581478.581501

[25] Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. 2012. Optimizing closures in O(0) time. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme 2012, Copenhagen, Denmark, September 9-15, 2012*, Olivier Danvy (Ed.). ACM, 30–35. doi:10.1145/2661103.2661106

[26] Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language*. Technical report 117. INRIA. http://gallium.inria.fr/~xleroy/publi/ZINC.pdf

[27] Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under Control: Compositionally Correct Closure Conversion with Mutable State. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 16:1–16:15. doi:10.1145/3354166.3354181

[28] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 271–283. doi:10.1145/237721.237791

[29] J. Gregory Morrisett and Robert Harper. 1997. Typed Closure Conversion for Recursively-Defined Functions. In *Second Workshop on Higher-Order Operational Techniques in Semantics, HOOTS 1997, Stanford, CA, USA, December 8-12, 1997 (Electronic Notes in Theoretical Computer Science, Vol. 10)*, Andrew D. Gordon, Andrew M. Pitts, and Carolyn L. Talcott (Eds.). Elsevier, 230–241. doi:10.1016/S1571-0661(05)80702-9

[30] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.* 21, 3 (1999), 527–568. doi:10.1145/319301.319345

[31] Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proc. ACM Program. Lang.* 3, ICFP (2019), 83:1–83:29. doi:10.1145/3341687

[32] Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159. doi:10.1016/0304-3975(75)90017-1

[33] Claudio Sacerdoti Coen. 2025. Artifact for the Closure Conversion and Abstract Machines paper. https://github.com/sacerdot/closure-conversion-machine/releases/tag/PPDP25.

[34] Ralf Schweimeier and Alan Jeffrey. 1999. A Categorical and Graphical Treatment of Closure Conversion. In *Fifteenth Conference on Mathematical Foundations of Progamming Semantics, MFPS 1999, Tulane University, New Orleans, LA, USA, April 28 - May 1, 1999 (Electronic Notes in Theoretical Computer Science, Vol. 20)*, Stephen D. Brookes, Achim Jung, Michael W. Mislove, and Andre Scedrov (Eds.). Elsevier, 481–511. doi:10.1016/S1571-0661(04)80090-2

[35] Zhong Shao and Andrew W. Appel. 1994. Space-Efficient Closure Representations. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994*, Robert R. Kessler (Ed.). ACM, 150–161. doi:10.1145/182409.156783

[36] Zhong Shao and Andrew W. Appel. 2000. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 129–161. doi:10.1145/345099.345125

[37] Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. 2021. Strictly capturing non-strict closures. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2021, Virtual Event, Denmark, January 18-19, 2021*, Sam Lindley and Torben Æ. Mogensen (Eds.). ACM, 74–89. doi:10.1145/3441296.3441398

[38] Zachary J. Sullivan, Paul Downen, and Zena M. Ariola. 2023. Closure Conversion in Little Pieces. In *International Symposium on Principles and Practice of Declarative Programming, PPDP 2023, Lisboa, Portugal, October 22-23, 2023*, Santiago Escobar and Vasco T. Vasconcelos (Eds.). ACM, 10:1–10:13. doi:10.1145/3610612.3610622

[39] Mitchell Wand and Paul Steckler. 1994. Selective and Lightweight Closure Conversion. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 435–445. doi:10.1145/174675.178044

[40] Yuting Wang and Gopalan Nadathur. 2016. A Higher-Order Abstract Syntax Approach to Verified Transformations on Functional Programs. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 752–779. doi:10.1007/978-3-662-49498-1_29

[41] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. doi:10.1006/INCO.1994.1093

Shallowly Embedded Functions

Benedikt M. Rips
Radboud University
Institute for Computing and Information Sciences
Nijmegen, The Netherlands
benedikt.rips@ru.nl

Niek Janssen
Radboud University
Institute for Computing and Information Sciences
Nijmegen, The Netherlands
niek.janssen3@ru.nl

Mart Lubbers
Radboud University
Institute for Computing and Information Sciences
Nijmegen, The Netherlands
mart@cs.ru.nl

Pieter Koopman
Radboud University
Institute for Computing and Information Sciences
Nijmegen, The Netherlands
pieter@cs.ru.nl

Abstract

A domain-specific language, DSL is tailored to a specific application domain to facilitate the production and maintenance of programs. Functions add an important abstraction and repetition mechanism to DSLs, just as for any other programming language. For the evaluation of embedded DSLs one can use functions in the host language for this purpose. However, the automatic replacement of host function calls by their body is undesirable in other interpretations of the DSL, like pretty printing and code generation, especially for recursive DSL functions.

In this paper, we offer an overview of the options for defining functions in an embedded DSL, in particular tagless-final, or class-based shallow embedding style. These functions are type safe, require minimal syntactic overhead, and are suitable for multiple interpretations of terms in the DSL.

CCS Concepts

• **Software and its engineering → Domain specific languages**; **Automated static analysis**; **Functional languages**; **Recursion**.

ACM Reference Format:
Benedikt M. Rips, Niek Janssen, Mart Lubbers, and Pieter Koopman. 2025. Shallowly Embedded Functions. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Domain-Specific Languages, DSLs, help to develop and maintain software for particular application domains. Embedded DSLs, so-called eDSLs, are implemented as libraries in host programming languages. Preferably, an eDSL supports multiple interpretations, for example printing, optimization, or evaluation. It is desirable that the type safety of the host language is also available in the eDSL. Embedding saves us from making a standalone tool chain with a parser, type checker, etc. for the DSL, as they are inherited from the host language. Typically, the host language becomes a powerful generator of eDSL programs [Krishnamurthi 2001]. Though some trade-offs have to be made for the correct typing and sharing of identifiers such as function names and when using them in function application.

Functional programming languages have shown to be well-suited host languages for eDSLs. This has been recognized long ago by Hudak [1998] and its large number of successors. Moreover, recent extensions of functional programming languages increase their quality as host languages for eDSLs. For example, the ability to define infix operators, generalized algebraic data types, multi-parameter type (constructor) classes, and higher-order/rank polymorphic function and data types.

In this paper, we focus on the definition of functions in eDSLs. Basically, functions are handled as identifiers with a fancy type. In a naive implementation, identifiers are just a variable tag, either a constructor or a function, and some identifier. In that approach, the host language compiler cannot check that the variable is properly defined nor that the type is used correctly. On the other hand, if we represent them by functions or function arguments in the host language, the compiler can check variables. This is an old idea known as Higher-Order Abstract Syntax, HOAS [Pfenning and Elliott 1988]. We use this concept to define functions in DSLs and thus reducing syntactic overhead and improving type safety. We parameterize the type of DSL expressions to mirror the type of the equivalent host language expression. Hereby, the host language's type system not only imposes type safety onto DSL expressions but also gives us type inference.

1.1 Requirements for DSL Functions

We would like embedded functions to fulfil certain requirements. First, we want to be able to control the types of the bound expressions if necessary, even for polymorphic, overloaded, and higher-order functions. Also, we would like to have the possibility to set bounds on the number of arguments of the function. This comes in handy when e.g. a compiler of the eDSL needs to store thunks in some finite memory location or to disallow partial function applications. Similarly, we want to choose in the eDSL design between first-order and higher-order functions. Second, function application

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

in the DSL should naturally resemble function application in the host language. It preferably does not require an explicit application construct. That is, we prefer to write `f x` over `ap f x` in our eDSL to minimize the semantic friction and syntactic overhead without losing control over the definition and callsites. Third, expressions shall be sharable in a manner that is accessible to the interpretations such that the implementation of sharing can be specific to the interpretation at hand. Fourth, embedded functions should be able to call themselves and each other. We require at least recursion, but preferably mutual recursion. Finally, we prefer to have single abstraction for binding, sharing, *and* recursion to make DSL expressions concise and uniform.

1.2 Research Contribution

Our work regards DSL binders represented in HOAS. The overall contribution is fourfold.

First, we present a DSL binder that is capable of defining functions of arbitrary arity (Section 3.2). We do so by providing a binder that is polymorphic in the type of its bound expression while still allowing DSL-level functions only and suitable type class instances for the arrow type. For the sake of completeness, we put our ideas in the context of the well-known technique for DSL binders for functions of fixed arity.

Second, we discuss DSL binders for mutually-recursive expressions (Section 3.3). We start out with an illustration of a naive approach that does not require additional syntax but only enables exposure of one of the multiple mutually-recursive expressions. To remedy this disadvantage, we extend the existing binders to tuples such that multiple expressions are bound simultaneously.

Third, we demonstrate a technique to define standalone DSL functions (Section 4). That is, instead of writing a closed DSL expression in which every DSL function is defined locally, we want to write a standalone DSL function that may be reused across multiple DSL expressions. This technique enables code deduplication and the implementation of libraries containing reusable DSL functions.

Last, to round things up, we list auxiliary techniques of which some are known from folklore and come in handy for certain DSLs. This includes type annotations for DSL expressions, named function arguments through records, and the ability to prohibit nesting of binders (Section 5).

Most code is written in the pure functional programming language Clean [Brus et al. 1987; Plasmeijer et al. 2021].[1] Yet, it can be translated to Haskell in a straightforward manner. In Section 3.2.3 and 5.3, we used Haskell instead of Clean due to some language features that are available in GHC but not in Clean.

2 The Example DSL

We demonstrate our ideas with an example DSL that is iteratively enriched by new constructs. The DSL contains basic functionality for lifting of values, arithmetics, boolean operations, comparisons, and conditionals.

To implement such a DSL, there are two main flavours of embedding techniques, namely deep embedding and shallow embedding. They are closely related and can be transformed isomorphically into each other [Atkey et al. 2009; Gibbons and Wu 2014]. In deep embedding, DSL constructs are represented as data types in the host language and interpretations are functions over these data types. In shallow embedding, it is exactly reversed: interpretations on the language are data types and language constructs are functions over these data types.

This paper focusses on a class-based shallow embedding. However, in the source code artefact, we also provide a deep embedded variant using generalized ADTs [Peyton Jones et al. 2006].

We start with classes for basic operators. As in any class-based embedding, the class variable `v` indicates the *interpretation* of the DSL. The argument of `v` indicates the type of the DSL expression, like in a GADT. Infix operators are used with the usual binding power and associativity to beautify the syntax. We add a dot to the operator name whenever required to avoid name clashes with the host language.[2]

```
class lit v :: a → v a | toString a
class arith v where
  (+.) infixl 6 :: (v a) (v a) → v a | + a
  (-.) infixl 6 :: (v a) (v a) → v a | - a
  (*.) infixl 7 :: (v a) (v a) → v a | * a
  (/.) infixl 7 :: (v a) (v a) → v a | / a
class bool v where
  (&&.) infixr 3 :: (v Bool) (v Bool) → v Bool
  (||.) infixr 3 :: (v Bool) (v Bool) → v Bool
class comp v where
  (==.) infix 4 :: (v a) (v a) → v Bool | == a
  (<.)  infix 4 :: (v a) (v a) → v Bool | < a
class If v :: (v Bool) (v a) (v a) → v a
```

The class variable `v` and its argument are also used to make the embedding sound. The latter guarantees that the type of DSL operators and its arguments match, e.g. that a conjunction is only used on Booleans. By making the type ascribed to a DSL term polymorphic in the interpretation `v`, one prevents inspection of terms, hereby ruling out exotic terms.

For code deduplication, sharing, and recursion, it seems natural to use the host language's recursive let binder. Yet, since it sits on the host level, we only reap its benefits during evaluation. For interpretations like printing, whose output leaves the host language, the bound expressions are not shared, and recursive expressions would lead to an infinite output. Thus, we add DSL-level binders to delegate sharing and fixation to the interpretation at hand [Oliveira and Löh 2013]. Specifically, we add a `cons` binder for constants and a `fun1` binder for single argument functions.

```
class cons v ::  ((v a)          → In  (v a)          (v b)) → v b
class fun1 v :: (((v a) → v b) → In ((v a) → v b) (v c)) → v c

:: In a b = (In) infix 0 a b // a prettier tuple
```

The `fun1` binder enables defining DSL unary functions. Compared to HOAS, in which one typically has functions of type `v (a → b)`, in our approach functions have the type `(v a) → v b`, hence have the advantage that we need neither an explicit apply nor an explicit variable construct. The `fun1` binder also permits recursion due to the function name being in scope inside its body. While it is possible to implement recursion with a fixed-point combinator [Barendregt

[1] All code is available as an artefact [Lubbers et al. 2023]. A concise guide to Clean for Haskell programmers is found in [Lubbers and Achten 2024].

[2] Functions in Clean have a fixed arity, so arguments are parenthesized when needed [Plasmeijer et al. 2021, §3.7].

2012], we chose a different approach here since we want our binder to support binding, sharing, *and* recursion, as laid out in Section 1.1.

The following definition of the factorial function `ex1_v2` serves as an example for a recursive function.[3] There is no higher meaning to the trivial definition of `one`, it simply serves as an illustration of the `cons` function. The argument of `ex1_v2` is the value of the argument of the factorial function, i.e. the value is inlined. This illustrates how the host language and the eDSL mix. For recursive functions, like `fac`, it is essential that we combine the function definition and its application in a single construct. The defined function is in scope of its own body as well as its applications.

```
ex1_v2 n =
  cons λone = lit 1 In
  fun1 λfac = (λn. If (n ==. one) one (n *. fac (n -. one))) In
  fac (lit n)
```

Currently, the `fun1` construct allows first-order functions with one argument only. In Section 3, we get rid of this limitation through a generalization over the type of the bound expressions.

2.1 Evaluation

Evaluation of DSL expressions takes place in a strict identity monad `E` and is completely standard. The definition of `E` and its relevant class instances are listed in Appendix A.

For the evaluation of the `cons` and `fun1` class, we uniformly substitute the term's body for all applications using a cyclic definition. The effectiveness of this technique depends crucially on lazy evaluation.

```
instance lit E where lit a = pure a
instance arith E where
  (+.) x y = (+) <$> x <*> y
  (-.) x y = (-) <$> x <*> y
  ...

instance cons E where cons f = let (val In body) = f val in body
instance fun1 E where fun1 f = let (val In body) = f val in body
```

In shallow embedding, the interpretation is embedded in the data type. Hence, evaluating expressions in our DSL is a matter of unpacking the data type `E`, as seen in `eval`.

```
eval :: (E a) → a
eval (E a) = a
```

For example, the evaluation `eval (ex1_v2 4)` of a call to the factorial function produces the value 24.

2.2 Printing

The printing interpretation for this DSL is slightly more complex. We use the tooling shown in Appendix B that uses a reader writer state monad, called `RWS`. Here, the log of the writer is a list of strings, the state contains fresh identifiers and the environment of the reader is not used. To generate concise prints we print literals without `lit` keyword and omit the dot suffix of infix operators. For brevity in presentation, we do not attempt to print minimal parentheses nor strive for an efficient implementation. Printing of the other basic classes is similar and shown in Appendix B.

[3] In Clean, →, = and . can be used for lambdas, we pick according to our taste.

```
instance lit Print where lit a = P (tell [toString a])
instance arith Print where
  (+.) x y = printBin x y "+"
  (-.) x y = printBin x y "-"
  ...
printBin x y op = P (tell ["("] ≫| runPrint x ≫| tell [op]
                                ≫| runPrint y ≫| tell [")"])
```

To print the binders, we use an idea similar to the previous section. For every bound expression, we generate a fresh identifier.[4] For brevity, we reuse the printing instance for the more general binders that we introduce in Section 3 and 3.2 below. In addition, it shows that this is indeed a special case of the more general binders.[5]

```
instance cons Print where cons f = def  f
instance fun1 Print where fun1 f = funa f
```

Evaluating `printMain (ex1_v2 4)` to print an application of the factorial function produces:

```
def v0 = 1 In
def v1 = λv2 → (If (v2==.v0)
    v0
    (v2*(v1 (v2-v0)))) In
(v1 4)
```

Note that, although we use a single `cons` and `fun1` definition in this example, these constructs can be nested arbitrarily. Each of these binders yields a valid value of type `v a`.

3 Handling Arity and Recursion

We now extend the above binders in two directions: the arity of functions and mutual recursion. Regarding arity, we demonstrate a technique that limits the functions to particular arities and another technique that enables functions of arbitrary arity.

3.1 Fixed Arity

The `cons` and `fun1` binders in the DSL above have no argument or a single argument respectively. We generalize both binders to a single unified binder by replacing the argument type `v a` of `fun1` by a more general `a` type variable which is exposed as a type class argument.

```
class funa a v :: ((a → v b) → In (a → (v b)) (v c)) → v c
```

This allows us to make instances of this class for tuples of varying size to resemble functions of varying arity. By making a `funa () v` instance for the unit type, we enable C-style functions without arguments. Note that this does not imply that tuples become part of the types of the DSL. The tuples sit in the host language and are just a way to denote multiple arguments in the DSL. This technique is useful to limit the number of arguments.

Consider for example the Ackermann function, a function with two arguments. It is defined as `ex2`.

```
ex2 = cons λzero = lit 0 In
      cons λone  = lit 1 In
      funa λack = (λ(m,n)→
```

[4] Unfortunately, the names given by the user of the DSL are not accessible without template metaprogramming, see also Section 4.

[5] The only consequence is that the printed name of all definitions becomes the name of the most general case. We do not consider that a problem since, whenever required, we make this name an additional parameter of a helper function that is called in the actual print class.

```
        If (m ==. zero) (n +. one)
        (If (n ==. zero) (ack (m -. one, one))
          (ack (m -. one, ack (m, n -. one))))) In
    ack (lit 2, lit 2)
```

The downside of this approach is that one needs to list `funa a v` instances for all types `a` that are bound inside the expression. To reduce the verbosity, you may use the quantified constraints [Bottu et al. 2017] type system extension to aggregate these constraints.

3.1.1 Evaluation. Evaluation is identical to the instances shown above. We implement it in terms of the most general binder `def` (Section 3.2) as it is just a specialization.

```
instance funa a E where funa f = def f
```

Evaluating the Ackermann program shown above produces 7 as expected.

3.1.2 Printing. Again, printing the binder is more work. The instance for `(Print a) Print` is equal to the `fun1 Print` instance. In the same style, we make an instance for `() Print` to allow functions with zero arguments, or a unit argument to be more precise. We showcase the instance for a pair, as used in the example `ex2` above. The difference to the single argument function is that we generate two symbolic arguments for the body and handle a tuple of arguments in applications.

```
instance funa (Print a,Print b) Print where
  funa f = P (fresh >>= λv → fresh >>= λa → fresh >>= λb →
    let (body In main) =
        f (λ(c,d). P (tell ["(",v," ("] >>| runPrint c >>|
        tell [", "] >>| runPrint d >>| tell [")"])) in
    tell ["funa ",v," = \\(",a,",",b,") → "] >>|
    runPrint (body (P (tell [a]),P (tell [b]))) >>|
    tell [" In λn"] >>| runPrint main)
```

This prints our Ackermann function example `ex2` as:

```
def v0 = 0 In
def v1 = 1 In
def v2 = λ(v3,v4) → (If (v3==.v0)
    (v4+v1)
    (If (v4==.v0)
      (v2 ((v3-v1), v1))
      (v2 ((v3-v1), (v2 (v3, (v4-v1))))))) In
(v2 (2, 2))
```

3.2 Arbitrary Arity

By yet another generalization, we define a binder for functions of arbitrary arity that can also be curried, i.e. there is no need to pack the arguments in a tuple. Furthermore, this binder binds not only functions but also constants. Again, we replace `a → v b` in the definition by a more general `a` type variable and supply appropriate instances.

```
class def a v :: (a → In a (v b)) → v b
```

We illustrate the power of this approach by an implementation of the algorithm that computes powers in logarithmic time.

```
ex3 = def λone = lit 1 In
      def λtwo = lit 2 In
      def λodd = (λn.
          If (n ==. one) (lit True)
          (If (n <. one) (lit False) (odd (n -. two)))) In
      def λpow = (λx n.
          If (n ==. lit 0)
            one
            (If (odd n)
               (x *. pow x (n -. one))
               (def λy. pow x (n /. two) In y *. y))) In
      pow (lit 3) (lit 5)
```

The symbols `one` and `two` denote constants, `odd` is function of a single argument, and `pow` is a function of two arguments. Also observe that the functions are recursive.

The `def` construct is in principle also general enough to allow binding higher-order functions. However, their availability depends on the interpretation `v`. Evaluation for example handles higher-order functions without any additional machinery since they are a core feature of the host language. In contrast, the printing of higher-order functions remains future work.

3.2.1 Evaluation. The evaluation again is general for any definition of an arbitrary type `a`.

```
instance def a E where def f = let (body In exp) = f body in exp
```

Defining such an instance of this class allows more than we want. It works for any type `a`. By defining more specific instances for various instance of `a` instead of this very general instance, we control the allowed arguments in detail.

Evaluating expression `eval ex3` yields the desired value 243.

3.2.2 Printing. Printing of the `lit` and `arith` classes is already defined in Section 2.2. That implementation is also used with the more general definitions. Here we define the `Print` instance of our most general definition class `def`.

The class `def` works for definitions with an arbitrary number of arguments. In the applied function occurrences, we print a generated name for the functions as well as the actual arguments. To print the body of the expression, we supply functions that print variable names as arguments to the defined function. The helper class `defType` does exactly that for various types of arguments. The Boolean argument of `actArg` indicated whether a closing parenthesis is needed.

```
class defType a where
  actArg  :: Bool (Print c) → a
  formArg :: a → Print c
```

First, we generate a fresh name `n` for the definition. Next, the defining function `f` is applied to the function that prints the actual arguments. Initially, this prints just the name of the function by `P (tell [n])`. The `actArg` adds the actual arguments one by one. To print the function body, we supply the formal arguments one by one with `formArg`. Finally, we only have to add some bookkeeping code to announce that there is definition and print the definition to finally print the body.

```
instance def a Print | defType a where
  def f = P (fresh >>= λn.
    let (a In b) = f (actArg False (P (tell [n]))) in
    incr >>| tell ["def ",n," = "] >>| runPrint (formArg a) >>|
    tell [" In"] >>| decr >>| nl >>| runPrint b)
```

The basic case for the class `defType` covers the case that there are no more arguments. In this situation, the object to be printed has type `Print a`. For the actual argument `actArg`, we just run the

given printer p and add a closing parenthesis whenever the Boolean argument b indicates that this is required. For the formal argument formArg, we just run the given printer.[6]

```
instance defType (Print a) where
  actArg b p | b         = P (runPrint p ≫| tell [")"])
             | otherwise = P (runPrint p)
  formArg p = P (runPrint p)
```

The instance of defType for (Print a)→b handles the case for a single function argument. Recursive calls handle multiple arguments one by one. The instance for an actual argument takes that argument as argument and prints it after the accumulator f. The function paren prints an open parenthesis whenever needed and calls actArg recursively. For the formal argument, a fresh argument variable v is generated. The function yields a printer that produces the corresponding lambda definition and provides the printer P (tell [v]) as argument to the function.

```
instance defType ((Print a)→b) | defType b where
  actArg b f = λx.
    paren b (runPrint f ≫| tell [" "] ≫| runPrint x)
  formArg  f = P (fresh ≫= λv → tell ["\\",v," → "] ≫|
    runPrint (formArg (f (P (tell [v])))))

paren :: Bool (PrintM ()) → a | defType a
paren b f = actArg True (P (if b f (tell ["("] ≫| f)))
```

This prints our power function example ex3 as:

```
def v0 = 1 In
def v1 = 2 In
def v2 = λv3 → (If (v3==.v0)
    True
    (If (v3<.v0)
      False
      (v2 (v3-v1)))) In
def v4 = λv5 → λv6 → (If (v6==.0)
    v0
    (If (v2 v6)
      (v5*(v4 v5 (v6-v0)))
      def v7 = (v4 v5 (v6/v1)) In
      (v7*v7))) In
(v4 3 5)
```

3.2.3 Restricting the Bindable Expressions. While the liberal def allows binding functions of arbitrary arity, we have no guarantees that these functions are from the object language. So one can use it to define host language constants and functions like in the following example:

```
one = def λone = 1 In lit one
```

Instead, we would like to restrict the type of the bindable expressions to liftable types, i.e., types with a shape like v a, (v a) (v b) → v c, and ((v a) → v b) (v a) → v b. To do so, we use an empty type class with selected instances to enumerate all types that we consider lifted.[7]

[6]In both cases the body is P (runPrint p) instead of just p to make the required type transition from Print a to Print c.

[7]Since the type class that we want to define does not have any members and its instances are incoherent, an implementation in Clean would involve a significant amount of boilerplate code. Hence, we give a definition in Haskell.

```
class Lift v a
instance {-# INCOHERENT #-} (Lift v a, Lift v b) ⇒ Lift v (a → b)
instance {-# INCOHERENT #-} Lift v (v a)
```

Since the instance resolution only works as soon as the type variable v is instantiated, we get overlapping instances, regardless of any overlap pragmas. Hence, we have to mark these instances as incoherent and mention the recursive case first so that the instance resolution matches the arrow type whenever applicable.[8]

With small changes to the instances given above, we are also able to alter the set of lifted types. For example, to not allow any type but only integers and Booleans to be lifted, we would have to replace the Lift v (v a) instance by a Lift v (v Int) and a Lift v (v Bool) instance. Or, to disallow higher-order functions, the (Lift v a, Lift v b) ⇒ Lift v (a → b) instance would have to be replaced by a Lift v b ⇒ Lift v (v a → b) instance.

With this machinery in place, we restrict the bindable expressions of the def combinator by constraining the type of the bound expressions to the Lift type class.

```
class Def v a where
  def :: Lift v a ⇒ (a → a `In` v b) → v b
```

Note that this idea is that it does not require any changes on the use site because it is merely an artificial restriction onto the types. Furthermore, declaring types as lifted is as easy as adding another instance of the Lift class.

3.3 Mutual Recursion

With the current eDSL binders, we can define mutually-recursive expressions as long as we only need to expose one of them. The trick is to define the expression that shall not be exposed inside the body of the expression that shall be exposed. For example, the even function is defined mutually-recursively through the odd function.

```
isSevenEven =
    def λone = lit 1 In
    def λisZero = (==.) (lit 0) In
    def λisOne = (==.) one In
    def λeven =
            def λodd = λn. If (isZero n)
                    (lit False)
                    (If (isOne n) (lit True) (even (n -. one))
            In λn. If (isZero n)
                    (lit True)
                    (If (isOne n) (lit False) (odd (n -. one)))
    In even (lit 7)
```

Compilation of such expressions requires advanced techniques like lambda lifting or closure conversion due to the nested binders. Also, being able to expose only one of the mutually-recursive expressions is a significant drawback.

Hence, we demonstrate a technique that allows to expose multiple mutually-recursive expressions on the same level. The basic idea is to define not only one expression but a tuple of expressions, so that our previous example of the mutually-recursive even and odd functions becomes:

```
evenOdd =
    def λone = lit 1 In
```

[8]See §6.8.8.5 of the GHC documentation: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/instances.html.

```
def λisZero = (λn. n ==. lit 0) In
def λisOne = (==.) one In
def λ(odd,even) =
   ( λn. If (isZero n) (lit False)
            (If (isOne n) (lit True) (even (n -. one)))
   , λn. If (isZero n) (lit True)
            (If (isOne n) (lit False) (odd (n -. one)))
   )
In odd (lit 7)
```

Intuitively, we expect the evaluation of a tuple of expressions to be already handled by the generic `def a E` instance. Even with the type of bound expressions restricted using the `Lift` type class, we expect it to work by declaring pairs as lifted types through a simple `(Lift v a, Lift v b) ⇒ Lift v (a, b)` instance. However, since the `(_, _) = ···` binder is strict in the tuple constructor, naive evaluation of a pair of binders leads to an infinite recursion.[9] Hence, we need to perform lazy pattern matching on the tuple, either at use site or at fixation site. To not put any burden onto the user, we change the fixation site by providing an explicit evaluation instance for tuples.

```
instance def (a,b) E where
  def f = let (pair In exp) = f (fst pair, snd pair) in exp
```

To print mutual recursive definitions, a tuple which prints the names is given as argument to the defining function. The obtained bodies and the main expression are printed in order. We just have to add some text to distinguish the parts.

```
instance def (a,b) Print | defType a & defType b where
  def f = P (fresh >>= λn. fresh >>= λm.
    let ((e1a, e1b) In e2) = f
          ( actArg False (P (tell [n]))
          , actArg False (P (tell [m]))) in
    incr >>| tell ["def (",n,",",m,") ="] >>| incr >>| nl >>|
    tell ["("] >>| runPrint (formArg e1a) >>| nl >>|
    tell [","] >>| runPrint (formArg e1b) >>| tell [") In "] >>|
    decr >>| decr >>| nl >>| incr >>| runPrint e2 >>| decr)
```

While we covered mutual recursion for two expressions only, it should be noted that this approach scales to three or more mutually-recursive expressions analogously.

4 Standalone Functions

Embedded function definitions with multiple interpretations as introduced above work fine. By choosing the appropriate variant, we determine what we want to allow in the DSL. Whenever desired, we extend the DSL with new operators or data types as well as functions with a different number or type of arguments.

There are three drawbacks for this approach. First, it requires that the entire DSL program is defined as a single block of code. This is fine for small programs, but for large programs this can hamper the readability. Second, it seems to obstruct the reuse of code. Finally, carefully designed function names are lost in showing the function definitions.

It is possible to make reusable functions in the current framework. We make a global function definition for each embedded definition we want to reuse. We start by adding an argument for each other definition used, like definition `one` in the function `fac` below. Next, we add an argument for the recursive calls of the function itself.[10] This is exactly equal to the use of a fixed-point combinator in λ-calculus. Finally, we have the arguments of the embedded function, either as normal function arguments or as a lambda function.

The example `ex4` shows how this looks for a factorial definition.

```
one _ = lit 1
fac one f n = If (n ==. lit 0) one (n *. f (n -. one))

ex4 = def λoneF = one oneF In
      def λf = fac oneF f In f (lit 4)
```

This evaluates and prints as if we had the definitions inlined as above. This approach enables the reuse of the functions `one` and `fac`. An alternative approach is to define the functions as continuation passing style. This method and the method before result in very complex types become more complex and all library functions still need to be declared explicitly.

```
withOne f = def λone → lit 1 In f one
withFac f = withOne λone →
    def λfac → (λn. If (n ==. lit 0) one (n *. fac (n -. one)))
    In f fac

ex5 = withFac λfac → fac (lit 4)
```

4.1 Named Functions

A more radical approach uses only named definitions. These definitions have a user-defined `ID` that must be unique. This `ID` solves the lost names' problem in printing and is an identifier to spot whether we have encountered this definition before. The equivalent of the definitions from Section 3 becomes the classes `fun` for function with an arbitrary argument and `def` for constant definitions.

```
class fun a v :: ID (a→v b) → a→v b
class term  v :: ID (v b) → v b

:: ID :== String
```

To specify concise type class constraints, we gather all relevant type classes in `funDef`. The type parameters are the interpretation `v` and the argument type `a` in function definitions.

```
class funDef v a | lit, arith, bool, comp, If v
                 & term v & fun (v a) v
```

The `even` and `odd` example below shows that this allows mutual recursion without the need to define the functions simultaneously as in Section 3.3.

```
Zero :: (v Int) | lit, term v
Zero = term "zero" (lit 0)

One :: (v Int) | lit, term v
One = term "one" (lit 1)

even :: ((v Int)→v Bool) | funDef v Int
even = fun "even" λn. If (n ==. Zero) (lit True)  (odd  (n -. One))

odd :: ((v Int)→v Bool) | funDef v Int
odd  = fun "odd " λn. If (n ==. Zero) (lit False) (even (n -. One))
```

[9] To delve into the problem, transform the HOAS binder into an open recursion style and then expand its fixation by means of equational reasoning.

[10] Technically, this is not required for non-recursive definitions like constants. For uniformity and simplicity, we add this argument always in our examples.

4.2 Evaluation

The evaluation is again straightforward, we just replace each definition by the body. The `ID` is not needed in this interpretation and stripped.

```
instance fun a E where fun  i a = a
instance term  E where term i a = a
```

The expression `eval (even (lit 5))` evaluates to `False`.

4.3 Printing

For the printing interpretation, we have to work a little harder. Here, we require the full tooling introduced in Appendix B. The state contains a mapping from `ID` to output of type `[String]`. Each time we encounter a new definition, we check this mapping for occurrence of the `ID`. When we have seen the definition before, we just use the `ID` to indicate a call to this definition. When the definition is not known, we print it like before and store the output of the writer monad at the position of the `ID` in the mapping. After we are done with printing, we collect all definitions from the mapping with `printAll`. Like in Section 3, we make instances for functions with a single argument as well as for tuples containing multiple arguments.

For simplicity, we assume that functions are not nested. One can handle the printing of nested functions for instance like their code generation by lifting all functions to the top level (see Section 6). We reuse the class `defType` from Section 3.2.2 for the type specific printing details. We only show the instances for definitions without argument, functions with a single argument and a tuple as argument.

```
instance term Print where
  term name f = P (printDef "term" name f >>| tell [name])
instance fun (Print a) Print where
  fun name f = actArg False
    (P (printDef "fun" name (formArg f) >>| tell [name]))
instance fun (Print a, Print b) Print where
  fun name f = actArg False
    (P (printDef "fun" name (formArg f) >>| tell [name]))
```

Both instances use the same helper function to add a definition to the mapping when this is needed.

```
printDef :: String ID a → PrintM () | defType a
printDef kind name f = gets (λs → 'M'.get name s.defs)
  >>= λmd → case md of
    ?Just _ = pure () // definition found
    ?None   = censor (λ_→[]) (listen runDefinition) >>= λ(_,def)→
              modify (λs → {s & defs = 'M'.put name def s.defs})
where
  runDefinition = modify (λs→{s & defs = 'M'.put name [] s.defs})
    >>| enter name // enter to context
    >>| tell [kind," ",name," = "] >>| runPrint (formArg f)
    >>| leave
```

To add some resilience to the system, we could add the compiler-generated name of the current function to the `ID`. This yields unique names when the user of the DSL accidentally reuses an identifier, but generates ugly function names. If available, a template metaprogramming system such as Template Haskell [Sheard and Peyton Jones 2002] can be used to generate unique identifiers for functions. Printing our example `printAll (even (lit 5))` produces:

```
fun even = λv0 → (If (v0==.zero) True  (odd (v0-one)))
fun odd  = λv1 → (If (v1==.zero) False (even (v1-one)))
term one = 1
term zero = 0
main = (even 5)
```

It is no silver bullet because DSL functions can be constructed on the fly using the host language as a macro language [Krishnamurthi 2001]. For example, the `times` function below unrolls a multiplication function in a sequence of additions by using the host language. Using just the name, location or generated identifier per function in the host language is not enough. This is mitigated by incorporating the arguments in the identifier, as done below.

```
times x = def ("times" +++ toString x)
    λy → foldr (+.) (lit 0) (repeatn x y)
```

5 Auxiliary Techniques

5.1 Indicating Types

The host language compiler is able to derive types for almost all DSL expressions though there are some exceptions. For these exceptions or software engineering reasons, it is convenient to specify types inside our DSL [Pierce 2002, §11.4]. Our host language Clean has no syntax for adding type ascriptions to expressions.[11] With the definition of type witnesses and two combinators we mimic the effect for the monomorphic definitions in our DSL. The type witnesses are values inside our DSL with suggestive names. In these definitions, we hide on purpose that these are types in our DSL rather than plain values.

```
Bool :: v Bool | lit v
Bool = lit False

Int :: v Int | lit v
Int = lit 42
```

We define two combinators named `:::` and `→.` to ascribe the type. The first infix combinator is used to indicate that an expression `e` has type `T` as `e ::: T`. The second infix operator constructs function types as `(λx. x +. lit 1) ::: Int →. Int`.

```
(:::) infix 1 :: a a → a
(:::) a t = a

(→.) infixr 2 :: a b → a→b
(→.) a b = λa → b
```

This approach is very similar to the use of `Proxy` and type ascriptions. The advantage of our approach with type witnesses is that we never need to write `Proxy` in the types and are not building on language extensions. The disadvantage is that it must be possible to create a value of the type, something that is always the case, e.g. a value of the `World` [Backus et al. 1990] or the `Void` type. Though `undef` can be used as the value is never evaluated anyway.

In this approach these witnesses are never materialized in any interpretation of the DSL. The definition of `:::` discards them. An application is the factorial definition below. The examples show

[11] Something that is available in e.g. Haskell98 [Peyton Jones 2003, §3.16].

that we can type definitions with various arities as well as sub expressions.

```
ex1_typed n =
    def λone → lit 1 ::: Int In
    def λequ → (λx y → x ==. y) ::: Int →. Int →. Bool In
    def λfac → (λn. If (equ n one ::: Bool)
            one
            (n *. fac (n -. one) ::: Int)) ::: Int →. Int In
    fac (lit n) ::: Int
```

5.2 Named Arguments

Sometimes it is convenient to indicate function arguments by a name rather than by their position. For instance when a function has multiple arguments of the same type. We can facilitate named arguments in the DSL by using a record type of the host language.

For instance, we calculate the compound interest of some loan based on four real numbers.[12] The record `Loan` introduces names for these values. All values are `Real` numbers in our DSL. Hence, they are interpretations `v` on such a value. This interpretation `v` parameterizes the record.

```
:: Loan v = { sum  :: v Real // principal sum
            , rate :: v Real // nominal annual interest rate
            , freq :: v Real // compound frequency
            , time :: v Real // overall length of time
            }
```

Using this record we can use named function arguments and parameters as in the example `compoundInterest`. Note that the order of fields in the definition of the function `ci` differs from the argument `loan1`. Allowing this is exactly the purpose of the named arguments.

```
compoundInterest =
  def λci = (λ{sum, rate, freq, time} →
            sum *. (lit 1.0 +. rate /. freq) ^. (freq *. time)) In
  ci { freq = lit 4.0, time = lit 6.0
     , sum = lit 1500.0, rate = lit 4.3 /. lit 100.0}
```

Without any changes to the given DSL implementation, this evaluates to `1938.8`.

We need to add an instance of the class `defType` from Section 3.2.2 when we want to print DSL expressions with instances of this type. We need to add argument identifiers to record fields to cope with multiple arguments of type `Loan`. Hence, it is more concise to print only this identifier as the formal argument of DSL definitions. Since records and record updates are part of the host language rather than the DSL, the only safe option is to print each record field explicitly in every application. This becomes rather verbose. A direct implementation yields for our compound interest example:

```
def v0 = λv1 → (v1.sum*((1+(v1.rate/v1.freq))^(v1.freq*v1.time)))
    In (v0 { sum = 1500, rate = (4.3/100), freq = 4, time = 6 })
```

5.3 Prohibiting Nested Definitions

The DSL binders introduced above can be nested arbitrarily. It is convenient that arguments of a function are available in a nested definition. See for instance example `ex3` (Section 3.2). The arguments `x` and `n` of function `pow` are used in the body of `y`. However, for some DSLs this is undesirable. Nested definitions require special attention in code generation to make the implicit arguments reachable. Well-known solutions to handle nested definitions are closure conversion and lambda lifting.

Using a slightly different function definition class, we prevent nested definitions. The key is to wrap the expression in which the bound term is used in a data type. Here we use a record, but an algebraic data type works equally well. Also note that we provide Haskell code here because the `Lift` type class from Section 3.2.3 which is implemented in Haskell is used.

```
data Main v b = Main { runMain :: v b }
```

By embedding the main expression in a record, we ensure that the type system enforces that the DSL user only writes definitions at the top level. To avoid name clashes with the DSL binder `def` that was introduced earlier, the binder for top-level terms only is called `defM`.

```
class DefM a v where
    defM :: Lift v a ⇒ (a → a `In` Main v b) → Main v b
instance DefM a E where
    defM f = let (body `In` expr) = f body in expr
```

Obviously, enforcing only top-level terms works only in a DSL where the `def` binder is not available. We can still have multiple definitions, but only definitions at the outermost level. Any `defM` definition at a nested position causes the required type error. Note that for this to work, it is important to restrict the type `a` of bindable terms by the `Lift` type class as laid out in Section 3.2.3. Otherwise, nesting in the form of

```
defM λt1 → (defM λt2 → ··· `In` Main ···) `In` t2
```

is still possible. Apart from the wrapping `Main` type, `defM` definitions are identical to `def` definitions. Also, all instances are similar. To prevent that the user of the DSL fools the system, the `runMain` accessor is only available inside the implementation of the DSL.

Yet another variant of the factorial example, this time with top-level terms only, is given below.

```
facM n =
  defM λone → lit 1 `In`
  defM λis0 → (==.) (lit 0) `In`
  defM λfac → (λn. If (is0 n) one (n *. fac (n -. one))) `In`
  Main $ fac (lit n)
```

6 Code Generation for Nested Functions

To show that this approach is not limited to simpler interpretations, we introduce code generation as a new interpretation for the example DSL. For this illustration, we use the standalone function variant (Section 4). To handle nested functions, we adjust the function definition by adding arguments for the captured variables and all applications are extended with the captured variables. This obtains the same effect as lambda lifting [Johnsson 1984].

The generated code is represented by the algebraic data type `Instruction` containing instructions for a stack machine. There are two instructions that contain data only used during compilation. Firstly, `Marker` is used to place markers in the code, and allow easy later patching the code. Secondly, `Arg` contains two data fields; the scope, and the argument number. The scope number is only used

[12] See https://en.wikipedia.org/wiki/Compound_interest for an explanation.

internally to implement lambda lifting. Extensible ADTs [Plasmeijer et al. 2021, §5.1.5], data types *à la carte* [Swierstra 2008], or classy deep embedding [Lubbers 2022] can be used to hide these constructors.

```
:: Instruction = Push Int | Arg Int Int
  | Add | Sub | Mul | Div | Le | Eq | And | Or | Not | Neg
  | Lbl Int | Jmp Int | JmpF Int | Jsr Int | Ret Int | Halt
  | Marker Int
```

6.1 Compilation

The compilation of our DSL follows the same schema as printing and uses a reader writer state monad. The reader part is not used. The state contains a counter for fresh identifiers, a map from function identifiers to the code of the body, a map from function names to identifiers and a map containing the required metadata used when calling a function. During execution of the monad, instructions are emitted through the writer part of the monad.

```
:: Compile a = C (CompileM ())
:: CompileM a :== RWS () [Instruction] CompileState a
:: CompileState =
  { fresh     :: Int
  , functions :: Map Int [Instruction] // Maps labels to instructions
  , funmap    :: Map String Int        // Maps names to labels
  , funcalls  :: Map Int [Instruction] // Maps labels to function calls
  }

runCompile :: (Compile a) → CompileM ()
runCompile (C a) = a
```

The instances of the DSL components regarding expressions follow the pattern familiar from printing them. For simplicity, we assume that all basic data types are converted to an integer (using `toInt`) to keep the stack representation homogeneous. This is added as a class constraint to the `lit` function. Only in the conditional expression, the state is used to generate fresh labels in order to implement the jumps between conditional branches.

```
instance lit Compile where
  lit a = C (tell [Push (toInt a)])
instance arith Compile where
  (+.) x y = C (runCompile x >>| runCompile y >>| tell [Add])
  ...
instance bool Compile where ···
instance comp Compile where ···
instance If Compile where
  If c t e = C (
    fresh >>= λelselabel → fresh >>= λendiflabel →
    runCompile c >>| tell [JmpF elselabel] >>|
    runCompile t >>| tell [Jmp endiflabel, Lbl elselabel] >>|
    runCompile e >>| tell [Lbl endiflabel])
```

6.2 Functions

Functions are implemented in this compiler similar to the printer. We show the `fun` instance for single argument functions, the implementation for other arities is more of the same. In the body, the arity and the name of the function are passed to `compOrRetrFunction`. The third argument of this function is a function that, given a label, provides a representation of the arguments to the function definition. The `compOrRetrFunction` produces a label used for calling the function. If the function was already encountered, only the label is returned, if the function is seen for the first time, the definition is generated. Using the label, `callFunction` is called with the code to evaluate the arguments and the label.

```
instance term Compile where
  term name f = fun name (λ()→f) ()
instance fun (Compile a) Compile where
  fun name f = λx → C (compOrRetrFunction 1 name
        (λlbl → f (C (tell [Arg lbl 0])))
    >>= callFunction (runCompile x))
```

6.2.1 Generating the Definition. The `compOrRetrFunction` function first checks if the function has been encountered before by looking it up in the `funMap`. If this is the case, we return the label immediately. Otherwise, we generate a fresh label and store it with the name in the `funMap`. Then we execute the body using the provided function `f` while capturing the output using `censor` and `listen` — similarly to what was done in the printing interpretation. The instructions are placed after performing the lambda lifting. Finally, the instructions for the function call preparation is stored in the `funcalls` field. Later, the `Marker` referencing this label is replaced by this sequence of instructions.

```
compOrRetrFunction :: Int String (Int → Compile a) → CompileM Int
compOrRetrFunction arity n f =
  gets (λs → 'M'.get n s.funmap) >>= λv → case v of
    ?Just i = pure i
    ?None = fresh // Generate a fresh label
      >>= λlbl → modify (λs→{s & funmap='M'.put n lbl s.funmap})
      >>| censor (λ_ → []) (listen (runCompile (f lbl)))
      >>= λ(_, def)→let la = findLiftedArguments lbl def in
          modify (λs → {s & functions=
            'M'.put lbl (lift arity lbl def la) s.functions})
      >>| modify (λs → {s & funcalls =
            'M'.put lbl [Arg lbl i \\ (_, i) ← la] s.funcalls})
      >>| pure lbl
```

Lambda lifting finds the captured arguments of nested functions and adds them to the definition and applications. In general, lifting all functions to top-level definitions requires an intensional analysis of the call graph [Morazán and Schultz 2008]. For the sake of simplicity and brevity, this implementation only handles simple cases, one level deep, where an argument of an outer function is used in an inner function. Lifted arguments are identified by having a different label than the current function and assigned a number. Lifting them is done by replacing the `Arg` instructions by `Arg` instructions with the label of the current function and with a patched number.

```
findLiftedArguments :: Int [Instruction] → [((Int, Int), Int)]
findLiftedArguments lbl def = zip2 (sort (removeDup
  [(f, i) \\ (Arg f i) ← def | f ≠ lbl])) [0..]

lift :: Int Int [Instruction] [((Int, Int), Int)] → [Instruction]
lift arity lbl def la = map replaceLift def
  ++ [Ret (arity + length la)]
where replaceLift :: Instruction → Instruction
      replaceLift a=:(Arg f i) = case lookup (f, i) la of
        ?None   = a
        ?Just a = Arg lbl (arity + a)
```

```
    replaceLift i = i
```

6.2.2 Calling Functions. The second part of the `fun` implementation is generating the code for calling the function. In case of a recursive call, it is not yet known what data from the context, i.e. lifted arguments, need to be inserted as well. Therefore, a marker is included that represents the context, this is later be replaced by the code for the context. After pushing the marker, the arguments are evaluated and a `Jsr` instruction is written.

```
callFunction :: (CompileM ()) Int → CompileM ()
callFunction args i = tell [Marker i] ≫| args ≫| tell [Jsr i]
```

6.3 Running the Compiler

Compilation is a matter of running the monad stack. This results in code for the main expression in the writer output and the code and metadata for the functions in the resulting state. The main expression decorated with a `Halt` instruction is concatenated with functions decorated with labels. The resulting code still contains markers, they are then replaced by the corresponding instructions to push the lifted arguments.

```
compile :: (Compile a) → [Instruction]
compile (C f) = foldr replaceMarkers [] (main ++ [Halt
  : flatten [[Lbl l:is] \\ (l, is) ← 'M'.toList st.functions]])
where
  (st, main) = execRWS f ()
    { fresh = 0,           functions = 'M'.newMap
    , funmap = 'M'.newMap, funcalls = 'M'.newMap
    }
  replaceMarkers (Marker i) acc = 'M'.find i st.funcalls ++ acc
  replaceMarkers i acc = [i:acc]
```

Using the following nested definition, `llift (lit 40)` evaluates to 42.

```
llift = fun "plustwo" λx →
  let local = fun "fplus" λy → y +. x
  in  local (lit 2)
```

Compiling this code produces the instructions below. Function label `0` is the `plustwo` function, label 1 is the `fplus` function. Before jumping to `fplus`, `Arg 1 0`, the context, is pushed, followed by the argument `Push 2`.

```
   Push 40
   Jsr 0
   Halt
0: Arg 1 0
   Push 2
   Jsr 1
   Ret 1
1: Arg 1 0
   Arg 1 1
   Add
   Ret 2
```

7 Related Work

The idea of embedding DSLs is due to Landin [1966]. Hudak [1998] introduced the notion of embedded modular DSLs in functional programming. Pfenning and Elliott [1988] introduce, with HOAS, a simply typed λ-calculus enriched with products and polymorphism to express syntax trees. This in turn is an extension of the second-order term language of Huet and Lang [1978].

Using a type parameter is known from Parametric HOAS [Chlipala 2008] and Boxes Go Bananas [Washburn and Weirich 2003]. While those techniques use the type parameter to prevent exotic terms, we use the type parameter to indicate the type of the DSL construct. This leverages the host language's type checker to rule out undesirable terms and to infer the type. In that sense, it is similar to the type parameter in GADTs. Yet, in contrast to a deep based embedding, our shallow embedding is simpler in that it omits some quantifiers.

A consequence of defining identifiers as function arguments and imposing the normal type constraints is that we can define polymorphic functions in the DSL, but we can only use them with a single type instance. Serrano et al. [2020] introduce a type extension to allow the polymorphic use of such a definition at the cost of an explicit type definition. In Koopman and Lubbers [2023] we show that we can achieve the same effect at the cost of an additional constructor in function definitions and hence an explicit function application operator. Oliveira and Löh [2013] introduces techniques for expression sharing and recursion that we reuse here.

Atkey et al. [2009]; Matsuda et al. [2023] converts the finally-tagless representation of lambda-terms to de Bruijn-indexed terms to fix the mismatch between host and guest language function semantics. This allows open terms as well other interpretations of function application. Our DSL functions are well-typed, implying that undesired terms are prevented, and can be recursive. By selecting the desired version of the definition, we can impose additional constraints, like a controlled number of arguments or first-order functions.

The original HOAS idea based on ADTs suffers from problems with exotic or junk terms [Chlipala 2008]. Our class-based HOAS approach exposes the issue if the interpretation is known since it allows the user to exploit internals of the interpretation. Making an expression parametric in the interpretation, i.e. quantifying over the interpretation `v`, resolves it.

Intentional analysis, like program optimization and partial evaluation, is generally hard with HOAS and shallow embeddings because we cannot inspect functions. This is solved for HOAS [Atkey et al. 2009] and class-based shallow embeddings [Carette et al. 2009; Kiselyov 2012] by converting to-and-from a deep representation or by using an intermediate data type capturing just enough state. Gill [2009] introduced an IO-based solution to observable sharing that uses type functions to provide type-safe observable sharing. McDonell et al. [2013] expand on this for a typed syntax tree.

Carette et al. [2009]; Kiselyov [2012] use λ-calculus with a fixed-point combinator instead of direct recursion. In this paper, we show that we can use also a fixed-point combinator, but that direct recursion is more elegant and direct. The direct approach for evaluation is similar to Fegaras and Sheard [1996] for a DSL that is always evaluated. Since we control what is done with applied function occurrences, we have fewer problems with intentional analysis and can easily make interpretations of the DSL like code generators and pretty printers.

Kiselyov [2024] explicates the challenges in generating safe low-level code. These techniques should integrate with our approach.

Every class constraint needed in some interpretations is part of the type classes constructing our DSL. Jones et al. [2012] shows how these constraints can be moved to the interpretations that require them.

Naming components of the language is not new. For example, Frost et al. [2008] used this technique to detect left recursion in parser combinators. We show that it can be used for DSL functions as well that can be recursive themselves.

Implementing named arguments by means of a record, as described in Section 5.2 has been discussed extensively in white and grey literature. To the best of our knowledge, it has never been published in a DSL context before, although it is known from folklore.

8 Conclusion

This paper shows that it is possible to define strongly-typed functions as part of an embedded DSL with multiple interpretations in a strongly-typed host language. Key to these type-safe definitions are two observations. First, the type parameter of DSL expressions mirrors its type. Second, identifiers for functions are provided in HOAS style, hence giving the identifier itself a type. The type checker of the host language then unifies the identifier's type with the type of the expression that the identifier is bound to. The only syntactical overhead is a DSL binding construct and a lambda for defining the identifier.

To allow multiple interpretations of the DSL we use a class-based embedding. Each interpretation of the DSL is an instance of the classes constructing the DSL. We demonstrate evaluation, pretty printing, and code generation of nested functions using lambda lifting for a first-order functional DSL.

We presented gradual additions to the binder to allow more general types of functions. In the first iteration, functions accept exactly one argument that is a single DSL value. In the second iteration, functions accept multiple arguments, encoded as a tuple of DSL values. This binder also showed how to restrict the arity of functions. In the third iteration, the most general case, DSL functions support any number of arguments, just like functions in the host language. In addition, we demonstrated a technique to restrict the types allowed in a DSL by enumerating these types through an empty type class. Hereby, the liberality of the DSL binder is decoupled from a restriction on the DSL types.

Besides enabling functions of arbitrary arity, our binder also handles recursive functions, or, more general, recursive terms. Furthermore, we illustrated a technique to extend the binder to mutually-recursive terms: Instead of binding a single expression only, we bind a tuple of expressions simultaneously.

These ideas were complemented with an approach for defining standalone functions to facilitate reuse. This enables us to implement libraries of DSL expressions.

Like with many eDSLs, the type errors that a user faces are difficult to comprehend. They are intertwined with the machinery which is not meant to be shown to the user, like e.g. the type classes for the restriction of eDSL types and the printing of arbitrary arity functions. There are well-known techniques to improve error messages of eDSLs [Serrano 2018; Serrano and Hage 2017].

The printing of arbitrary arity functions shows another problem: To pattern-match on arrow types, we need another type class that is not related to the DSL itself. This implies that, in order to print expressions, printing-specific type class constraints have to be listed in the expression's typing context. Yet, the nature of the printing implementation suggests that it underlies a generic concept that remains to be uncovered in forthcoming work.

An interesting property of the DSL binder is that, from the perspective of the type system, it allows higher-order functions. However, we do not know of an implementation of higher-order functions for non-trivial interpretations. Investigating their implementation seems like another exciting opportunity for future work.

References

Robert Atkey, Sam Lindley, and Jeremy Yallop. 2009. Unembedding Domain-Specific Languages. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, New York, NY, USA, 37–48. doi:10.1145/1596638.1596644 event-place: Edinburgh, Scotland.

John Backus, John H. Williams, and Edward L. Wimmers. 1990. An Introduction to the Programming Language FL. In *Research Topics in Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., USA, 219–247.

Henk P. Barendregt. 2012. *The Lambda Calculus Its Syntax and Semantics*. Number 40 in Studies in Logic. College Publ, London.

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified class constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. Association for Computing Machinery, New York, NY, USA, 148–161. doi:10.1145/3122955.3122967 event-place: Oxford, UK.

T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. 1987. Clean — A language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–384.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (sep 2009), 509–543. doi:10.1017/S0956796809007205

Adam Chlipala. 2008. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, New York, NY, USA, 143–156. doi:10.1145/1411204.1411226 event-place: Victoria, BC, Canada.

Leonidas Fegaras and Tim Sheard. 1996. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 284–294. doi:10.1145/237721.237792

Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. 2008. Parser Combinators for Ambiguous Left-Recursive Grammars. In *Practical Aspects of Declarative Languages*, Paul Hudak and David S. Warren (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–181.

Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 339–347. doi:10.1145/2628136.2628138 event-place: Gothenburg, Sweden.

Andy Gill. 2009. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. Association for Computing Machinery, New York, NY, USA, 117–128. doi:10.1145/1596638.1596653 event-place: Edinburgh, Scotland.

Paul Hudak. 1998. Modular Domain Specific Languages and Tools. In *Proceedings of the 5th International Conference on Software Reuse (ICSR '98)*. IEEE Computer Society, USA, 134. doi:10.1109/ICSR.1998.685738

Gérard P. Huet and Bernard Lang. 1978. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Informatica* 11 (1978), 31–55. doi:10.1007/BF00264598

Thomas Johnsson. 1984. Efficient Compilation of Lazy Evaluation. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction* (Montreal, Canada) *(SIGPLAN '84)*. Association for Computing Machinery, New York, NY, USA, 58–69. doi:10.1145/502874.502880

Will Jones, Tony Field, and Tristan Allwood. 2012. Deconstraining DSLs. *ACM SIGPLAN Notices* 47, 9 (Oct. 2012), 299–310. doi:10.1145/2398856.2364571

Oleg Kiselyov. 2012. Typed Tagless Final Interpreters. In *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, Jeremy Gibbons (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 130–174. doi:10.1007/978-3-642-32202-0_3

Oleg Kiselyov. 2024. Generating C: Heterogeneous metaprogramming system description. *Science of Computer Programming* 231 (2024), 103015. doi:10.1016/j.scico.2023.

103015
Pieter Koopman and Mart Lubbers. 2023. Strongly-Typed Multi-View Stack-Based Computations. In *Proceedings of the 25th International Symposium on Principles and Practice of Declarative Programming (PPDP '23)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3610612.3610623 event-place: Lisboa, Portugal.
Shriram Krishnamurthi. 2001. *Linguistic reuse.* PhD Thesis. Rice University, Houston, USA.
Peter J. Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (March 1966), 157–166. doi:10.1145/365230.365257
Mart Lubbers. 2022. Deep Embedding with Class. In *Trends in Functional Programming*, Wouter Swierstra and Nicolas Wu (Eds.). Springer International Publishing, Cham, 39–58. doi:10.1007/978-3-031-21314-4_3
Mart Lubbers and Peter Achten. 2024. Clean for Haskell Programmers. arXiv:2411.00037 [cs.PL] https://arxiv.org/abs/2411.00037
Mart Lubbers, Pieter Koopman, and Niek Janssen. 2023. Source code for the paper Shallowly Embedded Functions. doi:10.5281/zenodo.10225278
Kazutaka Matsuda, Samantha Frohlich, Meng Wang, and Nicolas Wu. 2023. Embedding by Unembedding. *Proc. ACM Program. Lang.* 7, ICFP (Aug. 2023). doi:10.1145/3607830 Place: New York, NY, USA Publisher: Association for Computing Machinery.
Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 49–60. doi:10.1145/2500365.2500595 event-place: Boston, Massachusetts, USA.
Marco T. Morazán and Ulrik P. Schultz. 2008. Optimal Lambda Lifting in Quadratic Time. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*. Springer-Verlag, Berlin, Heidelberg, 37–56.
Bruno C. d. S. Oliveira and Andres Löh. 2013. Abstract syntax graphs for domain specific languages. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation* (Rome, Italy) *(PEPM '13)*. Association for Computing Machinery, New York, NY, USA, 87–96. doi:10.1145/2426890.2426909
Simon Peyton Jones (Ed.). 2003. *Haskell 98 language and libraries: the revised report.* Cambridge University Press, Cambridge.
Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-Based Type Inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. Association for Computing Machinery, New York, NY, USA, 50–61. doi:10.1145/1159803.1159811
Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 199–208. doi:10.1145/53990.54010
Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press, Cambridge, Massachusetts.
Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. 2021. *Clean Language Report version 3.1.* Technical Report. Institute for Computing and Information Sciences, Nijmegen. 127 pages.
Alejandro Serrano. 2018. *Type Error Customization for Embedded Domain-Specific Languages.* Ph. D. Dissertation.
Alejandro Serrano and Jurriaan Hage. 2017. Type Error Customization in GHC: Controlling expression-level type errors by type-level programming. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages* (Bristol, United Kingdom) *(IFL '17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 15 pages. doi:10.1145/3205368.3205370
Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (Aug. 2020), 29 pages. doi:10.1145/3408971
Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. doi:10.1145/581690.581691 event-place: Pittsburgh, Pennsylvania.
Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436. doi:10.1017/S0956796808006758
Geoffrey Washburn and Stephanie Weirich. 2003. Boxes go bananas: encoding higher-order abstract syntax with parametric polymorphism. *SIGPLAN Not.* 38, 9 (Aug. 2003), 249–262. doi:10.1145/944746.944728

A Evaluation Tooling

Throughout this paper we use evaluators and printing interpretations of various eDSLs. We try to reuse the standard tooling, like instances for the monad classes, as much as useful. The required definitions for evaluation are given in this appendix. The next appendix contains the tooling for the print interpretations.

For the evaluation of DSL variants, we use the simplest type possible that generalizes to all monads.[13] There is no need to carry a state around since all variables are represented by functions or function arguments. We use the substitution mechanism of the host language to ensure type-safe and efficient replacement of variables by the appropriate value.

```
:: E a = E !a

instance pure E where pure a = E a
instance Monad E where bind (E a) f = f a
instance Functor E where fmap f (E a) = E (f a)
instance <*> E where (<*>) (E f) (E a) = E (f a)
```

The evaluation of the basic classes not listed in Section 2.1 are listed here. The instances for the class `bool` ensure that the second argument is only evaluated when that is necessarily.

```
instance bool E where
  (&&.) x y = x >>= λb → if b y (pure False)  // for lazy evaluation
  (||.) x y = x >>= λb → if b (pure True) y   // for lazy evaluation
instance comp E where
  (==.) x y = (==) <$> x <*> y
  (<.) x y = (<) <$> x <*> y
instance If E where If c t e = c >>= λb. if b t e
```

B Print Tooling

The print tooling is more sophisticated than the evaluation tooling. It is based on the reader writer state monad. The state `PS` is a record containing an integer `i` to generate fresh variables, a context that is a stack of function `ID`s, a mapping from `ID`s to output as a list of strings, and an indentation depth `ind`. Only the last DSL versions use all these fields. The writer monad is a list of strings, `[String]`, to denote the output of printing. The reader `PR` is not used and equals void, `()`.

```
:: PS =
  { i :: Int,                context :: [ID]
  , defs :: Map ID [String], ind :: Int}
:: PR :== ()
:: Print a = P (PrintM ())
:: PrintM a :== RWS PR [String] PS a
:: ID :== String
```

We use some convenience functions for this `RWS` monad. They are explained in context on their first use in this paper.

```
runPrint :: (Print a) → PrintM ()
runPrint (P a) = a

nl :: PrintM ()
nl = get >>= λs → tell ["\n" : ["  " \\ _ ← [1..s.ind]]]

incr :: PrintM ()
incr = modify λs → {s & ind = s.ind + 1}

decr :: PrintM ()
decr = modify λs → {s & ind = s.ind - 1}

fresh :: PrintM String
```

[13] In Clean, `!` indicates strictness, so this is the strict identity functor.

```
fresh = get >>= λs. put {s & i = s.i + 1}
  >>| pure ("v" + toString s.i)

printAll :: (Print a) → String
printAll (P f) = concat
  ('M'.foldrWithKey (λk v a. ["\n":v] ++ a) ["\n":main] st.defs)
where
  (st, main) = execRWS f ()
    {i=0, context=[], defs='M'.newMap, ind=0}

printMain :: (Print a) → String
printMain (P f) = concat main
where
  (st, main) = execRWS f ()
    {i=0, context=[], defs='M'.newMap, ind = 0}

prnt :: a → PrintM () | toString a
prnt s = tell [toString s]

show` :: (DSL a) → PrintM () | toString a
show` e = let (P p) = show1 e in p
```

The print instances for basic classes not listed in Section 2.2 are listed here.

```
instance bool Print where
  (&&.) x y = printBin x y "&&"
  (||.) x y = printBin x y "||"
instance comp Print where
  (==.) x y = printBin x y "=="
  (<.) x y  = printBin x y "<"
instance If Print where
  If c t e = P (tell ["(If "] >>| runPrint c  >>| tell [" "] >>|
    runPrint t >>| tell [" "] >>| runPrint e >>| tell [")"])
```

Operational Game Semantics for Generative Algebraic Effects and Handlers

Hamza Jaafar
Nantes Université
Nantes, France

Guilhem Jaber
Inria / Nantes Université
Nantes, France

Abstract

We present a *sound* operational game semantics model (*w.r.t.* contextual equivalence) of a typed language with algebraic effects and handlers and dynamic generation of effect instances *à la* [5, 11]. We exhibit the interactive aspect of effect propagation, and to address it precisely, we identify the adequate granularity of the operational semantics (*akin* to [15]) and the decomposition of normal forms into their interactive and their observational part. To account for this additional form of interaction, we extend the standard *pure* interaction interface of game semantics consisting of questions and answers with *effectful* moves that involve the propagation of effects and the yielding of *delimited* continuations. Finally, we extend the well-bracketed constraint on the behavior of the environment, from the use of continuations as is standard in game semantics, to fragments of captured delimited continuations.

CCS Concepts

• **Theory of computation → Denotational semantics**; *Control primitives*; **Operational semantics**.

Keywords

Algebraic Effects, Handlers, Game Semantics

ACM Reference Format:
Hamza Jaafar and Guilhem Jaber. 2025. Operational Game Semantics for Generative Algebraic Effects and Handlers. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Algebraic effects were originally introduced by Power and Plotkin [31, 32] to provide a direct denotational account for syntactic entities that *perform* effects such as **get**, **set** for global state, **raise** for exceptions, or **choose** for non-deterministic choice, *etc.*

The combination of algebraic operations with effect handlers [33] is a more recent and powerful abstraction for programming with effects. Effect handlers generalize exception handlers by giving access not only to the raised effect but also to its delimited continuation. This enables a modular and compositional approach to effectful programming, supports user-defined effects, and facilitates effect combination in a seamless way.

A central challenge with this abstraction lies in managing multiple occurrences of the same effect while preserving modularity. Two main approaches have been proposed: the *lexical approach* and the *generative approach*, as described in [11]. In the lexical approach [8, 9, 34], one introduces effect instances locally, and the handler responsible for these instances is determined statically, preventing instance leakage. This static discipline enforces a form of *effect safety*, ensuring that all performed effects are handled. In contrast, the generative approach provides the possibility of generating dynamically fresh instances of an effect [5], that can be referred by their names, and be passed around as values. While this increases flexibility, it introduces challenges such as aliasing between names of effect instances, that needs to be tamed in order to enforce effect safety via a type-and-effect-system [11].

In this paper, we adopt the generative approach, but allow for *unsafe interaction*: effect instances may remain unhandled, and aliasing is not prevented. Our goal is to model the interaction between a program and its environment in such a language. Trace models for higher-order languages capture program behavior as sequences of observable actions -calls and returns- where the arguments of these actions may themselves be (abstractions of) functions. We follow the methodology of *operational game semantics* (OGS) to define such a trace model.

The central idea of OGS is to represent programs as labelled transition systems (LTS), derived from the operational semantics of the language, in combination with a decomposition of normal forms into observable and interacting parts. This approach is particularly well suited to languages with generative effects, as demonstrated by Laird's seminal work on ML-style higher-order references [22]. OGS also naturally accommodates control operators such as `call/cc` or the μ-binder capturing the surrounding context of the $\lambda\mu$-calculus [18]. In the absence of such control features, it enforces a *well-bracketing* discipline on the interaction—matching calls with returns—as is standard in game semantics [2, 17].

To extend OGS to a language with algebraic effect handlers, we incorporate:

- actions representing the execution of an effect;
- a fine-grained representation of the control flow between the program and its environment, to account for the exchange of delimited continuations triggered by handled effects.

Our long-term objective is to obtain a fully abstract trace model, capturing contextual equivalence of our language. As a first step in this direction, we prove that trace equivalence of our model is *sound* with respect to contextual equivalence. Moreover, we show that a well-bracketing condition can be imposed on the environment's

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

interaction. The fact that enforcing well-bracketing does not endanger soundness might seem surprising. Indeed, control operators like `call/cc` capture the entire undelimited continuation of a computation, breaking well-bracketing by allowing non-local jumps. But delimited continuation operators—such as those used with algebraic effect handlers—capture only a portion of the continuation up to a delimiter, so that a relaxed notion of well-bracketing still holds, enforcing a stack discipline for such delimited continuation.

In the last part of the paper, we exhibit counterexamples to full abstraction: trace equivalence turns out to be too discriminating to characterize contextual equivalence.

To keep the model simple, we forbid the exchange of effect instances between the program and its environment. This avoids the need to dynamically track which effect instances have been disclosed—information that would otherwise be required for reasoning about interactions.

2 The language Λ_{eff}

We consider a fine-grained call-by-value programming language [26] with typed algebraic effects and handlers. The language is augmented with dynamical generation of effect instances, in the style of *Eff* [5] We stress that these instances are first-class values that can be exchanged in the same way exceptions or ML-style references, in say OCaml, could be shared. The *generative approach* coupled with the ability to exchange names introduces several challenges to the design of a strong static typing system that rules out unhandled performed effects [11]. We will not address any of these difficulties in the present work.

2.0.1 codata and names: We will call *data* all values of positive types that can be *fully observed*. In contrast, we will call *codata* any Λ_{eff}-expression that cannot be fully observed but can only be probed to extract partial observable information (*data*). These include *function-like* expressions: λ-abstractions and evaluation stacks, as well as *effectful* expressions: much like *λ-abstractions*, they can only be inspected by being put in a suitable computation context (a handler) that will trigger evaluation and potentially produce *data*.

Operational methods for proving equivalence of programs typically rely on syntactic *tokens* in order to observe and collect information about the behaviour of the environment. In the interactive setting of OGS, this is done through *names* which are semantic artefacts introduced to abstract the *codata* disclosed by the environment. These names are to be understood as nominal *free variables*.

In the presence of algebraic effects and handlers, a program not only gets to call functions of the environment, but it can also capture and use fragments of an environment evaluation stack. As it had already been observed in [7], whereas λ-abstractions belonging to the environment are just *free variables*, the language provides no syntactic equivalent for the environment's evaluation contexts, therefore it has to be extended to account for them.

In the following, we will consider from the start a language in which these semantic names have been reified as values, evaluation contexts or computations in the syntax. Further details about their role and the process by which they are generated will be however postponed to section 3.1.

Syntax of types and expressions

We present a *nominal syntax* that will prove to be expedient to the interactive semantics we are considering in the present paper. This introduces names that are to be understood as generalized variables. They are drawn from a countable set of names $\mathcal{N} = \mathcal{L} \cup \mathcal{C} \cup \mathcal{K}$ whose names we will range over with α. This set is partitioned into *function names* $\mathcal{L}$ whose elements we range over by f, g, *delimited* and *undelimited continuation names* range over by $\kappa, \varkappa \in \mathcal{K}$ and $c, d \in \mathcal{C}$ respectively, as well as *effect names* $e \in \mathcal{E}$.

Apart from these names that we highlight in the following figures, the rest of syntax Λ_{eff} should be standard. We give in figure 1 the algebra of types and the syntax of expressions.

The types are generated from the ground positive types $\mathbb{1}$, $\mathbb{B}$ and $\mathbb{N}$, for *unit*, *booleans* and *integers* respectively, as well as effect types $\mathbb{E}$. The latter are given by their corresponding signatures $\{(\mathbf{op}_i : \tau_i \twoheadrightarrow \varphi_i)_{i\in I}\}$ that associate each *operation symbol* **op** in the theory of $\mathbb{E}$ with its arity $\tau \twoheadrightarrow \varphi$, where τ is the type of its parameter and φ is its "return" type, *i.e.* the type expected by its enclosing continuation. The double-headed arrow $\twoheadrightarrow$ distinguishes them from λ-abstractions.

Application is of the form vw, restricted to values. More general form of application $\mathtt{tu}$ between terms can be defined as $\mathsf{let}\, x = \mathtt{t}\, \mathsf{in}\, \mathsf{let}\, y = \mathsf{u}\, \mathsf{in}\, xy$, specifying explicitely the order of execution, as found in Administriative Normal Form (ANF) languages [13].

The construct $\mathbf{new}\,\mathbb{E}$ is responsible for generating a fresh instance of the effect $\mathbb{E}$, whereas the expression $\iota\#\mathbf{op}\,\mathsf{v}$ is responsible for *performing* the effect. It is formed by an operation symbol **op** associated to an effect $\mathbb{E}$ and a particular instance ι of this effect, together with a value v that is provided to this operation.

The construct $\{\mathtt{t}\}\,\mathbf{with}\,\mathsf{h}$ is responsible for handling these effects, and it is formed by a potentially effectful term $\mathtt{t}$ and a handler h. The handlers are defined as a non-empty set of clauses; one *return clause* of the shape $\{\mathbf{ret}\, x \mapsto \mathtt{t}\}$ defining the default behaviour in case no effect is caught, and a finite set of *operation clauses* $\{\iota\#\mathbf{op}\, x\, k \mapsto \mathtt{t}\}$ defining the intended behaviour of each operation symbol **op** involved.

Given a handler $\mathsf{h} = \{\mathbf{ret}\, x \mapsto \mathtt{t}\} \uplus_{i\in I} (\{\iota_i\#\mathbf{op}_i\, x_i\, k_i \mapsto \mathsf{u}_i\}_i)$, we define $\mathsf{h}^{\mathbf{ret}} := \{\mathbf{ret}\, x \mapsto \mathtt{t}\}$ and $\mathsf{h}^{\iota_i\#\mathbf{op}_i} := \{\iota_i\#\mathbf{op}_i\, x_i\, k_i \mapsto \mathsf{u}_i\}$, and $\mathbf{hdl}(\mathsf{h}) := \{(\iota_i\#\mathbf{op}_i)_{i\in I}\}$ (*i.e.* the exact set of effects handled by h).

We overload the symbol **hdl** to define the function that associates an evaluation context to the set of effects it handles.

$$\mathbf{hdl}([]) := \emptyset \qquad \mathbf{hdl}(\mathsf{let}\, x = \mathsf{E}\, \mathsf{in}\, \mathtt{t}) := \mathbf{hdl}(\mathsf{E})$$
$$\mathbf{hdl}(\{\mathsf{E}\}\,\mathbf{with}\,\mathsf{h}) := \mathbf{hdl}(\mathsf{E}) \cup \mathbf{hdl}(\mathsf{h})$$

Λ_{eff} is also equipped with the standard construct $\mathbf{match}\,\mathsf{v}\,\mathbf{with}\,\mathsf{p}$ to destruct positive values. Pairs are destructed by being matched with the pattern $\langle x, y\rangle$ of a pattern case of the form $\{\langle x, y\rangle \Rightarrow \mathtt{t}\}$, and integers are pattern-matched over using $\{\mathsf{n}_0 \Rightarrow \mathtt{t}_0, \ldots, \mathsf{n}_k \Rightarrow \mathtt{t}_k\} \uplus \{x \Rightarrow \mathtt{t}\}$, where the last pattern serves as a catch-all case for exhaustiveness.

Evaluation contexts S, as introduced in [12], are terms with a single hole $[]$, that appear in specific positions corresponding to where the evaluation of a redex may happen, respectively $\mathsf{let}\, x = []\, \mathsf{in}\, \mathtt{t}$ and $\{[]\}\,\mathbf{with}\,\mathsf{h}$. Their syntax is extended with two notions of *continuation names* c and κ, that are a form of free variables used to represent respectively undelimited and delimited continuations. Such continuation names are called covariable in the $\lambda\mu$-calculus.

The fact that application is restricted to ANF avoids considering [] t and v [] as evaluation contexts. Plugging a term inside an undelimited continuation name $c[\mathtt{t}]$ can be understood as a named term in the $\lambda\mu$-calculus.

We write $\mathtt{S}[\mathtt{T}]$ for the hole filling operation, that corresponds to the operation of stacking-up $\mathtt{T}$ on top of $\mathtt{S}$.

types	τ, φ	::=	$\mathbb{1} \mid \mathbb{B} \mid \mathbb{N} \mid \tau \times \varphi \mid \tau \to \varphi \mid \mathbb{E}$
signatures	$\mathbb{E}$	::=	$\{(\mathbf{op}_i : \tau_i \twoheadrightarrow \varphi_i)_i\}$

(a) types

values	$\mathtt{v}, \mathtt{w}$	:=	$\langle\rangle \mid \mathtt{ff} \mid \mathtt{tt} \mid \mathtt{n} \mid \langle \mathtt{v}, \mathtt{w} \rangle \mid \iota$
		|	$x \mid \lambda x.\mathtt{t} \mid \boxed{f}$
effects	$\mathtt{e}$	::=	$\mathtt{v}\#\mathbf{op}\,\mathtt{w} \mid \boxed{e}$
terms	$\mathtt{t}, \mathtt{u}$	::=	$\mathbf{ret}\,\mathtt{v} \mid \mathtt{e} \mid \mathbf{new}\,\mathbb{E}$
		|	$\mathtt{w}\,\mathtt{v} \mid \mathtt{if}\,\mathtt{v}\,\mathtt{then}\,\mathtt{t}\,\mathtt{else}\,\mathtt{u}$
		|	$\mathbf{match}\,\mathtt{v}\,\mathbf{with}\,\mathtt{p}$
		|	$\mathtt{let}\,x = \mathtt{t}\,\mathtt{in}\,\mathtt{u} \mid \{\mathtt{t}\}\,\mathbf{with}\,\mathtt{h} \mid \alpha[\mathtt{t}]$
pattern cases	$\mathtt{p}$	::=	$\{x \Rightarrow \mathtt{t}\} \mid \{\langle x, y\rangle \Rightarrow \mathtt{t}\}$
		|	$\{\mathtt{n} \Rightarrow \mathtt{t}\} \uplus \mathtt{p}$
handlers	$\mathtt{h}$	::=	$\{\mathbf{ret}\,x \mapsto \mathtt{t}\} \mid \{\mathtt{v}\#\mathbf{op}\,x\,y \mapsto \mathtt{t}\} \uplus \mathtt{h}$

evaluation frames	$\mathtt{E}$	::=	$\mathtt{let}\,x = [\,]\,\mathtt{in}\,\mathtt{t} \mid \{[\,]\}\,\mathbf{with}\,\mathtt{h}$
evaluation contexts	$\mathtt{S}, \mathtt{T}$	::=	$[\,] \mid \boxed{c}[\mathtt{S}] \mid \boxed{\kappa}[\mathtt{S}] \mid \mathtt{E}[\mathtt{S}]$

(b) expressions

Figure 1: Λ_{eff} syntax.

2.1 Typing rules

The full typing system of Λ_{eff} is given in figure 2.

We consider two judgement relations $\vdash_{\mathtt{v}}$ and $\vdash_{\mathtt{c}}$ defined on values and terms respectively. The typing contexts are of the shape $\mathtt{I}; \Gamma; \Theta$ where $\mathtt{I}$ is an *effect instance context* mapping fresh effect instances ι with their types and where Θ and Γ are typing contexts respectively for variables and names.

We will often abuse the notations of maps and sets, *e.g.* $\Theta(x)$ or $\Gamma(\alpha)$, to denote the type τ such that $x : \tau \in \Theta$ or $\alpha : \tau \in \Gamma$.

The handlers occur in expressions, but are not first-class values. They are assigned types written $\tau \Rightarrow \varphi$ (that do not appear in the grammar of types in figure 1) as well as a distinct typing judgement given by the relation $\vdash_{\mathtt{h}}$. The understanding conveyed is that putting a computation of type τ inside a handler of type $\tau \Rightarrow \varphi$ gives a term of type φ. Again, the double arrow $\Rightarrow$ distinguishes handlers from λ-abstractions whose input are values instead of computations.

The rule Handler ensures that all the branches of the handler expect the same type τ and continue computing with the same type φ guaranteeing a static typing discipline.

The possibility of handling a freshly generated effect instance is a consequence of the ability to write a handler parametrized by a variable that can be substituted by this fresh instance, *e.g.*

$$\mathtt{let}\,\boxed{x} = \mathbf{new}\,\mathbb{E}\ \mathtt{in}\,\{\mathtt{t}\}\,\mathbf{with}\,\{\mathbf{ret}\,y \mapsto \mathtt{t};\ \boxed{x}\#\mathbf{op}\,y\,k \mapsto \mathtt{u}\}$$

The rule Eff checks whether the variable has already been instantiated, in which case the corresponding value and its type is to be found in the context $\mathtt{I}$, otherwise its type is to be found in Θ (like any other variable in the scope).

$$\textsc{Var}\ \frac{\Theta(x) = \tau}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} x : \tau} \qquad \textsc{FName}\ \frac{\Gamma(f) = \tau}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} f : \tau} \qquad \textsc{Inst}\ \frac{\mathtt{I}(\iota) = \mathbb{E}}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} \iota : \mathbb{E}}$$

$$\mathrm{I}_{\mathbb{1}}\ \frac{}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} \langle\rangle : \mathbb{1}} \qquad \mathrm{I}_{\mathtt{t}}\ \frac{}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} \mathtt{tt} : \mathbb{B}} \qquad \mathrm{I}_{\mathtt{f}}\ \frac{}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} \mathtt{ff} : \mathbb{B}}$$

$$\mathrm{I}_{\to}\ \frac{\mathtt{I}; \Gamma; \Theta, x : \tau \vdash_{\mathtt{c}} \mathtt{t} : \varphi}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} \lambda x.\mathtt{t} : \tau \to \varphi}$$

$$\mathrm{E}_{\to}\ \frac{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} \mathtt{w} : \tau \to \varphi \qquad \mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} \mathtt{v} : \tau}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{c}} \mathtt{w}\,\mathtt{v} : \varphi}$$

$$\mathrm{I}_{\mathbf{ret}}\ \frac{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} \mathtt{v} : \tau}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{c}} \mathbf{ret}\,\mathtt{v} : \tau}$$

$$\textsc{Handler}\ \frac{\left(\mathtt{I}; \Gamma; \Theta, x_i : \tau_i, \kappa_i : \varphi_i \to \tau \vdash_{\mathtt{c}} \mathtt{t}_i : \varphi \qquad \mathtt{I}; \Gamma; \Theta, x_i : \tau_i \vdash_{\mathtt{c}} \mathtt{v}_i\#\mathbf{op}_i\,x_i : \varphi_i\right)_{i \in \mathcal{I}} \quad \mathtt{I}; \Gamma; \Theta, x : \tau \vdash_{\mathtt{c}} \mathtt{t} : \varphi}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{h}} \{\mathbf{ret}\,x \mapsto \mathtt{t}\} \uplus \{\mathtt{v}_i\#\mathbf{op}_i\,x_i\,\kappa_i \mapsto \mathtt{t}_i\}_{i \in \mathcal{I}} : \tau \Rightarrow \varphi}$$

$$\mathrm{E}_{\mathbf{handle}}\ \frac{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{c}} \mathtt{t} : \tau \qquad \mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{h}} \mathtt{h} : \tau \Rightarrow \varphi}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{c}} \{\mathtt{t}\}\,\mathbf{with}\,\mathtt{h} : \varphi}$$

$$\frac{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{c}} \mathtt{t} : \tau \qquad \mathtt{I}; \Gamma; \Theta, x : \tau \vdash_{\mathtt{c}} \mathtt{u} : \varphi}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{c}} \mathtt{let}\,x = \mathtt{t}\,\mathtt{in}\,\mathtt{u} : \varphi}\ \mathrm{E}_{\mathtt{let}}$$

$$\textsc{Eff}\ \frac{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} \mathtt{w} : \mathbb{E} \qquad \mathbf{op} : \tau \twoheadrightarrow \varphi \in \mathbb{E} \qquad \mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{v}} \mathtt{v} : \tau}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{c}} \mathtt{w}\#\mathbf{op}\,\mathtt{v} : \varphi}$$

$$\textsc{EffName}\ \frac{e \in \mathrm{dom}(\Gamma)}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{c}} e :?} \qquad \frac{}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{c}} \mathbf{new}\,\mathbb{E} : \mathbb{E}}\ \textsc{New}$$

(a) type judgements for expressions

$$\textsc{Hole}\ \frac{}{\mathtt{I}; \Gamma \vdash_{\mathtt{s}} [\,] : \tau \rightsquigarrow \tau}$$

$$\frac{\mathtt{I}; \Gamma \vdash_{\mathtt{s}} \mathtt{T} : \varphi \rightsquigarrow \tau \qquad \mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{c}} \mathtt{t} : v \qquad x : \tau \in \Theta}{\mathtt{I}; \Gamma \vdash_{\mathtt{s}} \mathtt{let}\,x = \mathtt{T}\,\mathtt{in}\,\mathtt{t} : \varphi \rightsquigarrow v}$$

$$\frac{\mathtt{I}; \Gamma \vdash_{\mathtt{s}} \mathtt{T} : \tau \rightsquigarrow \varphi \qquad \mathtt{I}; \Theta \vdash_{\mathtt{h}} \mathtt{h} : \varphi \Rightarrow v}{\mathtt{I}; \Gamma \vdash_{\mathtt{s}} \{\mathtt{T}\}\,\mathbf{with}\,\mathtt{h} : \tau \rightsquigarrow v}$$

$$\textsc{ContName}\ \frac{\Gamma(c) = \tau \rightsquigarrow \bot}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{s}} c[\,] : \tau \rightsquigarrow \bot} \qquad \frac{\Gamma(\kappa) = \tau \rightsquigarrow \varphi}{\mathtt{I}; \Gamma; \Theta \vdash_{\mathtt{s}} \kappa[\,] : \tau \rightsquigarrow \varphi}$$

(b) typing of evaluation contexts.

Figure 2: type system of Λ_{eff}

The type ? denotes the *abstract* type of some effect name e, representing an algebraic effect being performed by the environment, but whose type is *unknown*. Its type is unknown as the enclosing delimited continuation belongs to the environment and could be expecting any type as input.

The typing of evaluation contexts is given by the typing relation $\vdash_s$ which assigns them types of the shape $\tau \rightsquigarrow \varphi$, with the understanding that the following rule is admissible.

$$\frac{\mathtt{I};\Theta \vdash_s \mathtt{S} : \tau \rightsquigarrow \varphi \qquad \mathtt{I};\Theta \vdash_c \mathtt{t} : \tau}{\mathtt{I};\Theta \vdash_c \mathtt{S}[\mathtt{t}] : \varphi}$$

When typing undelimited continuation names, we will sometimes use the syntactic sugar $\neg\tau$ for the type $\tau \rightsquigarrow \bot$.

2.2 Operational semantics

The propagation of effects through continuation names is to be seen as a form of interaction (in a *game semantics* sense) whereby the *program* inquires whether the *environment*'s code (corresponding to the abstract continuation) handles some specific effect or not. For this reason, we emphasize the stack structure of evaluation contexts and we privilege an abstract machine presentation of the *operational semantics* (*à la* [15]) whose reduction rules make this flow of information explicit.

The main component of the abstract machine configurations are *running computations* $\mathtt{M}$ whose shape is given by:

$$\mathtt{M} := \langle \mathtt{t} \mid \mathtt{S} \circ \mathtt{T} \rangle$$

It consists of the *active term* $\mathtt{t}$, the *forwarded evaluation context* $\mathtt{T}$, *followed* by the *active evaluation context* $\mathtt{S}$, and corresponds to a fine decomposition of the term $\mathtt{S}[\mathtt{T}[\mathtt{t}]]$.

The *reduction relation* $\mapsto_{\mathsf{op}}$ is defined in figure 3 over pairs $(\mathtt{M}, \mathtt{I})$ formed by a running computation $\mathtt{M}$ and an effect instance context $\mathtt{I}$ crucial to provide a semantics to **new** $\mathbb{E}$. In the following, we will write $\langle \mathtt{t} \mid \mathtt{S} \circ \mathtt{T} \mid \mathtt{I} \rangle$ for $(\langle \mathtt{t} \mid \mathtt{S} \circ \mathtt{T} \rangle \mid \mathtt{I})$ and we will often use $\langle \mathtt{t} \mid \mathtt{S} \rangle$ as *syntactic sugar* for $\langle \mathtt{t} \mid \mathtt{S} \circ [] \rangle$ and conflate a term $\mathtt{t}$ with $\langle \mathtt{t} \mid [] \rangle$, *i.e.* its initial embedding in running computations.

The $\mapsto_{\mathsf{push}}$ reduction takes care of the administrative task of unpiling the evaluation frames in the active term in order to expose the underlying redex for $\mapsto_{\mathsf{eval}}$ and $\mapsto_{\mathsf{fwd}}$.

The only non-standard rules are the starred ones $(\star_1)$ and $(\star_2)$, owing to the presence of names in the syntax.

The rule $(\star_1)$ extends the definition of the relation $\mapsto_{\mathsf{fwd}}$ to account for the propagation of effects in the presence of effect names and does that by mirroring the similar preceding rule. Naturally, given that e is abstract then so is the underlying effect instance, therefore the top evaluation frame $\mathtt{E}$ in the active evaluation context cannot *a priori* handle it and should be moved to the forwarded evaluation context. Similarly, the rule $(\star_2)$ accounts for continuation names κ and c when unpiling evaluation contexts.

Notice that we did not add the unsound rule (✗) which may seem like the innocuous dual to $(\star_1)$.

$$\langle \iota\#\mathbf{op}\ \mathtt{v} \mid \mathtt{S}[\kappa[]] \circ \mathtt{T} \rangle \mapsto_{\mathsf{fwd}} \langle \iota\#\mathbf{op}\ \mathtt{v} \mid \mathtt{S} \circ \kappa[\mathtt{T}] \rangle \qquad (\text{✗})$$

Indeed, as it will be evident in section 3, the ability to capture delimited continuations means that the environment can capture fragments of the program's evaluation context, and as a result, gain control of a program's handler. Therefore, the possibility of

$$\langle (\lambda x.\mathtt{t})\ \mathtt{v} \mid \mathtt{S} \mid \mathtt{I} \rangle \mapsto_{\mathsf{eval}} \langle \mathtt{t}\{x := \mathtt{v}\} \mid \mathtt{S} \mid \mathtt{I} \rangle$$

$$\langle \mathbf{match}\ \langle \mathtt{v}, \mathtt{w} \rangle\ \mathbf{with}\ \langle x, y \rangle \Rightarrow \mathtt{t} \mid \mathtt{S} \mid \mathtt{I} \rangle \mapsto_{\mathsf{eval}} \langle \mathtt{t}\{x := \mathtt{v}\}\{y := \mathtt{w}\} \mid \mathtt{S} \mid \mathtt{I} \rangle$$

$$\langle \mathbf{match}\ \mathtt{n}\ \mathbf{with}\ \mathtt{p} \mid \mathtt{S} \mid \mathtt{I} \rangle \mapsto_{\mathsf{eval}} \langle \mathtt{t} \mid \mathtt{S} \mid \mathtt{I} \rangle \quad \text{when } \{\mathtt{n} \Rightarrow \mathtt{t}\} \in \mathtt{p}$$

$$\langle \mathbf{match}\ \mathtt{n}\ \mathbf{with}\ \mathtt{p} \mid \mathtt{S} \mid \mathtt{I} \rangle \mapsto_{\mathsf{eval}} \langle \mathtt{t}\{x := \mathtt{n}\} \mid \mathtt{S} \mid \mathtt{I} \rangle \quad \text{when } \{\mathtt{n} \Rightarrow \_\} \notin \mathtt{p} \wedge \{x \Rightarrow \mathtt{t}\} \in \mathtt{p}$$

$$\langle\ \mathbf{new}\ \mathbb{E} \mid \mathtt{S} \mid \mathtt{I} \rangle \mapsto_{\mathsf{eval}} \langle \mathbf{ret}\ \iota \mid \mathtt{S} \mid \mathtt{I} \cdot [\iota \mapsto \mathbb{E}] \rangle$$

$$\langle \mathbf{ret}\ \mathtt{v} \mid \mathtt{S}[\mathsf{let}\ x = []\ \mathsf{in}\ \mathtt{t}] \mid \mathtt{I} \rangle \mapsto_{\mathsf{eval}} \langle \mathtt{t}\{x := \mathtt{v}\} \mid \mathtt{S} \mid \mathtt{I} \rangle$$

$$\langle \mathbf{ret}\ \mathtt{v} \mid \mathtt{S}[\{[]\}\ \mathbf{with}\ \mathtt{h}] \mid \mathtt{I} \rangle \mapsto_{\mathsf{eval}} \langle \mathtt{t}\{x := \mathtt{v}\} \mid \mathtt{S} \mid \mathtt{I} \rangle \quad \text{when } \mathtt{h}^{\mathbf{ret}} = \{\mathbf{ret}\ x \mapsto \mathtt{t}\}$$

$$\langle \iota\#\mathbf{op}\ \mathtt{v} \mid \mathtt{S}[\{[]\}\ \mathbf{with}\ \mathtt{h}] \circ \mathtt{T} \mid \mathtt{I} \rangle \mapsto_{\mathsf{eval}} \langle \mathtt{t}\{x := \mathtt{v}\}\{y := \lambda z.\{\mathtt{T}[\mathbf{ret}\ z]\}\ \mathbf{with}\ \mathtt{h}\} \mid \mathtt{S} \mid \mathtt{I} \rangle \quad \text{when } \mathtt{h}^{\mathbf{op}} = \{\iota\#\mathbf{op}\ x\ y \mapsto \mathtt{t}\}$$

$$\langle \iota\#\mathbf{op}\ \mathtt{v} \mid \mathtt{S}[\mathtt{E}] \circ \mathtt{T} \mid \mathtt{I} \rangle \mapsto_{\mathsf{fwd}} \langle \iota\#\mathbf{op}\ \mathtt{v} \mid \mathtt{S} \circ \mathtt{E}[\mathtt{T}] \mid \mathtt{I} \rangle \quad \text{when } \iota\#\mathbf{op} \notin \mathbf{hdl}(\mathtt{E})$$

$$\langle e \mid \mathtt{S}[\mathtt{E}] \circ \mathtt{T} \mid \mathtt{I} \rangle \mapsto_{\mathsf{fwd}} \langle e \mid \mathtt{S} \circ \mathtt{E}[\mathtt{T}] \mid \mathtt{I} \rangle \qquad (\star_1)$$

$$\langle \mathtt{E}[\mathtt{t}] \mid \mathtt{S} \mid \mathtt{I} \rangle \mapsto_{\mathsf{push}} \langle \mathtt{t} \mid \mathtt{S}[\mathtt{E}] \mid \mathtt{I} \rangle$$

$$\langle \alpha[\mathtt{t}] \mid \mathtt{S} \mid \mathtt{I} \rangle \mapsto_{\mathsf{push}} \langle \mathtt{t} \mid \mathtt{S}[\alpha[]] \mid \mathtt{I} \rangle \qquad (\star_2)$$

Figure 3: operational semantics

$\kappa[]$ handling the effect (even though $\iota\#\mathbf{op}\ \mathtt{v}$ is abstract from the environment's perspective) cannot *a priori* be ruled out.

Non-termination. Recursion and divergence can be restored with Landin's knot[24] through an encoding of higher-order state using handlers. We illustrate with the following diverging term, that we will denote by Ω, the shape of the *backpatching* it involves.

$$\begin{aligned}\mathtt{let}\ f = \Big(&\{\mathsf{let}\ f = \iota\#\mathbf{read}\ \langle\rangle\ \mathsf{in}\ f\ \langle\rangle\}\ \mathbf{with}\\ &\mid \mathbf{ret}\ x \mapsto \mathbf{ret}\ (\lambda g.\ \mathbf{ret}\ x)\\ &\mid \iota\#\mathbf{read}\ \langle\rangle\ k \mapsto \mathbf{ret}\ (\lambda g.\ k\ g\ g)\Big)\\ \mathtt{in}\ f\ &(\lambda x.\ \mathsf{let}\ y = \iota\#\mathbf{read}\ \langle\rangle\ \mathsf{in}\ y\ x)\end{aligned}$$

(1) Ω encodes a higher-order state reader effect given by the signature $\{\mathbf{read} : \mathbb{1} \twoheadrightarrow (\mathbb{1} \to \mathbb{1})\}$ with the handler [1] $\mathtt{h} = \{\mathbf{ret}\ x \mapsto \mathbf{ret}\ (\lambda g.\ \mathbf{ret}\ x);\ \iota\#\mathbf{read}\ \langle\rangle\ k \mapsto \mathbf{ret}\ (\lambda g.\ k\ g\ g)\}$ reading from a "cell" identified by the effect instance ι.

(2) The state is initialised with $\lambda x.\ \mathsf{let}\ y = \iota\#\mathbf{read}\ \langle\rangle\ \mathsf{in}\ y\ x$, which reads a function from the "cell" ι, binds it to y and then passes its argument to it.

[1] $k\ g\ g$ is syntactic sugar for $\mathsf{let}\ x = k\ g\ \mathsf{in}\ x\ g$.

(3) The term $\mathsf{let}\ f = \iota\#\mathbf{read}\ \langle\rangle\ \mathsf{in}\ f\ \langle\rangle$ makes a call to this unguarded self-calling function and triggers the loop.

Notations: We will write $(\mathtt{M} \mid \mathtt{I}) \mapsto_{\mathtt{x}}$, where $\mathtt{x} \in \{\mathrm{op}, \mathrm{eval}, \mathrm{fwd}\}$ to indicate that there exist an effect instance context $\mathtt{I}'$ and a term $\mathtt{N}$ such that $(\mathtt{M} \mid \mathtt{I}) \mapsto_{\mathtt{x}} (\mathtt{N} \mid \mathtt{I}')$. Its negation will be denoted by $(\mathtt{M} \mid \mathtt{I}) \not\mapsto_{\mathtt{x}}$.

We will also write $(\mathtt{M} \mid \mathtt{I}) \Downarrow_{\mathrm{op}} (\mathtt{N} \mid \mathtt{I}')$ to mean that $(\mathtt{M} \mid \mathtt{I}) \mapsto^*_{\mathrm{op}} (\mathtt{N} \mid \mathtt{I}')$ and that $(\mathtt{N} \mid \mathtt{I}') \not\mapsto_{\mathrm{op}}$, and the notation $(\mathtt{M} \mid \mathtt{I}) \Downarrow_{\mathrm{op}}$ will be used to assert the existence of $\mathtt{I}'$ and $\mathtt{N}$ in *s.t.* $(\mathtt{M} \mid \mathtt{I}) \Downarrow_{\mathrm{op}} (\mathtt{N} \mid \mathtt{I}')$.

For divergence, we will adopt the notation $(\mathtt{M} \mid \mathtt{I}) \Uparrow_{\mathrm{op}}$ to indicate that $(\mathtt{M} \mid \mathtt{I})$ diverges, *i.e.* forall $(\mathtt{N} \mid \mathtt{I}')$ such that $(\mathtt{M} \mid \mathtt{I}) \mapsto^*_{\mathrm{op}} (\mathtt{N} \mid \mathtt{I}')$ then $(\mathtt{N} \mid \mathtt{I}') \mapsto_{\mathrm{op}}$.

2.3 Metatheory

2.3.1 Interactive/Private fragment. In the following, we will mostly be interested in the *interactive* and *private* subset of terms and computations on which our OGS semantics operate. In this fragment, generated instances remain private to the program and are never made available to the external environment.

Since the syntax of expressions of such a fragment does not differ from that Λ_{eff}, we will capture it by a restricted typing judgement relation $\Vdash_{\mathtt{c}} \subset \vdash_{\mathtt{c}}$, that will ensure that no effect instances can be exchanged between the program and its environment. Indeed, given an Λ_{eff}-term $\mathtt{I}; \Theta \vdash_{\mathtt{c}} \mathtt{t} : \tau$, the possibility of $\mathtt{t}$ exchanging effect instances with a compatible environment is indicated by the occurrence of effect signatures $\mathbb{E}$ in Γ.

Definition 2.1 (interactive expressions). A term $\mathtt{t}$ is said to be *interactive* when it is well-typed and admits a typing of the form $\mathtt{I}; \Gamma \Vdash_{\mathtt{c}} \mathtt{t}$, defined as

$$\frac{\mathtt{I}; \Gamma; \emptyset \vdash_{\mathtt{c}} \mathtt{t} : \bot \qquad \mathrm{codom}(\Gamma) \subset \mathcal{P}}{\mathtt{I}; \Gamma \Vdash_{\mathtt{c}} \mathtt{t}}$$

where $\mathcal{P}$ is the smallest set of types that subsumes $\{\mathbb{1}, \mathbb{B}, \mathbb{N}\}$ and is closed by the type constructors $\rightarrow$, $\rightsquigarrow$, $\times$ and $\neg$.

A running computation $\mathtt{M}$ is said to be *interactive* and written $\mathtt{I}; \Gamma \Vdash_{\mathtt{c}} \mathtt{M}$ when the underlying term is interactive, *i.e.*

$$\frac{\mathtt{I}; \Gamma; \emptyset \vdash_{\mathtt{c}} \mathtt{S}[\mathtt{T}[\mathtt{t}]] : \bot}{\mathtt{I}; \Gamma \Vdash_{\mathtt{c}} \langle \mathtt{t} \mid \mathtt{S} \circ \mathtt{T} \rangle}$$

2.3.2 Normal forms. Since *normal forms* describe the interaction interface between the program and its environment, we will privilege a useful presentation where we identify the *interaction patterns* that are causing a computation to be stuck. By expliciting these patterns, the normal forms are written as $\mathtt{P}(\alpha)$; they involve a *copattern* $\mathtt{P}$ (described in figure 5) representing the program controlled code and a *name* α abstracting an environment controlled codata. The token $\Box$ represents a *codata hole* whose *filling*, *i.e.* the syntactic substitution thereof, is given by the application $(\mathtt{P}, x) \mapsto \mathtt{P}(x)$ of *copatterns* to *codata* defined as $\mathtt{P}(x) := \mathtt{P}\{\Box \mapsto x\}$.

Borrowing from the terminology of [7], we identify in figure 4 *open-stuck* terms in which the reduction of the term depends on an abstract function f or an abstract delimited continuation $\kappa[]$ and where the continuation is controlled by the program, as well as *context-stuck* terms corresponding to unhandled effects and fully-evaluated returners inside an abstract continuation $c[]$. The two

$$\begin{array}{lcll} \mathtt{Nf} & ::= & \langle \mathbf{ret}\ \mathtt{v} \mid c[]\ \rangle \mid \langle \mathbf{ret}\ \mathtt{v} \mid \mathtt{S}[\kappa[]] \rangle & \textit{evaluated returners} \\ & \mid & \langle \mathtt{e} \mid \mathtt{S}[\kappa[]] \circ \mathtt{T} \rangle \mid \langle \mathtt{e} \mid c[] \circ \mathtt{T} \rangle & \textit{unhandled effects} \\ & \mid & \langle f\ \mathtt{v} \mid \mathtt{S} \rangle & \textit{open applications} \end{array}$$

Figure 4: interactive normal forms of Λ_{eff}.

classes of normal forms correspond to different forms of interaction; the former calls for a *question* whereas the latter for an *answer*.

$$\begin{array}{lcll} \mathtt{P} & ::= & \langle \mathbf{ret}\ \mathtt{v} \mid \Box \rangle \mid \langle \mathtt{e} \mid \Box \circ \mathtt{S} \rangle & \textit{context-stuck} \\ & \mid & \langle \mathbf{ret}\ \mathtt{v} \mid \mathtt{S}[\Box] \rangle \mid \langle \mathtt{e} \mid \mathtt{S}[\Box] \circ \mathtt{T} \rangle & \textit{open-stuck} \\ & \mid & \langle \Box\ \mathtt{v} \mid \mathtt{S} \rangle & \end{array}$$

(a) syntax of copatterns

$$\left.\begin{array}{l} \textsc{cont-val}\ \dfrac{\mathtt{I}; \Gamma \vdash_{\mathtt{v}} \mathtt{v} : \tau}{\mathtt{I}; \Gamma \vdash_{\mathtt{p}} \langle \mathbf{ret}\ \mathtt{v} \mid \Box \rangle : \neg\neg\tau} \\[2ex] \textsc{cont-eff}\ \dfrac{\mathtt{I}; \Gamma; \vdash_{\mathtt{c}} \mathtt{e} : \tau \qquad \mathtt{I}; \Gamma \vdash_{\mathtt{s}} \mathtt{S} : \tau \rightsquigarrow \varphi}{\mathtt{I}; \Gamma \vdash_{\mathtt{p}} \langle \mathtt{e} \mid \Box \circ \mathtt{S} \rangle : \neg\neg\varphi} \end{array}\right\} \textit{context-stuck}$$

$$\left.\begin{array}{l} \textsc{lambda}\ \dfrac{\mathtt{I}; \Gamma \vdash_{\mathtt{v}} \mathtt{v} : \tau \qquad \mathtt{I}; \Gamma \vdash \mathtt{S} : \neg\varphi}{\mathtt{I}; \Gamma \vdash_{\mathtt{p}} \langle \Box\ \mathtt{v} \mid \mathtt{S} \rangle : \neg(\tau \rightarrow \varphi)} \\[2ex] \textsc{delim-cont}\ \dfrac{\mathtt{I}; \Gamma \vdash_{\mathtt{p}} \mathtt{P} : \neg\neg\tau \qquad \mathtt{I}; \Gamma \vdash \mathtt{S} : \neg\varphi}{\mathtt{I}; \Gamma \vdash_{\mathtt{p}} \mathtt{P}(\mathtt{S}[\Box]) : \neg(\tau \rightsquigarrow \varphi)} \end{array}\right\} \textit{open-stuck}$$

(b) typing judgements for copatterns

Figure 5: copatterns: syntax and types.

The interactive fragment of Λ_{eff} satisfies the two following properties pertaining to type preservation and normal forms.

Lemma 2.2 (subject reduction). *Given two interactive running computations* $\mathtt{M}$ *and* $\mathtt{N}$ *s.t.* $\mathtt{I}; \Gamma \Vdash_{\mathtt{c}} \mathtt{M}$ *and* $(\mathtt{M} \mid \mathtt{I}) \mapsto_{\mathrm{op}} (\mathtt{N} \mid \mathtt{I}')$*, then* $\mathtt{I}'; \Gamma \Vdash_{\mathtt{c}} \mathtt{N}$.

Lemma 2.3 (normal forms shape). *For all interactive computations* $\mathtt{I}; \Gamma \Vdash_{\mathtt{c}} \mathtt{M}$ *such that* $(\mathtt{M} \mid \mathtt{I}) \not\mapsto_{\mathrm{op}}$*, there exists a copattern* $\mathtt{P}$ *and a name* $\alpha \in \mathrm{dom}(\Gamma)$ *s.t.* $\mathtt{M} = \mathtt{P}(\alpha)$.

Our notion of program equivalence builds on the *Closed Instantiations of Uses* (CIU) equivalence, introduced by Mason and Talcott [27]. Whereas Morris-style contextual equivalence [28] quantifies over general contexts—including those placing holes under λ-abstractions to handle open terms—CIU-equivalence restricts attention to evaluation contexts and handles open terms by quantifying over substitution maps as well.

In our setting, we refine this idea by leveraging the distinction between variables and names: we define equivalence only for interactive terms $\mathtt{I}; \Gamma \Vdash_{\mathtt{c}} \mathtt{M}$, where $\mathtt{M}$ contains no free variables, and

all names are drawn from Γ. We assume that Γ contains exactly one continuation name, and no delimited continuation names. We refer to such typing contexts as initial, since these constraints do not generally hold as interaction proceeds. Equivalence between $\mathtt{M}_1$ and $\mathtt{M}_2$ is then defined via mutual approximation. Quantifying over all name assignments for Γ, the continuation name d is interpreted as an evaluation context. This yields a form of CIU-equivalence based solely on quantification over name assignments.

We begin by introducing the type judgments for name assignments. We range over these by γ and δ and assign them name type contexts, as shown below. Here, $\mathtt{I}, \Gamma$ denotes a typing context, and Δ is a name context.

$$\mathtt{I}; \Gamma \vdash \gamma : \Delta := \forall \alpha \in \mathrm{dom}(\gamma). \exists \mathtt{x} \in \{\mathtt{v}, \mathtt{c}, \mathtt{s}\}.\ \mathtt{I}; \Gamma \vdash_{\mathtt{x}} \gamma(\alpha) : \Delta(\alpha)$$

We will use ε to denote an empty name assignments (map).

Definition 2.4 (ciu-approximation). We consider two interactive terms $\mathtt{M}_1, \mathtt{M}_2$ such that both $\emptyset; \Gamma \Vdash_{\mathtt{c}} \mathtt{M}_i$ (for $i \in \{1, 2\}$), with Γ initial. Then $\mathtt{M}_1$ is said to be *ciu-approximated* by $\mathtt{M}_2$, written $\Gamma \vdash \mathtt{M}_1 \preceq_{ciu} \mathtt{M}_2$, when for all instance contexts $\mathtt{I}$, for all continuation names $c_{\mathtt{f}}$ and for all name assignments γ such that $\mathtt{I}; c_{\mathtt{f}} : \neg\mathbb{1} \vdash \gamma : \Gamma$, if $(\mathtt{M}_1\{\gamma\} \mid \mathtt{I}) \Downarrow_{\mathrm{op}} c_{\mathtt{f}}[\langle\rangle]$ then $(\mathtt{M}_2\{\gamma\} \mid \mathtt{I}) \Downarrow_{\mathrm{op}} c_{\mathtt{f}}[\langle\rangle]$.

We note that this definition excludes terms with names of positive types such as $\mathbb{B}$ or $\mathbb{N}$, which should rather be handled using free variables. A mild generalization of our approach could accommodate such terms by also quantifying over variable substitutions. However, for the sake of simplicity, we leave this extension to future work.

3 Representing Interaction

In this section, we describe the *interactive execution* of a term (*Proponent*) with an abstract *environment* (*Opponent*) through a bipartite labelled transition system $\mathcal{L}_{\mathrm{OGS}}$ This is achieved in §3.5 by synchronizing the interaction LTS $\mathcal{L}_{\mathsf{AI}}$ with two disciplinary LTSs $\mathcal{L}_{\mathcal{T}}$ and $\mathcal{L}_{wb}$ that constrain Opponent's behaviour to conform to what is possible for an environment written in Λ_{eff} as well. These LTSs are introduced as follows:

- In §3.1, we introduce the key decomposition process of a normal form of Λ_{eff} into their observable part and their interacting part.
- In §3.2, we define $\mathcal{L}_{\mathsf{AI}}$ that characterizes the general game of interaction of Λ_{eff}, based on this decomposition of normal forms.
- In §3.3, we define$\mathcal{L}_{\mathcal{T}}$ that characterizes the type discipline of the interaction inherent to the typing of Λ_{eff}.
- In§3.4, the well-bracketing condition constraining the way delimited continuations can be used in Λ_{eff} is characterized by $\mathcal{L}_{wb}$.

3.1 Nominal abstraction

When an *active term* reaches a *normal form*, it discloses only the information needed (*i.e.* at the appropriate level of *abstraction*) for the environment to continue computing a value.
In a dual manner, upon receiving a this information, the *passive environment* (who knows its type) inspects its structure to infer the types of the underlying abstract components.

This is achieved through an *abstraction* and an appropriate *type inference* processes.

abstraction process: It is essentially the process that transforms a *syntactic object* into an ultimate pattern [25] (its positive skeleton) together with a corresponding name assignment map (its negative filling). Therefore given an object o, it generates a *nominal pattern* p and a name assignment γ with the understanding that $o = p\{\gamma\}$. Concretely, the codata occurring in the syntactic object are abstracted away and transformed into *names* to be handed over to the environment as part of various *nominal patterns*, capturing the fact that the latter's behaviour can only be parametric *w.r.t.* these *codata* (unlike *data* that could be inspected to trigger a *data-dependent* reaction).

nominal type inference: The general form of judgements in the type inference systems is the following: $\Gamma \vdash p : \tau \triangleright \Delta$; that is given a nominal pattern p of some expected type τ (inputs), this process infers the type of every name $n \in \mathrm{supp}(p)$ occurring in p and returns a name type context Δ (ouptput).

In the following, we will define the *nominal abstraction* of a few syntactic categories, where we will first introduce, for each, their abstract counterpart.

3.1.1 Abstract values. These are the values exchanged between Proponent and Opponent during their interaction. They are defined over a set of *function names* $f \in \mathcal{L}$:

$$\mathtt{V}, \mathtt{W} := \langle\rangle \mid \mathtt{n} \mid \mathtt{ff} \mid \mathtt{tt} \mid f \mid \langle \mathtt{V}, \mathtt{W} \rangle$$

We define in figure 6a the corresponding abstraction process of values, denoted by the infix symbol $\nearrow$, and the type inference system.

3.1.2 Abstract generalised values. Not only the structure of values is exposed in the interaction of Opponent and Proponent, but so are some fully-evaluated computations. These *generalised values*[2] are in turn abstracted away into patterns; *nominal generalised values*, which are defined as:

$$\mathtt{C}, \mathtt{D} := \mathbf{ret}\, \mathtt{V} \mid e \mid r[e]$$

where $e \in \mathcal{E}$ is an abstract *effect name* and $r \in \mathtt{List}(\mathcal{K})$ is a *abstract (delimited) evaluation stack* defined over the set $\mathcal{K}$ of *delimited continuations names*.

Analogously to abstract values, we define the nominal abstraction of generalised values in figure 6b, where the abstraction process is denoted by $\nearrow\!\!\!\!\!/$.

The type inference judgements are written $\Gamma \vdash_{\mathtt{c}} \mathtt{C} : \tau \triangleright \Delta_1; \Delta_2$ where the inferred type context is split into two: the first one for function names and the second one for effect and delimited continuation names.

3.1.3 Abstract normal forms. Continuing with our nominal abstraction of expressions, we introduce *abstract normal forms* and *abstract copatterns* and for the sake of a uniform treatment in-keeping with the classification of copatterns of figure 5, we will express them in terms of *abstract computations* in order to exhibit the underlying

[2] computations that *are* but cannot *do*.

$$\frac{\emptyset \vdash_{\mathsf{v}} \mathsf{v} : \tau \qquad \tau \in \{\mathbb{1}, \mathbb{N}, \mathbb{B}\}}{\mathsf{v} \nearrow (\mathsf{v}; \varepsilon)} \qquad \frac{}{\lambda x.\mathtt{t} \nearrow (f; [f \mapsto \lambda x.\mathtt{t}])}$$

$$\frac{\mathsf{v} \nearrow (\mathsf{V}; \gamma) \qquad \mathsf{w} \nearrow (\mathsf{W}; \gamma')}{\langle \mathsf{v}, \mathsf{w} \rangle \nearrow (\langle \mathsf{V}, \mathsf{W} \rangle, \gamma \cdot \gamma')} \qquad \frac{}{f \nearrow (g; [g \mapsto f])}$$

$$\frac{\emptyset \vdash_{\mathsf{v}} \mathsf{v} : \tau \qquad \tau \in \{\mathbb{1}, \mathbb{N}, \mathbb{B}\}}{\vdash_{\mathsf{v}} \mathsf{v} : \tau \triangleright \emptyset} \qquad \frac{}{\vdash_{\mathsf{v}} f : \tau \to \varphi \triangleright f : \tau \to \varphi}$$

$$\frac{\vdash_{\mathsf{v}} \mathsf{V} : \tau \triangleright \Gamma \qquad \vdash_{\mathsf{v}} \mathsf{W} : \varphi \triangleright \Delta}{\vdash_{\mathsf{v}} \langle \mathsf{V}, \mathsf{W} \rangle : \tau \times \varphi \triangleright \Gamma \cdot \Delta}$$

(a) abstraction of values.

$$\frac{\mathsf{v} \nearrow (\mathsf{V}; \gamma)}{\mathbf{ret}\, \mathsf{v} \nRightarrow (\mathbf{ret}\, \mathsf{V}; \gamma)} \qquad \frac{}{\iota\#\mathbf{op}\, \mathsf{v} \nRightarrow (e; [e \mapsto \iota\#\mathbf{op}\, \mathsf{v}])}$$

$$\frac{\vdash_{\mathsf{v}} \mathsf{V} : \tau \triangleright \Delta}{\Gamma \vdash_{\mathsf{c}} \mathbf{ret}\, \mathsf{V} : \tau \triangleright \Delta; \emptyset} \qquad \frac{}{\Gamma \vdash_{\mathsf{c}} e :? \triangleright \emptyset; e :?}$$

$$\frac{\Gamma \vdash_{\mathsf{c}} r[e] : \tau}{\Gamma \vdash_{\mathsf{c}} \kappa :: r[e] : \varphi \triangleright \emptyset; \kappa : \tau \rightsquigarrow \varphi}$$

(b) abstraction of generalised values.

Figure 6: nominal abstraction process.

generalized questions and answers structure.

$$\begin{array}{rcll} \mathsf{p}, \mathsf{q} & ::= & \langle \mathsf{C} \mid \Box \rangle & \textit{context-stuck} \\ & \mid & \langle \Box\, \mathsf{C} \mid d \rangle | \langle \Box\, \mathsf{V} \mid d \rangle & \textit{open-stuck} \end{array}$$

where $\langle r[e] \mid \Box \rangle$, $\langle \Box[r[e]] \mid d \rangle$ and $\langle \Box[\mathbf{ret}\, \mathsf{V}] \mid d \rangle$ are *syntactic sugar* for $\langle e \mid \Box \circ r \rangle$, $\langle e \mid d[\Box] \circ r \rangle$ and $\langle \mathbf{ret}\, \mathsf{V} \mid d[\Box] \rangle$ respectively, and where the *hole filling* is given by the application $(\mathsf{p}, \alpha) \mapsto \mathsf{p}[\alpha]$ where $\mathsf{p}[\alpha] := \mathsf{p}\{\Box := \alpha\}$.

In figure 7 we introduce the nominal abstraction of *copatterns*. Naturally, the abstraction process produces an *abstract copattern*, a name assignment γ, in addition to a component ξ, the *abstract forward*, carrying information about the *forward evaluation stack*. It is either of the shape $\varnothing$ (indicating the absence of effects) or (e, r, δ) (indicating a performed effect e to be forwarded or handled) where r represents the *abstract (delimited) stack* through which said effect has been propagated and δ is another name assignment for abstracting the effects and the surrounding evaluation contexts.

The type inference system similar to the one for generalised values in that it maintains two distinct name type contexts Γ and Δ where the first one contains function and undelimited continuation names, whereas the second one is only for effect and delimited continuation names.

Note that in this definition we use abstract evaluation contexts F that are either a stack of evaluation frames E or just a hole $[]$. The former case reflects the fact that only the *concrete* fragments of the evaluation contexts are abstracted, wheras the latter case reflects the fact that the abstraction of both $\mathsf{E}[\iota\#\mathbf{op}\, \mathsf{v}]$ and $\iota\#\mathbf{op}\, \mathsf{v}$ should result in a pattern of the form $\kappa[e]$ where κ represents either E or $[]$. Indeed, omitting κ when abstracting $\iota\#\mathbf{op}\, \mathsf{v}$ breaks the uniformity and carries an information (that the surrounding delimited continuation is the identity) that should not be disclosed.

$$\textsc{context-stuck}\ \frac{\mathsf{v} \nearrow (\mathsf{V}, \gamma_{\mathsf{V}})}{\langle \mathbf{ret}\, \mathsf{v} \mid \Box \rangle \nRightarrow (\langle \mathbf{ret}\, \mathsf{V} \mid \Box \rangle, \gamma_{\mathsf{V}}, \varnothing)}\ \textsc{return}$$

$$\textsc{perform}\ \frac{}{\langle \iota\#\mathbf{op}\, \mathsf{v} \mid \Box \circ \mathsf{F} \rangle \nRightarrow (\langle \kappa[e] \mid \Box \rangle, \varepsilon, (e, \kappa, [e \mapsto \iota\#\mathbf{op}\, \mathsf{v}] \cdot [\kappa \mapsto \mathsf{F}]))}$$

$$\textsc{forward}\ \frac{}{\langle e \mid \Box \circ \mathsf{F}[r] \rangle \nRightarrow (\langle \kappa :: r[e] \mid \Box \rangle, \varepsilon, (e, \kappa :: r, [\kappa \mapsto \mathsf{F}]))}$$

where $\mathsf{F} ::= [] \mid \mathsf{F}[\mathsf{E}]$

$$\textsc{open-stuck}\ \frac{\mathsf{v} \nearrow (\mathsf{V}, \gamma_{\mathsf{V}})}{\langle \Box\, \mathsf{v} \mid \mathsf{S} \rangle \nRightarrow (\langle \Box\, \mathsf{V} \mid c \rangle, \gamma_{\mathsf{V}} \cdot [c \mapsto \mathsf{S}], \varnothing)}$$

$$\frac{\mathsf{P} \nRightarrow (\mathsf{p}, \gamma, \xi)}{\mathsf{C}[\mathsf{S}[\Box]] \nRightarrow (\mathsf{p}[c[\Box]]], \gamma \cdot [c \mapsto \mathsf{S}], \xi)}$$

(a) abstracting copatterns

$$\textsc{context-stuck}\ \frac{\Gamma \vdash_{\mathsf{c}} \mathsf{C} : \tau \triangleright \Gamma_{\mathsf{C}}; \Delta_{\mathsf{C}}}{\Gamma \vdash_{\mathsf{p}} \langle \mathsf{C} \mid \Box \rangle : \neg\neg\tau \triangleright \Gamma_{\mathsf{C}}; \Delta_{\mathsf{C}}}$$

$$\textsc{open-stuck}\ \frac{\Gamma \vdash_{\mathsf{c}} \mathsf{C} : \tau \triangleright \Gamma_{\mathsf{C}}; \Delta_{\mathsf{C}}}{\Gamma \vdash_{\mathsf{p}} \langle \Box\, \mathsf{C} \mid d \rangle : \neg(\tau \rightsquigarrow \varphi) \triangleright \Gamma_{\mathsf{C}}, d : \neg\varphi; \Delta_{\mathsf{C}}}$$

$$\frac{\vdash_{\mathsf{v}} \mathsf{V} : \tau \triangleright \Gamma_{\mathsf{V}}}{\Gamma \vdash_{\mathsf{p}} \langle \Box\, \mathsf{V} \mid d \rangle : \neg(\tau \to \varphi) \triangleright \Gamma_{\mathsf{V}}, d : \neg\varphi; \emptyset}$$

(b) copatterns typing rules.

Figure 7: nominal abstraction of copatterns.

Example 3.1. Perhaps the simplest example is a fully-evaluated returner in the form of $c[\mathbf{ret}\, \mathsf{v}]$ that can be written as $(\Box\, \mathbf{ret}\, \mathsf{v})(c)$ and whose abstraction can be expressed as an abstract copattern $\mathsf{p} = \langle \mathbf{ret}\, \mathsf{V} \mid \Box \rangle$ and the abstract codata c.

Remark 1. *Notice how the abstraction process of normal forms is not total. It is only defined on effectful normal forms of a specific shape (involving the evaluation contexts* F*) that reflects the order of effect propagation and the incrementation of the* abstract forward *mirrors this gradual interactive process. The choice of keeping* γ *and* δ *(of* ξ*) and to combine abstract codata belonging to both the program and the environment in* r *has a technical benefit as well as a conceptual one; as it maintains the distinction between passive interactions and interactions with observation.*

3.2 Interaction LTS

In the *abstract interaction*, the executing term (Proponent) will be represented by an LTS configuration in order to be evaluated against

an *abstract environment* (Opponent) that models all compatible *program environments* while exhibiting the *interaction* between the two that gets obscured by plain *syntax substitution*. To this effect, the configurations will carry an *information* component that keeps track of the history of the *interaction*. It consists of a map between the names and the *concrete* code of codata that Proponent has disclosed and rendered accessible to Opponent.

We define next the abstract interactive LTS $\mathcal{L}_{\mathsf{AI}}$ that is given by $(\mathcal{A}, \mathcal{M}, \xrightarrow{\mathbf{m}}_{\mathsf{I}})$.

3.2.1 configurations, moves and transitions.

states	$\mathcal{A}$	$\ni$	$\mathbb{I}, \mathbb{J}$	$::=$	$\langle \mathtt{I};\, \gamma;\, ⧈\, \xi \rangle$	*passive state*
				$\mid$	$\langle (\mathtt{M} \mid \mathtt{I});\, \gamma;\, ⧈\, \delta \rangle$	*active state*
moves	$\mathcal{M}$	$\ni$	$\mathbf{m}$	$:=$	$\boldsymbol{\alpha}.\mathrm{p}^{\oplus} \mid \boldsymbol{\alpha}.\mathrm{p}^{\ominus}$	

The moves are given by the previously described *abstract copatterns* where the action $\boldsymbol{\alpha}.\mathrm{p}$ is to be understood as the combination of an *interaction handle* α and an input p: the abstract codata α belonging to the *passive* player is being probed with the *active* player's abstract copattern p.

The superscript $\oplus$ (resp. $\ominus$) is there to indicate that Proponent (resp. Opponent) is the *active* player performing the *move*. We will sometimes range over these polarized moves by **p** and **o** respectively. Breaking down all the possible *interactive transitions* witnessed by $\boldsymbol{\alpha}.\mathrm{p}$, we have:

- $\boldsymbol{c}.\langle \mathbf{ret}\, \mathtt{V} \mid \Box \rangle$ returning a value to the abstract context $c[]$.
- $\boldsymbol{c}.\langle r[e] \mid \Box \rangle$ propagating e and the delimiting continuation r to the abstract context $c[]$.
- $\boldsymbol{f}.\langle \Box\, \mathtt{V} \mid d \rangle$ requesting the result of the application $f\, \mathtt{V}$ in $d[]$.
- $\boldsymbol{\kappa}.\langle \Box[\mathbf{ret}\, \mathtt{V}] \mid d \rangle$ requesting the result of the computation $\kappa[\mathbf{ret}\, \mathtt{V}]$ in $d[]$.
- $\boldsymbol{\kappa}.\langle \Box[r[e]] \mid d \rangle$ delegating the handling or propagation of e with its delimiting continuation r to κ and requesting the result in $d[]$.

Putting a modality $⧈ \in \{\mathbf{!}, \mathbf{?}\}$ in front of the abstract forward ξ indicates who is the author of the effect. Accordingly, $\mathbf{!}(e, \_, \delta)$ means that Proponent performed the effect and that Opponent *can only forward* it, while $\mathbf{?}\delta$, on the other hand, reads as Opponent *may forward* the effect.

Note that is only a convenient notation since the former case would always be witnessed by the fact that $e \in \mathrm{dom}(\delta)$ and the latter by $\mathrm{dom}(\delta) \cap \mathcal{E} = \emptyset$. For this reason we will often omit writing the modality. We will also omit the abstract forward altogether from configurations when it is equal to $\emptyset$, *i.e.* $\langle \mathtt{I}; \gamma \rangle$ (resp $\langle (\mathtt{M} \mid \mathtt{I}); \gamma \rangle$) is syntactic sugar for $\langle \mathtt{I}; \gamma; \emptyset \rangle$ (resp $\langle (\mathtt{M} \mid \mathtt{I}); \gamma; \emptyset \rangle$). For a compact presentation, we will abuse the notation of maps and substitution on ξ when it is the constituting component δ that is intended instead.

After a series of internal reduction steps $\mapsto^{*}_{\mathrm{op}}$, the *active* configuration reaches a *normal form*. It performs a transition into an *passive state* where the *codata* in the corresponding copattern is *abstracted* then disclosed or forwarded to the environment via the information components. The underlying abstraction process is defined in figure 8, where $\mathbf{abstract}(\mathtt{P}[\alpha], \xi) \nearrow\!\!\nearrow (\boldsymbol{\alpha}.\mathrm{p}, \gamma, \xi')$ is to be read as **abstract** taking a decomposed normal form and an effect

$$\textsc{pure}\quad \frac{\mathtt{P} \nearrow\!\!\nearrow (\mathrm{p}, \gamma_{\mathrm{p}}, \emptyset)}{\mathbf{abstract}(\mathtt{P}[\alpha], \emptyset) \nearrow\!\!\nearrow (\boldsymbol{\alpha}.\mathrm{p}, \gamma_{\mathrm{p}}, \emptyset)}$$

$$\textsc{perform}\quad \frac{\langle \iota\#\mathbf{op}\, \mathtt{v} \mid \mathtt{S}[\Box] \circ \mathtt{F} \rangle \nearrow\!\!\nearrow (\mathrm{p}, \gamma_{\mathrm{p}}, \xi)}{\mathbf{abstract}(\langle \iota\#\mathbf{op}\, \mathtt{v} \mid \mathtt{S}[\alpha[]] \circ \mathtt{F} \rangle, \emptyset) \nearrow\!\!\nearrow (\boldsymbol{\alpha}.\mathrm{p}, \gamma_{\mathrm{p}}, \xi)}$$

$$\textsc{forward}\quad \frac{\langle e \mid \mathtt{S}[\Box] \circ \mathtt{F}[r] \rangle \nearrow\!\!\nearrow (\mathrm{p}, \gamma_{\mathrm{p}}, \xi)}{\mathbf{abstract}(\langle \mathtt{e} \mid \mathtt{S}[\alpha[]] \circ \mathtt{F}[\mathtt{S}] \rangle, (e, r, \_)) \nearrow\!\!\nearrow (\boldsymbol{\alpha}.\mathrm{p}, \gamma_{\mathrm{p}}, \xi)}$$

(a) abstracting normal forms

$$\textsc{pure/perform}\quad \frac{}{\mathbf{concretize}(\boldsymbol{\alpha}.\mathrm{p}, \gamma, \emptyset) = \mathrm{p}[\gamma(\alpha)]}$$

$$\textsc{forward}\quad \frac{\xi = (e, r, \delta) \qquad \mathrm{p} = \langle \kappa :: r[e] \mid \_ \rangle \qquad \mathtt{P} = \mathrm{p}\{\delta\}}{\mathbf{concretize}(\boldsymbol{\alpha}.\mathrm{p}, \gamma, \xi) = \mathtt{P}[\gamma(\alpha)]}$$

(b) concretizing abstract normal forms

Figure 8: the dualizing process.

forward component ξ and producing the next action $\boldsymbol{\alpha}.\mathrm{p}$ and the corresponding information components γ, ξ'.

A transition from a *passive configuration* admits *branching*. Indeed, since the *program environment* is abstract and not fixed, it can use any of the disclosed information to perform a *move* triggering a transition into an *active configuration* and the *concretization* of a running computation. This process is given by the function **concretize** defined in figure 8, which takes as input an action $\boldsymbol{\alpha}.\mathrm{p}$, the disclosed information γ and ξ in order to output the corresponding running computation shaped $\mathtt{P}[\gamma(\alpha)]$.

The transitions of $\mathcal{L}_{\mathsf{AI}}$ are defined in figure 9. The underlying relation $\rightarrow_{\mathsf{I}}$ is split into pure and effectful transitions for clarity of presentation. The effectful transitions are described by the relation $\rightarrow_{\mathbf{fwd}}$, defined in figures 9d and 9e. These figures describe the entire effectful interaction: from internal effect propagation and eventual internal effect handling (figure 9d), to the point where control returns to the pure world. The underlying effectful trace s produced by $\rightarrow_{\mathbf{fwd}}$ is then used to define the effectful case of $\rightarrow_{\mathsf{I}}$ via the rule Forward (figure 9c).

Remark 2. *As mentioned in section 2.2 where we have introduced* $\mapsto_{\mathrm{op}}$*, the reduction* $\mapsto_{\mathrm{fwd}}$ *captures a flow of information; that of effect propagation. A series of* $\mapsto_{\mathrm{fwd}}$ *reductions not followed by an* $\mapsto_{\mathrm{eval}}$ *reduction results in an unhandled effect forwarded to the environment, which in turn could potentially handle it, capturing the associated delimited continuation in the process.*

This information flow is a form of interaction between the program and its environment consisting in inquiring whether the environment handles the effect or not. The abstract effect and its enclosing evaluation context are provisionally forwarded as a fragment of the would-be captured delimited continuation in case the environment catches the effect and handles it. In case it does not, a pseudo copy-cat action

$$\textsc{Eval}\ \frac{(\mathtt{M} \mid \mathtt{I}) \mapsto_{\mathrm{eval}} (\mathtt{N} \mid \mathtt{I}')}{\langle (\mathtt{M} \mid \mathtt{I});\ \gamma\rangle \xrightarrow{\mathrm{eval}}_{\mathsf{I}} \langle (\mathtt{N} \mid \mathtt{I}');\ \gamma\rangle}$$

(a) Internal evaluation

$$\textsc{Px}\ \frac{\mathbf{abstract}(\mathtt{Nf}, \varnothing) \nearrow\!\!\!\nearrow (\boldsymbol{\alpha}.\mathrm{p},\ \gamma_{\mathrm{p}})}{\langle (\mathtt{Nf} \mid \mathtt{I});\ \gamma\rangle\text{`} \xrightarrow{\boldsymbol{\alpha}.\mathrm{p}^{\oplus}}_{\mathsf{I}} \langle \mathtt{I};\ \gamma{\cdot}\gamma_{\mathrm{p}}\rangle}$$

$$\textsc{Ox}\ \frac{}{\langle \mathtt{I};\ \gamma\rangle \xrightarrow{\boldsymbol{\alpha}.\mathrm{p}^{\ominus}}_{\mathsf{I}} \langle (\mathtt{p}[\gamma(\alpha)] \mid \mathtt{I});\ \gamma\rangle}$$

(b) Pure moves

$$\textsc{Forward}\ \frac{\mathbb{I} \overset{\mathrm{s}}{\Rightarrow}_{\mathbf{fwd}} \mathbb{J}}{\mathbb{I} \overset{\mathrm{s}}{\Rightarrow}_{\mathsf{I}} \mathbb{J}}$$

(c) Effect forwarding sequences

$$\textsc{Forward}\ \frac{(\mathtt{M} \mid \mathtt{I}) \mapsto_{\mathrm{fwd}} (\mathtt{N} \mid \mathtt{I}')}{\langle (\mathtt{M} \mid \mathtt{I});\ \gamma;\ \xi\rangle \rightarrow_{\mathbf{fwd}} \langle (\mathtt{N} \mid \mathtt{I}');\ \gamma;\ \xi\rangle}$$

$$\textsc{PHandle}\ \frac{(\mathtt{M} \mid \mathtt{I}) \mapsto_{\mathrm{eval}}}{\langle (\mathtt{M} \mid \mathtt{I});\ \gamma;\ \mathbf{!}\delta\rangle \xrightarrow{hdl^{\oplus}}_{\mathbf{fwd}} \langle (\mathtt{M}\{\gamma\} \mid \mathtt{I});\ \gamma\rangle}$$

$$\frac{}{\langle \mathtt{I};\ \gamma;\ \mathbf{?}\xi\rangle \xrightarrow{hdl^{\ominus}}_{\mathbf{fwd}} \langle \mathtt{I};\ \gamma{\cdot}\xi\rangle}\ \textsc{OHandle}$$

(d) Internal effect manipulation

$$\textsc{Peff}\ \frac{\mathbf{abstract}(\mathtt{Nf},\ \delta) \nearrow\!\!\!\nearrow (\boldsymbol{\alpha}.\mathrm{p},\ \gamma_{\mathrm{p}},\ \xi)}{\langle (\mathtt{Nf} \mid \mathtt{I});\ \gamma;\ \delta\rangle \xrightarrow{\boldsymbol{\alpha}.\mathrm{p}^{\oplus}}_{\mathbf{fwd}} \langle \mathtt{I};\ \gamma{\cdot}\gamma_{\mathrm{p}};\ \xi\rangle}$$

$$\frac{\mathbf{concretize}(\boldsymbol{\alpha}.\mathrm{p},\ \gamma,\ \xi) = (\mathtt{M},\ \delta)}{\langle \mathtt{I};\ \gamma;\ \xi\rangle \xrightarrow{\boldsymbol{\alpha}.\mathrm{p}^{\ominus}}_{\mathbf{fwd}} \langle (\mathtt{M} \mid \mathtt{I});\ \gamma;\ \delta\rangle}\ \textsc{Oeff}$$

(e) Effectful moves

Figure 9: transitions of the Interaction LTS

(forwarding back the same effect but with an extended evaluation context) is expected instead.

By way of example, we will give an informal presentation of the *interaction* of a term with a *concrete* program context, which will map to one possible *interactive execution path* against an *abstract environment* in $\mathcal{L}_{\mathsf{Al}}$.

Example 3.2. We consider the interaction of the term $\mathtt{u} := \mathsf{let}\ y = \mathbf{new}\ \mathbb{E}\ \mathsf{in}\ \{f\ (\lambda x.y\#\mathbf{op}\ \langle\rangle)\}\ \mathbf{with}\ \mathtt{h}^{y}$, where $\mathtt{h}^{y} := \{\mathbf{ret}\ x \mapsto \mathbf{ret}\ x\} \uplus \{y\#\mathbf{op}\ x\ \kappa \mapsto \kappa\ 5\}$, with the program context $\mathsf{let}\ f = (\lambda y.y\ \mathtt{n};\ \mathbf{ret}\ \mathtt{tt})\ \mathsf{in}\ [\,]$, that is in turn equivalent to the ciu-environment $([\,],\ [f \mapsto \lambda g.g\ \mathtt{v};\mathbf{ret}\ \mathtt{tt}])$.

We embed the term $\mathtt{u}$ into the *active* configuration $\langle d_0[\mathtt{u}],\ \varepsilon\rangle$. Moreover, we consider the *passive* configuration $\langle \gamma_o\rangle$ representing the ciu-environment, where $\gamma_o = [d_0 \mapsto c_{\mathtt{f}}[\,]] \cdot [f \mapsto \lambda x.x\ \mathtt{v};\ \mathbf{ret}\ \mathtt{tt}]$.

Reading from top to bottom, figure 10 illustrates both the concrete interaction and its corresponding path in the abstract interaction LTS. In this example, $\Gamma \Vdash \mathtt{t}$ indicates that the active player, represented bt $\mathtt{t}$ is being evaluated in the environment given by Γ (we write only names for simplicity). Horizontal lines annotate the interaction points with the corresponding move.

At each one of these points, the active player makes a move and then swaps roles with the passive player. The codata α associated with each move is highlighted as $\boxed{\alpha}$. Concrete codata and their corresponding names are colored identically to indicate their association.

3.3 Typing Constraints

Definition 3.3. Type configurations $\mathbb{T}; \mathbb{S}$ are of the shape $\langle \Gamma_o; \Delta_o \mid \Gamma_p; \Delta_p\rangle$ where:

- Γ_p, Γ_o are two *disjoint* type contexts for *Program names* and *Environment names*, respectively.
- Δ_p, Δ_o are two *disjoint* type contexts for effects and forwarded *abstract delimited continuations*.
- For $y \in \{o, p\}$. Γ_y and Δ_y are disjoint.

The typing judgements $\mathbb{E} \wr \mathbb{T}$ and $\mathbb{P} \wr \mathbb{S}$ denotes that the configurations $\mathbb{E}$ and $\mathbb{P}$ are of type $\mathbb{T}$ and $\mathbb{S}$ respectively. We define the corresponding typing relation figure 11.

We will denote by $\mathbb{T}^{\perp}$ the type configuration dual to $\mathbb{T}$ defined by:

$$\langle \Gamma_o; \Delta_o \mid \Gamma_p; \Delta_p\rangle^{\perp} := \langle \Gamma_p; \Delta_p \mid \Gamma_o; \Delta_o\rangle$$

The *Type LTS* $\mathcal{L}_{\mathcal{T}}$ is defined as $(\mathcal{T}; \mathcal{M}; \xrightarrow{\mathbf{m}}_{\mathcal{T}})$ with $\xrightarrow{\mathbf{m}}_{\mathcal{T}}$ defined in figure 12, where $\mathcal{T}$ is the set of type configurations.

3.4 Well-bracketing Constraints

The interactive configurations $\mathbb{I} \in \mathcal{A}$ we have considered thus far represent Proponent's one-sided perspective on the computation. The *information component* contains all of its *codata* that had been disclosed up to the current stage of the computation, namely the *continuations*. However, this component (and by extent any *passive configuration*), as it is defined, is history-agnostic whereas the language Λ_{eff} only permits a constrained usage of these *abstract continuations* that respects a certain history-sensitive *bracketing* discipline.

In this section, we define the well-bracketing LTS, which addresses this issue by imposing constraints on Opponent's moves. These constraints ensure that the resulting well-bracketed behaviours correspond to concrete, implementable interactions. As noted in the introduction, our notion of well-bracketing relaxes the standard one: although it allows arbitrary invocations of captured delimited continuations, each such invocation must still respect a stack discipline that governs the underlying delimited continuations.

3.4.1 The control-flow of delimited continuations. When an effect is performed by one *Player* and gets propagated beyond its immediate controlled scope, it initiates a sequence of effectful moves until the effect is handled or reaches *top-level*.

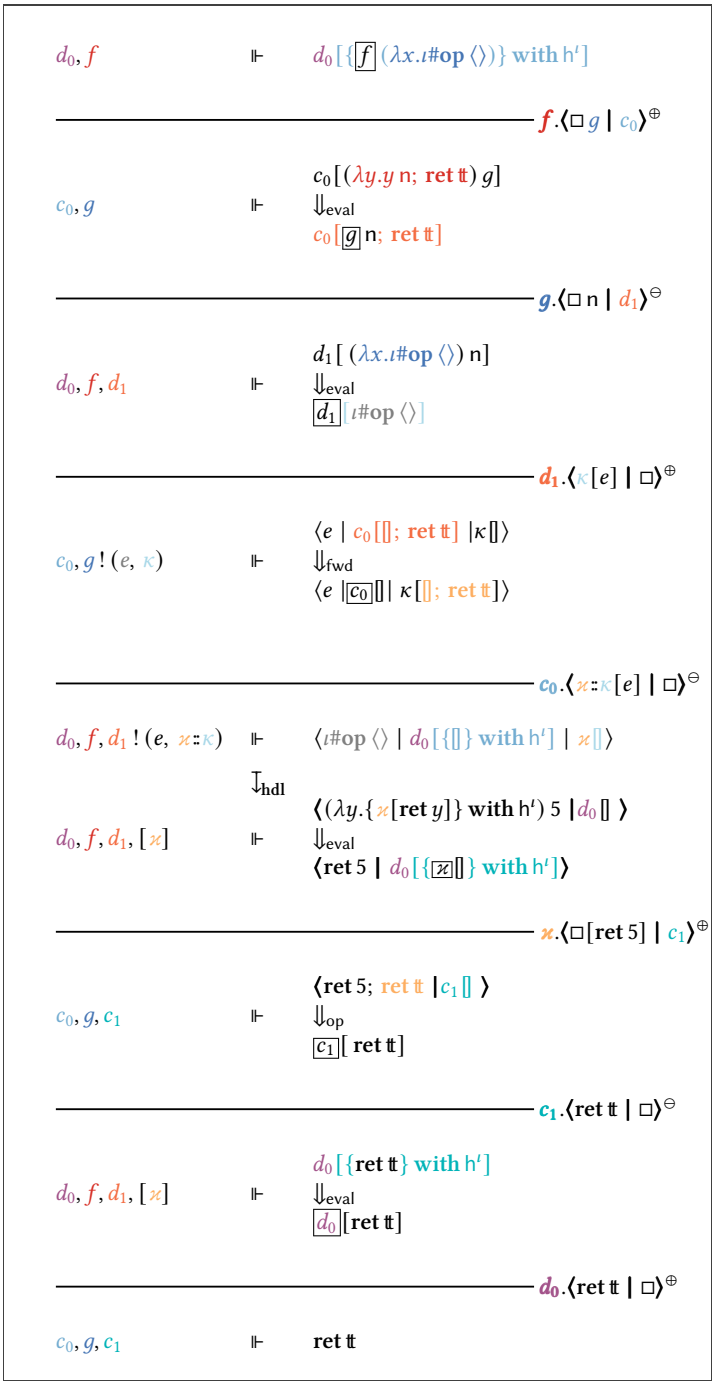


Figure 10: effectful interaction of u and γ_o.

PASSIVE CONF.

$$\frac{\mathrm{dom}(\gamma) = \Gamma_p \qquad \Gamma_o \vdash \gamma : \Gamma_p \qquad \Gamma_p \vdash \xi : \Delta_o}{\langle \mathtt{I};\ \gamma;\ \xi\rangle \Subset \langle \Gamma_o;\ \Delta_o \mid \Gamma_p;\ \Delta_p\rangle}$$

ACTIVE CONF.

$$\frac{\mathrm{dom}(\gamma) = \Gamma_p \qquad \mathtt{I}; \Gamma_o \cdot \Delta_o \Vdash_{\mathtt{c}} \mathtt{M} \qquad \Gamma_o \vdash \delta : \Delta_p}{\langle (\mathtt{M} \mid \mathtt{I});\ \gamma;\ \delta\rangle \Subset \langle \Gamma_o;\ \Delta_o \mid \Gamma_p;\ \Delta_p\rangle}$$

Figure 11: configuration typing.

PACTION

$$\frac{\Gamma_o(\alpha) = \tau \qquad \Gamma_o \cdot \Delta_p \cdot \Delta_o \vdash_{\mathtt{p}} \mathtt{q} : \neg\tau \triangleright \Gamma_{\mathtt{q}}; \Delta_{\mathtt{q}}}{\langle \Gamma_o;\ \Delta_o \mid \Gamma_p;\ \Delta_p\rangle \xrightarrow{\boldsymbol{\alpha}.\mathtt{q}^{\oplus}}_{\mathcal{T}} \langle \Gamma_o;\ \Delta_o \mid \Gamma_p \cdot \Gamma_{\mathtt{q}};\ \Delta_p \cdot \Delta_{\mathtt{q}}\rangle}$$

PHANDLE

$$\frac{}{\langle \Gamma_o;\ \Delta_o \mid \Gamma_p;\ \Delta_p\rangle \xrightarrow{hdl^{\oplus}}_{\mathcal{T}} \langle \Gamma_o \cdot \Delta_o;\ \emptyset \mid \Gamma_p;\ \emptyset\rangle}$$

(a) proponent transitions.

$$\frac{\mathbb{T} \xrightarrow{\mathtt{m}}_{\mathcal{T}} \mathbb{S}}{\mathbb{T}^{\perp} \xrightarrow{\mathtt{m}^{\perp}}_{\mathcal{T}} \mathbb{S}^{\perp}}$$

(b) symmetrical transitions.

Figure 12: Definition of $\mathcal{L}_{\mathcal{T}}$

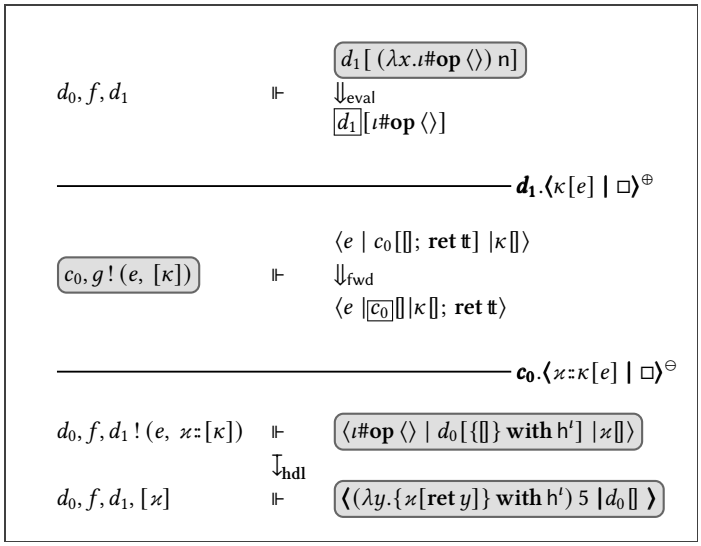


Figure 13: effectful forward trace.

To visualise this part of the interaction, we will zoom in at the computation underlying it in the concrete example of 3.2.

In figure.13, we only highlight the player responsible for triggering the effect (in its active and passive states). The resulting effectful sequence is $d_1.\langle \kappa[e] \mid \Box\rangle^{\oplus} c_0.\langle \varkappa :: \kappa[e] \mid \Box\rangle^{\ominus}$, which can be understood as a *pseudo-copycat* sequence in which Opponent forwards the computation $\langle e \mid d_1[]\ \circ\ \kappa[]\ \rangle$ as is to the abstract continuation $c_0[]$.

Now if we wish to generalize this sequence to one of arbitrary length n, $\mathbb{I} \overset{s}{\Rightarrow}_{\mathbf{fwd}} \mathbb{J}$, the generated configuration will always be of the following shape:

$$\frac{\mathbb{I} \overset{s}{\Rightarrow}_{\mathbf{fwd}} \mathbb{J} \qquad \mathbb{J} \xrightarrow{\mathbf{hdl}}_{\mathbf{fwd}}}{\mathbb{J} = \langle \iota \# \mathbf{op}\, \mathsf{v} \mid \mathsf{T}[\{[]\}\ \mathbf{with}\ \mathsf{h}] \circ \mathsf{S}_{n+1}[\kappa_n[\mathsf{S}_n[\cdots \kappa_0[\mathsf{S}_0]\cdots]]] \mid \mathtt{I} \rangle}$$

Consequently, the captured continuation will be of the shape:

$$\lambda x.(\{\mathsf{S}_{n+1}[\kappa_n[\mathsf{S}_n[\cdots \kappa_0[\mathsf{S}_0[\mathbf{ret}\, x]]]\cdots]]\}\ \mathbf{with}\ \mathsf{h})$$

In the case the continuation is discarded and not used, the control-flow stack remains intact. Otherwise, every call to the captured delimited continuation entails exactly one query addressed to each one of the n environment controlled fragments; *i.e.* a call to each of the abstract evaluation stacks κ_i in the order in which they have been disclosed.

We define the well-bracketing LTS as $\mathcal{L}_{wb} := (\mathcal{W}, \mathcal{M}, \rightarrow_{wb})$ where the configurations are described below and the transitions $\xrightarrow{\mathbf{m}}_{wb}$ are defined in figure 14.

effect forwarding information

η	$::=$	$\varnothing$	*absence of effect*
	$\mid$	$(f,\ e)$	*effect e forwarded through f*

configurations

$\mathbb{W}, \mathbb{U}$	$:=$	$\langle \sigma \mid \eta \mid \phi \rangle$

where $\sigma \in \mathtt{List}(\mathcal{C} \cup \mathcal{K})$, $f \in \mathcal{F} = \mathtt{List}(\mathcal{K})$ denote a *control-flow stack* and a *call frame*, respectively, and $\phi \in \wp(\mathcal{F})$ denotes the set of all captured call frames.

Remark 3. *In Λ_{eff}, arbitrary handlers can be defined as no restrictions are imposed on the usage of delimited continuations. This is reflected semantically in Opponent's transitions in $\mathcal{L}_{wb}$, which permit unrestricted use of captured delimited continuation, thus allowing multi-shot continuations and re-entrant continuation calls.*

In this setting, calling any captured delimited continuations only amounts to querying the top delimited continuation name κ of the corresponding call frame $\kappa :: f$ and stacking the remainder f on top of the control flow stack.

3.5 The OGS LTS

The OGS LTS $(\mathcal{O}, \mathcal{M}, \rightarrow_{\text{ogs}})$ is the transition system that can generate the interaction traces[3] compatible with what our language $(\Lambda_{\text{eff}}, \mapsto_{\text{op}})$ permits.

It is defined as the product of $\mathcal{L}_{\mathsf{AI}}$, $\mathcal{L}_{\mathcal{T}}$ and $\mathcal{L}_{wb}$ where the underlying labelled transitions systems are synchronised on $\mathcal{M} \cup \{hdl^{\oplus}, hdl^{\ominus}\}$. The configurations in $\mathcal{O}$ satisfy some conditions that guarantee the well-typedness of $\mathcal{L}_{\mathsf{AI}}$; in that Opponent and Proponent agree on types during their interaction, in addition to ensuring that Opponent's use of continuations is well-bracketed.

In the following we will denote $\langle \mathbb{I} \parallel \mathbb{T} \parallel \mathbb{W} \rangle \in \mathcal{A} \times \mathcal{T} \times \mathcal{W}$ any triple of configurations satisfying $\mathbb{I} \wr \mathbb{T}$ and $\mathbf{Compat}(\mathbb{I}, \mathbb{W})$, where the second proposition morally means that the control-flow history in $\mathbb{W}$ is *compatible* with the interactive information in $\mathbb{I}$. Due to lack of space, the formal definition of this predicate is deferred to the section D.2 of the appendix.

[3]For a reader familiar with *game semantics*, this LTS specifies the *plays*.

$$\begin{array}{lcl}
\mathbf{forward}(\varnothing, \boldsymbol{c}.\ \langle e \mid \square \circ \varkappa \rangle^{\ominus}) & := & (e,\ []) \\
\mathbf{forward}(\varnothing, \boldsymbol{\kappa}.\ \langle e \mid d[\square] \circ \varkappa \rangle^{\ominus}) & := & (e,\ []) \\
\mathbf{forward}(\eta,\ \mathbf{o}) & := & \eta
\end{array}$$

$$\begin{array}{lcl}
\mathbf{call/ret}(\alpha :: \sigma,\ _,\ \boldsymbol{\alpha}.\mathrm{p}^{\ominus}) & := & \sigma \\
\mathbf{call/ret}(\sigma,\ \phi,\ \boldsymbol{\kappa}.\ \langle e \mid d[\square] \circ r \rangle^{\ominus}) & := & f \Cup \sigma \\
\mathbf{call/ret}(\sigma,\ \phi,\ \boldsymbol{\kappa}.\ \langle \mathbf{ret}\, \mathsf{V} \mid d[\square] \rangle^{\ominus}) & := & f \Cup \sigma \\
\quad \text{when} \quad \kappa :: f \in \phi & &
\end{array}$$

$$\frac{\eta' = \mathbf{forward}(\eta,\ \mathbf{o}) \qquad \sigma' = \mathbf{call/ret}(\sigma,\ \phi,\ \mathbf{o})}{\langle \sigma \mid \eta \mid \phi \rangle \xrightarrow{\mathbf{o}}_{wb} \langle \sigma' \mid \eta' \mid \phi \rangle}$$

$$\frac{}{\langle \sigma \mid (e,\ f) \mid \phi \rangle \xrightarrow{hdl^{\ominus}}_{wb} \langle \sigma \mid \varnothing \mid \phi \cup \{f\} \rangle}$$

(a) opponent transitions

$$\begin{array}{lcl}
\mathbf{forward}((e,\ f), \boldsymbol{d}.\ \langle e \mid \square \circ \kappa :: r \rangle^{\oplus}) & := & (f \Cup [\kappa],\ e) \\
\mathbf{forward}((e,\ f), \boldsymbol{\varkappa}.\ \langle e \mid c[\square] \circ \kappa :: r \rangle^{\oplus}) & := & (f \Cup [\kappa],\ e) \\
\mathbf{forward}(\varnothing,\ \mathbf{p}) & := & \varnothing
\end{array}$$

$$\begin{array}{lcl}
\mathbf{call/ret}(\sigma,\ _,\ \boldsymbol{\varkappa}.\ \langle \mathbf{ret}\, \mathsf{V} \mid c[\square] \rangle^{\oplus}) & := & c :: \sigma \\
\mathbf{call/ret}(\sigma,\ _,\ \boldsymbol{\varkappa}.\ \langle e \mid c[\square] \circ _ \rangle^{\oplus}) & := & c :: \sigma \\
\mathbf{call/ret}(\sigma,\ _,\ \boldsymbol{f}.\ \langle \square\, \mathsf{V} \mid c \rangle^{\oplus}) & := & c :: \sigma \\
\mathbf{call/ret}(\sigma,\ _,\ \mathbf{p}) & := & \sigma
\end{array}$$

$$\frac{\sigma' = \mathbf{call/ret}(\sigma,\ \mathbf{p}) \qquad \eta' = \mathbf{forward}(\eta,\ \mathbf{p})}{\langle \sigma \mid \eta \mid \phi \rangle \xrightarrow{\mathbf{p}}_{wb} \langle \sigma' \mid \eta' \mid \phi \rangle}$$

$$\frac{}{\langle \sigma \mid \varnothing \mid \phi \rangle \xrightarrow{hdl^{\oplus}}_{wb} \langle \sigma \mid \varnothing \mid \phi \rangle}$$

(b) proponent transitions

Figure 14: $\mathcal{L}_{wb}$ transitions.

3.5.1 Embedding the interactive language. Finally, we define an interactive term embedding into an initial *active* $\mathcal{L}_{\text{OGS}}$ configuration and ciu-environment embedding into an initial *passive* $\mathcal{L}_{\text{OGS}}$ configuration.

Definition 3.4 (term embedding). Given an interactive term $\mathtt{I}; \Gamma \Vdash_{\mathsf{c}} \mathsf{M}$, we define its embedding

$$\langle \mathtt{I}; \Gamma \Vdash_{\mathsf{c}} \mathsf{M} \rangle^{\oplus}_{\text{ogs}} := \langle \langle (\mathsf{M} \mid \mathtt{I});\ \varepsilon \rangle \parallel \langle \Gamma; \emptyset \mid c_{\mathsf{f}} : \neg\mathbb{1}; \emptyset \rangle \parallel \langle [d];\ \varnothing;\ \emptyset \rangle \rangle$$

with d the unique continuation name of Γ.

Definition 3.5 (environment embedding). Given a name assignment $\mathtt{I}; c_{\mathsf{f}} : \neg\mathbb{1} \vdash \gamma : \Gamma$, we define its embedding

$$\langle \mathtt{I}; c_{\mathsf{f}} : \neg\mathbb{1} \vdash \gamma : \Gamma \rangle^{\ominus}_{\text{ogs}} := \langle \langle \emptyset; \gamma \rangle \parallel \langle c_{\mathsf{f}} : \neg\mathbb{1}; \emptyset \mid \Gamma; \emptyset \rangle \parallel \langle [c_{\mathsf{f}}];\ \varnothing;\ \emptyset \rangle \rangle$$

4 The OGS model

4.1 Interpretation of expressions

Following our equivalence of programs introduced in definition 2.4, we only observe convergence, *i.e.* a full evaluation into a value returner of the shape **ret** v. We introduce a notion of *complete trace* in order to capture this *complete* evaluation.

Definition 4.1 (complete traces). : A trace $\mathtt{t}\,\mathbf{m} \in \mathbf{Tr}_{\mathrm{ogs}}$ is said t be *complete* and written $\mathcal{CP}(\mathtt{t}\,\mathbf{m})$ when:

$$\mathcal{CP}(\mathtt{t}\,\mathbf{m}) := \begin{cases} \langle [] \mid \varnothing \mid \emptyset \rangle \overset{\mathtt{t}\,\mathbf{m}}{\Longrightarrow}_{wb} \langle [] \mid \eta \mid \phi \rangle \ \text{ for some } \eta \text{ and } \phi \\ \mathbf{m} = \boldsymbol{c}.\langle \mathbf{ret}\ \mathtt{V} \mid \square \rangle^{\ominus} \ \text{ for some } \mathtt{V} \text{ and } c. \end{cases}$$

Accordingly, we define the set of complete *traces* of a configuration:

$$\mathbf{CTr}_{\mathrm{ogs}}(\mathbb{G}) := \{\mathtt{t} \in \mathbf{Tr}_{\mathrm{ogs}}(\mathbb{G}) \mid \mathcal{CP}(\mathtt{t})\}$$

4.2 Soundness of the model

Theorem 4.2 (Adequacy). *Given an interactive term* $\mathtt{I};\Gamma \Vdash_{\mathtt{c}} \mathtt{M}$ *and a compatible name assignment* $\mathtt{I}';c_{\mathtt{f}} : \neg\mathbb{1} \vdash \gamma : \Gamma$*, then by writing* $\mathbb{P}$ *and* $\mathbb{E}$ *for their respective* ogs *embeddings, we have:*

$$\exists \mathtt{I}'' \supseteq \mathtt{I}'\cdot\mathtt{I}.\ (\mathtt{M}\{\gamma\} \mid \mathtt{I}'\cdot\mathtt{I}) \Downarrow_{\mathrm{op}} (c_{\mathtt{f}}[\,\mathbf{ret}\,\langle\rangle] \mid \mathtt{I}'')$$
$$\Longleftrightarrow$$
$$\exists \mathtt{t} \in \mathbf{CTr}_{\mathrm{ogs}}(\mathbb{P}).\ \mathtt{t}^{\perp}\,\boldsymbol{c_{\mathtt{f}}}.\langle \mathbf{ret}\,\langle\rangle \mid \square\rangle^{\oplus} \in \mathbf{CTr}_{\mathrm{ogs}}(\mathbb{E})$$

The proof of the adequacy theorem 4.2 is detailed in Appendix section B. It relies on the introduction of another LTS (*cf.* Appendix section A) that formalizes the notion of *concrete interaction* from example 3.2. This allows us to define semantic notions of composition and observation (at the level of configurations and complete traces), which are then shown, again in the appendix in sections C and D, to coincide with their syntactic/operational counterparts.

Theorem 4.3 (Soundness). *Taking* $\mathtt{M}_1, \mathtt{M}_2$ *two interactive terms such that both* $\mathtt{I};\Gamma \Vdash_{\mathtt{c}} \mathtt{M}_i$ *(for* $i \in \{1,2\}$*), suppose that* $\mathbf{CTr}(\langle \mathtt{I};\Gamma \vdash \mathtt{M}_1\rangle) \subseteq \mathbf{CTr}(\langle \mathtt{I};\Gamma \vdash \mathtt{M}_2\rangle)$*. Then* $\Gamma \vdash \mathtt{M}_1 \preceq_{ciu} \mathtt{M}_2$.

Proof. We take a continuation name $c_{\mathtt{f}}$, an instance context $\mathtt{I}' \supseteq \mathtt{I}$ and a name assignment γ such that $\mathtt{I}';c_{\mathtt{f}} : \neg\mathbb{1} \vdash \gamma : \Gamma$. We write $\mathbb{G}_{\mathrm{P},i}$ for $\langle \mathtt{I};\Gamma \Vdash_{\mathtt{c}} \mathtt{T}_i\rangle^{\oplus}_{\mathrm{ogs}}$, and $\mathbb{G}_{\mathrm{O}}$ for $\langle \mathtt{I}';c_{\mathtt{f}} : \neg\mathbb{1} \vdash \gamma : \Gamma\rangle^{\ominus}_{\mathrm{ogs}}$.

Suppose that $(\mathtt{M}_1\{\gamma\} \mid \mathtt{I}') \Downarrow_{\mathrm{op}} c_{\mathtt{f}}[\langle\rangle]$. Then from theorem 4.2, we get the existence of a complete trace $\mathtt{t} \in \mathbf{CTr}(\mathbb{G}_{\mathrm{P},1})$ such that $\mathtt{t}_1^{\perp}\boldsymbol{c_{\mathtt{f}}}.\langle \mathbf{ret}\,\langle\rangle \mid \square\rangle^{\oplus} \in \mathbf{CTr}(\mathbb{G}_{\mathrm{O}})$. From $\mathbf{CTr}(\mathbb{G}_{\mathrm{P},1}) \subseteq \mathbf{CTr}(\mathbb{G}_{\mathrm{P},2})$, we get that $\mathtt{t} \in \mathbf{CTr}(\mathbb{G}_{\mathrm{P},2})$. Finally, from theorem 4.2, we get that $(\mathtt{M}_2\{\gamma\} \mid \mathtt{I}') \Downarrow_{\mathrm{op}} c_{\mathtt{f}}[\langle\rangle]$, as we wanted. □

4.3 Counterexample to full-abstraction

The interaction of the term $\mathtt{u} := \mathsf{let}\ y = \mathbf{new}\,\mathbb{E}\ \mathsf{in}\ \{f\,(\lambda x.y\#\mathbf{op}\,\langle\rangle)\}\ \mathbf{with}\ \mathtt{h}^{y}$ (from example 3.2) with the environment $C := \mathsf{let}\ f = (\lambda g.g\,\mathtt{n};\mathbf{ret}\,\mathtt{tt})\ \mathsf{in}\ []$ was represented by the following trace

$$\mathtt{s_u} = \boldsymbol{f}.\langle \square\, g \mid c_0\rangle^{\oplus}\boldsymbol{g}.\langle \square\,\mathtt{n} \mid d_1\rangle^{\ominus}\boldsymbol{d_1}.\langle \kappa[e] \mid \square\rangle^{\oplus}\boldsymbol{c_0}.\langle \varkappa :: \kappa[e] \mid \square\rangle^{\ominus}$$
$$\boldsymbol{\kappa_c}.\langle \square[\mathbf{ret}\,5] \mid c_1\rangle^{\oplus}\boldsymbol{c_1}.\langle \mathbf{ret}\,\mathtt{tt} \mid \square\rangle^{\ominus}\boldsymbol{d_0}.\langle \mathbf{ret}\,\mathtt{tt} \mid \square\rangle^{\oplus}$$

Now considering the term $\mathtt{t} := f\,(\lambda x.5)$, that is contextually equivalent to u, and observing how it interacts with the same environment C, we get:

$$\begin{array}{lll} \langle c_{\mathtt{f}}[\mathtt{t}], \varepsilon, \varnothing\rangle & \xrightarrow{\boldsymbol{f}.\langle \square\, g \mid d\rangle^{\oplus}}_{\mathrm{I}} & \langle [g \mapsto (\lambda x.5);\ d \mapsto c_{\mathtt{f}}[]];\ \varnothing\rangle \\ & \xrightarrow{\boldsymbol{g}.\langle \square\,\mathtt{V} \mid c\rangle^{\ominus}}_{\mathrm{I}} & \langle c[(\lambda x.5)]\ \mathtt{V};\ [g \mapsto (\lambda x.5);\ d \mapsto c_{\mathtt{f}}[]];\ \varnothing\rangle \\ & \xrightarrow{\boldsymbol{c}.\langle \mathbf{ret}\,5 \mid \square\rangle^{\oplus}}_{\mathrm{I}} & \langle [g \mapsto (\lambda x.5);\ d \mapsto c_{\mathtt{f}}[]];\ \varnothing\rangle \\ & \xrightarrow{\boldsymbol{d}.\langle \mathbf{ret}\,\mathtt{t} \mid \square\rangle^{\ominus}}_{\mathrm{I}} & \langle c_{\mathtt{f}}[\mathbf{ret}\,\mathtt{tt}], [g \mapsto (\lambda x.5);\ d \mapsto c_{\mathtt{f}}[]];\ \varnothing\rangle \\ & \xrightarrow{\boldsymbol{c_{\mathtt{f}}}.\langle \mathbf{ret}\,\mathtt{t} \mid \square\rangle^{\oplus}}_{\mathrm{I}} & \langle [g \mapsto (\lambda x.5);\ d \mapsto c_{\mathtt{f}}[]];\ \varnothing\rangle \end{array}$$

that is an interaction witnessed by the trace $\mathtt{s_t}$

$$\mathtt{s_t} = \boldsymbol{f}.\langle \square\, g \mid d\rangle^{\oplus}\boldsymbol{g}.\langle \square\,\mathtt{V} \mid c\rangle^{\ominus}\boldsymbol{c}.\langle \mathbf{ret}\,5 \mid \square\rangle^{\oplus}\boldsymbol{d}.\langle \mathbf{ret}\,\mathtt{tt} \mid \square\rangle^{\ominus}\boldsymbol{c_{\mathtt{f}}}.\langle \mathbf{ret}\,\mathtt{tt} \mid \square\rangle^{\oplus}$$

It is clear that $\mathtt{s_u}$ is not a trace generated by t, and $\mathtt{s_t}$ is not a trace generated by u, so that their respective set of complete traces are incomparable. Therefore, a complete OGS model requires a notion of equivalence coarser than equality of traces, that would identify $\mathtt{s_t}$ and $\mathtt{s_u}$.

5 Conclusion and related work

We have provided a sound interactive model for a language with dynamically-scoped effect handlers and fresh generation of effect instances. Our model does not allow so far exchange of effect instances between the program and its environment, that would necessitate to keep track of such disclosed instances in a specific set, to enforce a non-omniscience property as in [14]. The well-bracketing LTS would also have to keep track of the exchanged delimited continuations when the program performs a public effect (using a disclosed instance that is) because the environment, in this case, would be able to handle it, and thus its behaviour must be bound accordingly.

In future work, we will present a notion of equivalence between set of complete traces, coarser than equality, that we conjecture to be fully-abstract.

Game semantics has a long history of providing a fully-abstract interactive models for languages with control operators, starting from the relaxation of the well-bracketed condition by Laird to capture control operators like call/cc [20] and exceptions [21]. Game models for both statically bound, first-class continuations and locally declared, dynamically bound prompts were presented in [23], via a monadic presentation of such effects. In this work, prompts and exceptions cannot be referred by their names and passed around.

To represent them as values, nominal game semantics has been developed as a versatile framework to handle generative effects like dynamic name creation [1], higher-order references [29] and generative exceptions and handlers [30].

Operational techniques like applicative, normal-form and environmental bisimulations has also been developed for higher-order language with fine-grained control operators like static delimited continuations via shift and reset operations [6], dynamic ones via prompt and reset [4], and algebraic effects and handlers were considered in [7]. This last work is the most relevant to us, however they do not consider fresh generation of instances. Using a notion of Kripke normal-form bisimulations as introduced in [16, 19], that is directly derived from an OGS model, it would be interesting to explore the development of normal form bisimulations for such fresh generation of effect instances.

References

[1] Samson Abramsky, Dan R. Ghica, Andrzej S. Murawski, C.-H. Luke Ong, and Ian David Bede Stark. 2004. Nominal Games and Full Abstraction for the Nu-Calculus. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*. IEEE Computer Society, 150–159. doi:10.1109/LICS.2004.1319609

[2] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full Abstraction for PCF. *Inf. Comput.* 163, 2 (dec 2000), 409–470. doi:10.1006/inco.2000.2930

[3] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2014. Distilling Abstract Machines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 363–376. doi:10.1145/2628136.2628154

[4] Andrés Aristizábal, Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. 2017. Environmental Bisimulations for Delimited-Control Operators with Dynamic Prompt Generation. *Log. Methods Comput. Sci.* 13, 3 (2017). doi:10.23638/LMCS-13(3:27)2017

[5] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123. doi:10.1016/j.jlamp.2014.02.001 Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.

[6] Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. 2019. Bisimulations for Delimited-Control Operators. *Log. Methods Comput. Sci.* 15, 2 (2019). doi:10.23638/LMCS-15(2:18)2019

[7] Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. 2020. A complete normal-form bisimilarity for algebraic effects and handlers. In *Formal Structures for Computation and Deduction*.

[8] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29.

[9] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30. doi:10.1145/3428194

[10] Vincent Danos, Hugo Herbelin, and Laurent Regnier. 1996. Game semantics and abstract machines. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 394–405.

[11] Paulo Emílio de Vilhena and François Pottier. 2023. A type system for effect handlers and dynamic labels. In *Programming Languages and Systems: 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings*. Springer Nature Switzerland Cham, 225–252.

[12] Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science* 103, 2 (Sept. 1992), 235–271. doi:10.1016/0304-3975(92)90014-7

[13] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. *SIGPLAN Not.* 28, 6 (June 1993), 237–247. doi:10.1145/173262.155113

[14] Dan R Ghica and Nikos Tzevelekos. 2012. A system-level game semantics. *Electronic Notes in Theoretical Computer Science* 286 (2012), 191–211.

[15] Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect Handlers via Generalised Continuations. 30 (2020), e5. doi:10.1017/S0956796820000040

[16] Daniel Hirschkoff, Guilhem Jaber, and Enguerrand Prebet. 2023. Deciding Contextual Equivalence of ν-Calculus with Effectful Contexts. In *Foundations of Software Science and Computation Structures - 26th International Conference, FoSSaCS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13992)*, Orna Kupferman and Pawel Sobocinski (Eds.). Springer, 24–45. doi:10.1007/978-3-031-30829-1_2

[17] J.M.E. Hyland and C.-H.L. Ong. 2000. On Full Abstraction for PCF. *Inf. Comput.* 163, 2 (dec 2000), 285–408. doi:10.1006/inco.2000.2917

[18] Guilhem Jaber and Andrzej S Murawski. 2021. Complete trace models of state and control. *Programming Languages and Systems* 12648 (2021), 348.

[19] Guilhem Jaber and Andrzej S. Murawski. 2021. Compositional relational reasoning via operational game semantics. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–13. doi:10.1109/LICS52264.2021.9470524

[20] James Laird. 1997. Full abstraction for functional languages with control. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*. 58–67. doi:10.1109/LICS.1997.614931

[21] James Laird. 2001. A Fully Abstract Game Semantics of Local Exceptions. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 105–114. doi:10.1109/LICS.2001.932487

[22] James Laird. 2007. A Fully Abstract Trace Semantics for General References. In *Proceedings of the 34th International Conference on Automata, Languages and Programming* (Wrocław, Poland) *(ICALP'07)*. Springer-Verlag, Berlin, Heidelberg, 667–679.

[23] James Laird. 2017. Combining control effects and their models: Game semantics for a hierarchy of static, dynamic and delimited control effects. *Ann. Pure Appl. Log.* 168, 2 (2017), 470–500. doi:10.1016/J.APAL.2016.10.011

[24] P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (Jan. 1964), 308–320. doi:10.1093/comjnl/6.4.308

[25] Søren B. Lassen and Paul Blain Levy. 2008. Typed normal form bisimulation for parametric polymorphism. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*. IEEE, 341–352.

[26] Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, Vol. 2. Springer.

[27] Ian Mason and Carolyn Talcott. 1991. Equivalence in functional languages with effects. *Journal of Functional Programming* 1, 3 (1991), 287–327. doi:10.1017/S0956796800000125

[28] James Hiram Morris Jr. 1969. *Lambda-calculus models of programming languages*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[29] Andrzej S Murawski and Nikos Tzevelekos. 2011. Game semantics for good general references. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE, 75–84.

[30] Andrzej S Murawski and Nikos Tzevelekos. 2014. Game semantics for nominal exceptions. In *Foundations of Software Science and Computation Structures: 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 17*. Springer, 164–179.

[31] Gordon Plotkin and John Power. 2001. Semantics for algebraic operations. *Electronic Notes in Theoretical Computer Science* 45 (2001), 332–345.

[32] Gordon Plotkin and John Power. 2002. Computational effects and operations: An overview. (2002).

[33] Gordon D Plotkin and Matija Pretnar. 2013. Handling algebraic effects. *Logical methods in computer science* 9 (2013).

[34] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* 3, POPL (2019), 5:1–5:29. doi:10.1145/3290318

A Concrete Interaction

$\mathcal{L}_{\text{ci}}$ can be seen as an abstract machine that performs a variant of the linear head reduction, which is known to correspond to interaction in game semantics [10], (having a global environment γ, it is similar to the Milner Abstract Machine [3]), computing the interaction between a program configuration and an environment configuration that are *composable*.

The purpose of the concrete interaction is to *evaluate* a term inside an environment while exhibiting the *interaction* between the two that gets obscured by plain *syntax substitution*. To this effect, the program being evaluated and its environment will remain decoupled, but will each carry an *information* component that will be threaded through their interaction. This component keeps track of the history of the interaction and it consists of a map between the names and the *concrete* code of codata that each party has disclosed and rendered accessible to the other.

A.1 Concrete interaction LTS

The concrete interaction describes how a term and a complementary *concrete* environment interact by disclosing information and alternate between passive and active states. It emphasizes the interaction points between the two by highlighting the information they disclose to each other. At every stage of the interaction, the passive player is described by an accumulative component tracking its past contribution to the interaction, whereas the active player description contains in addition an interactive term representing the current *active* computation.

It is a polarized abstract machine whose configurations involve a *passive* and a *decoupled* complementary *passive* $\mathcal{L}_{\text{AI}}$ configurations (*cf.* 3.2).

We formalize this notion of complementarity by the following definition.

Definition A.1 (Complementarity). Let $\mathbb{P} = \langle \mathtt{I}_a; \gamma_a; \xi_a \rangle$, $\mathbb{A} = \langle (\mathtt{M} \mid \mathtt{I}_p); \gamma_p; \delta_p \rangle \in \mathcal{A}$ such that $\mathtt{M} = \langle \mathtt{t} \mid \mathtt{S} \circ \mathtt{T} \rangle$ and $c_{\mathtt{f}} \in C$. We say that $\mathbb{P}$ and $\mathbb{A}$ are $c_{\mathtt{f}}$-complementary and write $\mathbb{P} \dashv\vdash_c \mathbb{A}$ when:

- $\mathtt{I}_a \cap \mathtt{I}_p = \emptyset$ and $\mathrm{dom}(\gamma_a) \cup \mathrm{dom}(\gamma_p) = \emptyset$
- $\exists y \in \{a, p\}. \exists d \in \mathrm{dom}(\gamma_y).\ c_{\mathtt{f}} \in \mathrm{supp}(\gamma_y(d))$
- $\forall y \in \{a, p\}. \forall d \in \mathrm{dom}(\gamma_y).\ (\mathrm{supp}(\gamma_y(d)) \setminus \{c_{\mathtt{f}}\}) \subseteq \mathrm{dom}(\gamma_{y^\perp})$
- $\mathrm{supp}(\mathrm{codom}(\xi_a)) \subseteq \mathrm{dom}(\gamma_p)$ and $\mathrm{supp}(\mathrm{codom}(\delta_p)) \subseteq \mathrm{dom}(\gamma_a)$
- $\mathrm{supp}(\mathtt{t}), \mathrm{supp}(\mathtt{S}) \subseteq \mathrm{dom}(\gamma_a)$ and $\mathrm{supp}(\mathtt{T}) \subseteq \mathrm{dom}(\gamma_a) \cup \mathrm{dom}(\xi_a)$.
- There exists a type configuration $\mathbb{T} = \langle \Gamma_o; \Delta_o \mid \Gamma_p; \Delta_p \rangle \in \mathcal{T}$ and a type τ *s.t.* either:
  $\mathbb{P} \wr \langle \Gamma_o, c_{\mathtt{f}} : \neg\tau; \Delta_o \mid \Gamma_p; \Delta_p \rangle$ and $\mathbb{A} \wr \mathbb{T}^\perp$ or $\mathbb{P} \wr \mathbb{T}$ and $\mathbb{A} \wr \langle \Gamma_p, c_{\mathtt{f}} : \neg\tau; \Delta_p \mid \Gamma_o; \Delta_o \rangle$.

We will sometimes omit the index c on the relation $\dashv\vdash_c$ in contexts that do not require us to be explicit and just write $\mathbb{P} \dashv\vdash \mathbb{A}$.

A.1.1 *configurations and transitions.*

configurations $\quad \mathrm{Confs}_{\text{ci}} \ni \mathbb{C}, \mathbb{D} ::= \langle \mathbb{P} \dashv\vdash \mathbb{A} \rangle$

The CI transitions are given by $\mapsto_{\text{ci}} := \overset{\text{eval}}{\mapsto}_{\text{ci}} \cup \overset{\text{dual}}{\mapsto}_{\text{ci}} \cup \overset{\text{hdl}}{\mapsto}_{\text{ci}}$ and are defined in figure 15. Similar to $\mathcal{L}_{\text{AI}}$, $\overset{\text{eval}}{\mapsto}_{\text{ci}}$ corresponds to evaluating a term through $\mapsto_{\text{eval}}$ down to an normal form, while $\overset{\text{hdl}}{\mapsto}_{\text{ci}}$ marks the end of effect propagation whereby the *active* player gets to capture the fragments of the *forward stack* that belong to the environment.

On the other hand, $\overset{\text{dual}}{\mapsto}_{\text{ci}}$ acts on configurations in normal form by dualizing the perspective of the computation; i.e taking the perspective of the environment.

The dualizing process. It is defined as the composition of two processes **abstract** and **concretize**; The polarity of the *abstract normal form* $\boldsymbol{\alpha}.\mathrm{p}$ produced by **abstract** is switched in order to take the environment's perspective, then the concretization is carried out on $\boldsymbol{\alpha}.\mathrm{p}$ as explained above.

Abstracting a *copattern* into the shapes $\langle r[e] \mid \mathtt{S}[\Box] \rangle$ or $\langle r[e] \mid \Box \rangle$; that is an abstract effect e with its delimiting abstract evaluation stack r is a gradual process. The fragments κ making up r cannot be computed from the syntax alone, an interaction has to force the environment to perform the necessary "abstract" $\mapsto_{\text{fwd}}$ reductions on stuck terms such as $\langle \iota\#\mathbf{op}\ \mathtt{v} \mid \mathtt{S}[\kappa[]] \circ \mathtt{T} \rangle$ or $\langle e \mid c[] \circ \mathtt{T} \rangle$ in oder to progressively compute r. The exchanged delimited continuations are gradually put in δ, whose disclosure and concretization is delayed until the effect has been effectively handled (cf. handling rule in figure 15).

Evaluating

$$\frac{(\mathtt{M} \mid \mathtt{I}_p \cdot \mathtt{I}_a) \mapsto_{\text{op}} (\mathtt{N} \mid \mathtt{I}_p \cdot \mathtt{I}'_a)}{\langle \mathtt{I}_p; \gamma_p \dashv\vdash (\mathtt{M} \mid \mathtt{I}_a); \gamma_a \rangle \overset{\text{eval}}{\mapsto}_{\text{ci}} \langle \mathtt{I}_p; \gamma_p \dashv\vdash (\mathtt{N} \mid \mathtt{I}'_a); \gamma_a \rangle}$$

Handling

$$\langle \mathtt{I}_a; \gamma_p; ?(\_, \_, \delta_p) \dashv\vdash (\mathtt{M} \mid \mathtt{I}_p); \gamma_a; \delta_a \rangle \overset{\text{hdl}}{\mapsto}_{\text{ci}} \langle \mathtt{I}_a; \gamma_p \cdot \delta_p \dashv\vdash (\mathtt{M} \mid \mathtt{I}_p); \gamma_a \rangle$$

(a) Internal evaluation

Dualizing

$$\frac{\mathbf{abstract}(\mathtt{Nf}, \delta_p) \nearrow\!\!\nearrow (\boldsymbol{\alpha}.\mathrm{p}, \gamma_{\text{abs}}, \xi_{\text{abs}}) \qquad \mathbf{concretize}(\boldsymbol{\alpha}.\mathrm{p}, \gamma_p, \xi_p) = \mathtt{M}}{\langle \mathtt{I}_a; \gamma_p; \xi_p \dashv\vdash (\mathtt{Nf} \mid \mathtt{I}_p); \gamma_a; \delta_a \rangle \overset{\text{dual}}{\mapsto}_{\text{ci}} \langle \mathtt{I}_p; \gamma_a \cdot \gamma_{\text{abs}}; \xi_a[\xi_{\text{abs}}] \dashv\vdash (\mathtt{M} \mid \mathtt{I}_a); \gamma_p; \xi_p \rangle}$$

(b) Interaction interface

Figure 15: transitions of the Concrete Interaction LTS

A.2 Properties of the CI LTS

Lemma A.2 (Complementarity preservation). *Let* $\mathbb{P} \in \mathrm{Confs}^{\ominus}_{\text{ci}}$, $\mathbb{A} \in \mathrm{Confs}^{\oplus}_{\text{ci}}$ *such that* $\mathbb{P} \dashv\vdash \mathbb{A}$. *If* $\langle \mathbb{P} \dashv\vdash \mathbb{A} \rangle \mapsto_{\text{ci}} \langle \mathbb{P}' \dashv\vdash \mathbb{A}' \rangle$, *then* $\mathbb{P}' \dashv\vdash \mathbb{A}'$.

Proof. Let's write $\mathbb{A} = \langle (\mathtt{M} \mid \mathtt{I}_a); \gamma_p; \delta_p \rangle$, $\mathbb{P} = \langle \mathtt{I}_p; \gamma_a; \xi_a \rangle$ and $\mathtt{M} = \langle \mathtt{t} \mid \mathtt{S} \circ \mathtt{T} \rangle$.

Case $\langle \mathbb{A} \dashv\vdash \mathbb{P} \rangle \overset{\text{op}}{\mapsto}_{\text{ci}} \langle \mathbb{A}' \dashv\vdash \mathbb{P}' \rangle$.
Then $\xi_a = \varnothing$ and $\delta_p = \varnothing$ and there exists $\mathtt{N} = \langle \mathtt{u} \mid \mathtt{S}' \circ \mathtt{T}' \rangle$ and $\mathtt{I}'_a$ s.t. $(\mathtt{M} \mid \mathtt{I}_a) \mapsto_{\text{op}} (\mathtt{N} \mid \mathtt{I}'_a)$, $\mathbb{A}' = \langle (\mathtt{N} \mid \mathtt{I}'_a); \gamma_p; \varnothing \rangle$ and $\mathbb{P}' = \mathbb{P}$.

Subcase $\mathtt{I}'_a \cap \mathtt{I}_p = \emptyset$.
It is immediate that $\mathrm{supp}(\mathtt{T}') \subseteq \mathrm{supp}(\mathtt{M}) \subseteq \mathrm{dom}(\gamma_a) \cup \mathrm{dom}(\delta_p)$. We have $\mathrm{supp}(\mathtt{u}), \mathrm{supp}(\mathtt{S}') \subseteq \mathrm{supp}(\mathtt{t}) \cup \mathrm{supp}(\mathtt{S})$, then by hypothesis we get $\mathrm{supp}(\mathtt{u}), \mathrm{supp}(\mathtt{S}') \subseteq \mathrm{supp}(\mathrm{dom}(\gamma_a))$. W.l.o.g. suppose $\exists \mathbb{T} = \langle \Gamma_o; \emptyset \mid \Gamma_p; \emptyset \rangle$ and $\subseteq \mathrm{dom}(\gamma_a)$. $\mathbb{P} \wr \langle \Gamma_o; \emptyset \mid \Gamma_p \cdot [c \mapsto \neg\tau]; \emptyset \rangle$ and $\mathbb{A} \wr \mathbb{T}^\perp$, then we have

$\mathtt{I}_a;\Gamma_o \Vdash_c \mathtt{M}$ and from lemma 2.2, we have $\mathtt{I}'_a;\Gamma_o \Vdash_c \mathtt{N}$, thus $\mathbb{A}' \between \mathbb{T}^\perp$. The remaining complementarity conditions are trivially satisfied.

Case $\langle \mathbb{A} \dashv\!\vdash \mathbb{P}\rangle \overset{\text{hdl}}{\mapsto}_{\text{ci}} \langle \mathbb{A}' \dashv\!\vdash \mathbb{P}'\rangle$.

Then $\mathbb{A}' = \langle(\mathtt{M} \mid \mathtt{I}_a);\ \gamma_p;\ \varnothing\rangle$ and $\mathbb{P}' = \langle \mathtt{I}_p;\ \gamma_a \cdot \delta_a\ \ \varnothing\rangle$ where $\xi_a = (\_, \_, \delta_a)$.

Since $\text{supp}(\mathtt{M}) \subseteq \text{dom}(\gamma_a) \cup \text{dom}(\delta_a)$ then $\text{supp}(\mathtt{M}) \subseteq \text{dom}(\gamma_a \cdot \delta_a)$.

W.l.o.g. let $\mathbb{T} = \langle \Gamma_o; \Delta_o \mid \Gamma_p; \Delta_p\rangle$ and suppose $\mathbb{P} \between \langle \Gamma_o; \Delta_o \mid \Gamma_p \cdot [c \mapsto \neg\tau]; \Delta_p\rangle^\perp$ and $\mathbb{A} \between \mathbb{T}$., Taking $\mathbb{S} = \langle \Gamma_o \cdot \Delta_o; \emptyset \mid \Gamma_p; \emptyset\rangle$, since $\mathtt{I}_a; \Gamma_o \cdot \Delta_o \Vdash_c \mathtt{M}$, we can verify that $\mathbb{A}' \between \mathbb{S}$ and $\mathbb{P} \between \langle \Gamma_o \cdot \Delta_o; \emptyset \mid \Gamma_p \cdot [c \mapsto \neg\tau]; \emptyset\rangle^\perp$.

Case $\langle \mathbb{A} \dashv\!\vdash \mathbb{P}\rangle \overset{\text{dual}}{\mapsto}_{\text{ci}} \langle \mathbb{A}' \dashv\!\vdash \mathbb{P}'\rangle$.

By taking **abstract**$(\mathtt{M}, \delta_p) \nearrow\!\!\!\nearrow (\boldsymbol{\alpha}.\mathtt{p}, \gamma_{\text{abs}}, \xi_p)$ and writing **concretize**$(\boldsymbol{\alpha}.\mathtt{p}, \gamma_a; \xi_a) = (\mathtt{N}, \delta_a)$ with $\mathtt{N} = \langle\!\langle \mathtt{u} \mid \mathtt{S}' \circ \mathtt{T}' \rangle\!\rangle$, we have $\mathbb{A}' = \langle(\mathtt{N} \mid \mathtt{I}_p);\ \gamma_a;\ \delta_a\rangle$ and $\mathbb{P}' = \langle \mathtt{I}_a;\ \gamma_p \cdot \gamma_{\text{abs}};\ \delta_p \cdot \xi\rangle$.

We have that $\text{supp}(\text{codom}(\delta_a)) = \text{supp}(\text{codom}(\xi_a))$ and that $\text{supp}(\text{codom}(\xi)) \subseteq \text{supp}(\mathtt{t})$, thus by hypothesis, we have $\text{supp}(\text{codom}(\delta_a)) \subseteq \text{dom}(\gamma_p) \subseteq \text{dom}(\gamma_p \cdot \gamma_{\text{abs}})$, and we also have $\text{supp}(\text{codom}(\xi \cdot \delta_p)) \subseteq \text{supp}(\gamma_a)$.

W.l.o.g. let $\mathbb{T} = \langle \Gamma_o; \Delta_o \mid \Gamma_p; \Delta_p\rangle$ and suppose $\mathbb{P} \between \langle \Gamma_o; \Delta_o \mid \Gamma_p \cdot [c \mapsto \neg\tau]; \Delta_p\rangle^\perp$ and $\mathbb{A} \between \mathbb{T}$., Since $\mathtt{I}_a; \Gamma_o \cdot \Delta_o \Vdash_c \mathtt{M}$ and $\alpha \in \text{dom}(\Gamma_o)$ (because $\alpha \in \text{supp}(\mathtt{t}) \cup \text{supp}(\mathtt{S})$), then there exists τ s.t. $\Gamma_o(\alpha) = \tau$.

By writing $\Gamma_o \cdot \Delta_o \vdash_{\mathtt{p}} \mathtt{p} : \neg\tau \triangleright \Gamma_{\mathtt{p}}; \Delta_{\mathtt{p}}$ and taking $\mathbb{S} = \langle \Gamma_p \cdot \Gamma_{\mathtt{p}}; \Delta_p \cdot \Delta_{\mathtt{p}} \mid \Gamma_o; \Delta_o\rangle$, it is easy to verify that $\mathbb{A}' \between \mathbb{S}$ and $\mathbb{P}' \between \langle \Gamma_o; \Delta_o \mid \Gamma_p \cdot \Gamma_{\mathtt{p}} \cdot [c \mapsto \neg\tau]; \Delta_p \cdot \Delta_{\mathtt{p}}\rangle$.

□

In the following, we will use the notation $\langle \mathbb{P} \dashv\!\vdash \mathbb{A}\rangle$ to denote the couple $(\mathbb{P}, \mathbb{A}) \in \mathcal{A}^2$ satisfying $\mathbb{P} \dashv\!\vdash \mathbb{A}$.

A.2.1 Concrete observation. If an observation amounts to evaluating an observable value, an environment configuration can only do this after a switch of perspective, i.e through a $\overset{\text{dual}}{\mapsto}_{\text{ci}}$ transition. For this reason we will define the obsrevation transition $\Downarrow_{\text{ci}}$ a closure on $\overset{\text{dual}}{\mapsto}_{\text{ci}}$ (up-to $\overset{\text{eval}}{\mapsto}_{\text{ci}}$ and $\overset{\text{hdl}}{\mapsto}_{\text{ci}}$) such that $\mathbb{C} \Downarrow_{\text{ci}} \mathbb{D}$ means $\mathbb{C} \mapsto^*_{\text{ci}} \mathbb{D}$ where $\mathbb{C}$ and $\mathbb{D}$ are of opposite perspective.

Definition A.3. Taking $\Rightarrow_{\text{dual}}$ to be $(\overset{\text{hdl}}{\mapsto}_{\text{ci}} \cup \overset{\text{eval}}{\mapsto}_{\text{ci}})^* \overset{\text{dual}}{\mapsto}_{\text{ci}} (\overset{\text{hdl}}{\mapsto}_{\text{ci}} \cup \overset{\text{eval}}{\mapsto}_{\text{ci}})^*$ we define:

$$\mathbb{C} \Downarrow_{\text{ci}} \mathbb{D} \ :\Longleftrightarrow\ \exists k \in \mathbb{N}.\, \mathbb{C} \Rightarrow^{2*k+1}_{\text{dual}} \mathbb{D} \wedge \mathbb{D} \not\mapsto_{\text{ci}}$$

where, for $n \in \mathbb{N}$, $\Rightarrow^{\text{n}}_{\text{dual}} := \overbrace{\Rightarrow_{\text{dual}} \cdots \Rightarrow_{\text{dual}}}^{\text{n times}}$.

A.2.2 Relating $\mathcal{L}_{\text{AI}}$ and $\mathcal{L}_{\text{ci}}$. We observe that, by definition, the *interaction* between a program and its environment given in the form of *concrete interaction* hides the *parallel abstract composition* of the underlying complementary passive and active $\mathcal{L}_{\text{AI}}$ configurations.

The following proposition captures how the $\mapsto_{\text{ci}}$ transitions, can be equivalently re-expressed in terms of *complementarity* and and the transition $\rightarrow_{\text{I}}$.

Proposition A.4 (Parallel composition).

Evaluating

$$\frac{\mathbb{J} \dashv\!\vdash \mathbb{I} \qquad \mathbb{I} \xrightarrow{\text{eval}}_{\text{I}} \mathbb{I}'}{\langle \mathbb{J} \dashv\!\vdash \mathbb{I}\rangle \overset{\text{eval}}{\mapsto}_{\text{ci}} \langle \mathbb{J} \dashv\!\vdash \mathbb{I}'\rangle}$$

Handling

$$\frac{\mathbb{J} \dashv\!\vdash \mathbb{I} \qquad \mathbb{J} \xrightarrow{hdl^{\ominus}}_{\text{I}} \mathbb{K} \qquad \mathbb{I} \xrightarrow{hdl^{\oplus}}_{\text{I}} \mathbb{J}}{\langle \mathbb{J} \dashv\!\vdash \mathbb{I}\rangle \overset{\text{hdl}}{\mapsto}_{\text{ci}} \langle \mathbb{J} \dashv\!\vdash \mathbb{J}\rangle}$$

Dualizing

$$\frac{\mathbb{J} \dashv\!\vdash \mathbb{I} \qquad \mathbb{J} \xrightarrow{\mathbf{m}}_{\text{I}} \mathbb{J} \qquad \mathbb{I} \xrightarrow{\mathbf{m}^\perp}_{\text{I}} \mathbb{K}}{\langle \mathbb{J} \dashv\!\vdash \mathbb{I}\rangle \overset{\text{dual}}{\mapsto}_{\text{ci}} \langle \mathbb{K} \dashv\!\vdash \mathbb{J}\rangle}$$

Corollary A.5. *Given* $\mathbb{I} \in \text{Confs}_{\text{act}}$ *and* $\mathbb{J} \in \text{Confs}_{\text{pas}}$ *such that* $\mathbb{I} \dashv\!\vdash \mathbb{J}$*, we have:*

$$\langle \mathbb{J} \dashv\!\vdash \mathbb{I}\rangle \mapsto^*_{\text{ci}} \langle \mathbb{J}_0 \dashv\!\vdash \mathbb{J}_1\rangle \iff \exists \mathtt{t} \in \mathbf{Tr}_{\text{AI}},\ i \in \{0,1\}.\ \mathbb{J} \overset{\mathtt{t}}{\Rightarrow}_{\text{I}} \mathbb{J}_0 \wedge \mathbb{I} \overset{\mathtt{t}^\perp}{\Longrightarrow}_{\text{I}} \mathbb{J}_1$$

where $\overset{\mathbf{m}}{\Longrightarrow}_{\text{I}} := \xrightarrow{hdl}_{\text{I}} \xrightarrow{\text{eval}}{}^*_{\text{I}} \xrightarrow{\mathbf{m}}_{\text{I}} \xrightarrow{hdl}_{\text{I}} \xrightarrow{\text{eval}}{}^*_{\text{I}}$

B Adequacy

Now that all the necessary definitions are available, we will state the main intermediate lemmas for proving adequacy: lemmas B.1 and B.2, and we will prove them in the following sections C and D, respectively.

Lemma B.1 (Bisimilarity). *Taking an interactive term* $\mathtt{I}_1;\Gamma \Vdash_c \mathtt{M}$ *and a compatible name assignment* $\mathtt{I}_1';d:\neg\mathbb{1}\vdash\gamma:\Gamma$*, then:*

$$\langle \mathtt{I}_1',\ \gamma;\ \bot \dashv\vdash (\mathtt{M}\mid \mathtt{I}_1);\ \varepsilon,\ \bot\rangle \Downarrow_{\mathrm{ci}} \langle \mathtt{I}_2;\ \gamma_2;\ \bot \dashv\vdash (d[\mathbf{ret}\,\langle\rangle]\mid \mathtt{I}_2');\ \gamma_1;\ \bot\rangle$$
$$\Longleftrightarrow$$
$$(\mathtt{M}\{\gamma\}\mid \mathtt{I}_1'\cdot\mathtt{I}_1)\Downarrow_{\mathrm{op}} (d[\mathbf{ret}\,\langle\rangle]\mid \mathtt{I}_2'\cdot\mathtt{I}_2)$$

Lemma B.2 (Full observation). *Taking an interactive term* $\mathtt{I};\Gamma\Vdash_c \mathtt{M}$ *and a compatible name assignment* $\mathtt{I}';d:\neg\mathbb{1}\vdash\gamma:\Gamma$*, then by writing* $\mathbb{G}=\langle\mathbb{I}\parallel\mathbb{T}\parallel\mathbb{W}\rangle$ *and* $\mathbb{H}=\langle\mathbb{J}\parallel\mathbb{S}\parallel\mathbb{U}\rangle$ *for their respective* ogs *embeddings:*

$$\exists\mathbb{K},\gamma'.\ \langle\mathbb{J}\dashv\vdash\mathbb{I}\rangle\Downarrow_{\mathrm{ci}}\langle\mathbb{K}\dashv\vdash d[\mathbf{ret}\,\langle\rangle];\ \gamma';\ \bot\rangle$$
$$\Longleftrightarrow$$
$$\exists \mathtt{t}\in\mathbf{CTr}_{\mathrm{ogs}}(\mathbb{G}).\ \mathtt{t}^{\bot}\,\boldsymbol{d}.\langle\mathbf{ret}\,\langle\rangle\mid\Box\rangle^{\oplus}\in\mathbf{CTr}_{\mathrm{ogs}}(\mathbb{H})$$

Theorem B.3 (Adequacy). *Given an interactive term* $\mathtt{I};\Gamma\Vdash_c\mathtt{M}$ *and a compatible name assignment* $\mathtt{I}';d:\neg\mathbb{1}\vdash\gamma:\Gamma$*, then by writing* $\mathbb{P}$ *and* $\mathbb{E}$ *for their respective* ogs *embeddings, we have:*

$$((\mathtt{M}\{\gamma\}\mid \mathtt{I}'\cdot\mathtt{I})\Downarrow_{\mathrm{op}}(d[\,\mathbf{ret}\,\langle\rangle]\mid \mathtt{I}'')$$
$$\Longleftrightarrow$$
$$\exists \mathtt{t}\in\mathbf{CTr}_{\mathrm{ogs}}(\mathbb{P}).\ \mathtt{t}^{\bot}\,\boldsymbol{d}.\langle\mathbf{ret}\,\langle\rangle\mid\Box\rangle^{\oplus}\in\mathbf{CTr}_{\mathrm{ogs}}(\mathbb{E})$$

Proof. Follows from the two preceding lemmas. □

C Bisimilarity (proof of lemma B.1) $(\mathcal{L}_{\mathrm{ci}},\mapsto_{\mathrm{ci}})$ and $(\Lambda_{\mathrm{eff}},\mapsto_{\mathrm{op}})$ are weakly bisimilar

We start by stating the main result of this section; that is modulo $\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}}$ and $\overset{\mathrm{hdl}}{\mapsto}_{\mathrm{ci}}$ reductions, the transition systems $(\mathcal{L}_{\mathrm{ci}},\mapsto_{\mathrm{ci}})$ and $(\Lambda_{\mathrm{eff}},\mapsto_{\mathrm{op}})$ are realted by a weak bisimulation. Afterwards we will introduce the necessary definitions and lemmas in order to prove it.

The proof relies on a notion of *telescoped substitution* to avoid cycles in the concatenation of mappings coming from the composition of two interactive configurations. This enforces the absence of livelocks in the interaction.

Definition C.1. A *telescoped substitution* δ is a substitution seen as a stack of mappings $[\alpha_0\mapsto \mathtt{v}_0,\dots,\alpha_k\mapsto \mathtt{v}_k]$ such that for all $i\in\{1,\dots,k\}$, we have $\mathrm{supp}(\mathtt{v}_i)\cap\{\alpha_i,\dots,\alpha_k\}=\emptyset$.

Proposition C.2. *Given a* telescoped substitution $[\alpha_0\mapsto \mathtt{v}_0,\dots,\alpha_k\mapsto \mathtt{v}_k]$*, for all* $i\in\{1,\dots,k\}$*, we have* $\mathrm{supp}(\mathtt{v}_i)\subseteq\{\alpha_0,\dots,\alpha_i\}$.

A telescoped substitution δ can be transformed into a substitution δ^* such that for any $\alpha\in\mathrm{dom}(\delta)$ $\mathrm{supp}(\delta(\alpha))\subseteq\mathrm{supp}(\mathtt{v}_0)$.

Definition C.3. Taking a telescoped substitution δ and $k\in\mathbb{N}^*$, we define the iterated telescoped substitution δ^i as:

$$\delta^1:=\delta\qquad\qquad \delta^{i+1}:=\Pi_{\mathtt{n}\in\mathrm{dom}(\delta)}[\mathtt{n}\mapsto\delta(\mathtt{n})\{\delta^i\}]$$

We then define δ^* as δ^k with k the size of the domain of δ. Then if $\Gamma\cdot\Delta\vdash\delta:\Gamma$ and $\mathrm{dom}(\Delta)=\mathrm{supp}(\mathtt{v}_0)$, we have $\Delta\vdash\delta^*:\Gamma$.

Lemma C.4. *Taking a telescoped substitution* δ *and* $\alpha\in\mathrm{dom}(\delta)$*, then there exists a name* $\alpha\in\mathrm{dom}(\delta)$ *and a natural number* $k\in\mathbb{N}^*$ *such that* $\mathrm{supp}(\delta^k(\alpha))\cap\mathrm{dom}(\delta)=\emptyset$.

We define a bisimulation between $\mapsto_{\mathrm{ci}}$ and $\mapsto_{\mathrm{op}}$ by collapsing $\mathcal{L}_{\mathrm{ci}}$ configurations into operational ones.

Lemma C.5. *Let* $\langle\mathbb{E}\dashv\vdash\mathbb{P}\rangle\in\mathrm{Confs}_{\mathrm{ci}}$ *such that* $\mathbb{E}\dashv\vdash_d\mathbb{P}$*. If we write* $\mathbb{E}=\langle\mathtt{I}_p;\ \gamma_p;\ \xi_p\rangle$ *and* $\mathbb{P}=\langle(\mathtt{M}\mid\mathtt{I}_a);\ \gamma_a;\ \delta_a\rangle$ *and* $\mathbb{P}\wr\langle\Gamma_o;\Delta_o\mid\Gamma_p;\Delta_p\rangle$*, then* $\delta=\gamma_a\cdot\gamma_p$ *is a telescoped substituition, and there exists* $\mathtt{I}_a\cdot\mathtt{I}_p;\ d:\neg\mathbb{1}\vdash\delta^*:\Gamma_p\cdot\Gamma_o$.

Definition C.6. We define the function $\Phi^{\mathrm{ci}}_{\mathrm{op}}:\mathrm{Confs}_{\mathrm{ci}}\to\Lambda_{\mathrm{eff}}$ as the function:

$$\langle\mathtt{I}_p;\ \gamma_p;\ \xi_p\dashv\vdash(\mathtt{M}\mid\mathtt{I}_a);\ \gamma_a;\ \delta_a\rangle\mapsto(\mathtt{M}\{(\gamma_p\cdot\gamma_a)^*\cdot\delta_e\}\mid\mathtt{I}_p\cdot\mathtt{I}_a)$$

with

$$\delta_e:=\begin{cases}\varepsilon & \text{if } \xi_p=\bot\\ \delta_a\cdot\delta_p & \text{if for some } \mathtt{Y}\in\{a,p\},\ \xi_p=!(e,\_,\delta_p)\end{cases}$$

We write $\mathscr{B}^{\mathrm{ci}}_{\mathrm{op}}$ for the functional relation corresponding to $\Phi^{\mathrm{ci}}_{\mathrm{op}}$.

Lemma C.7. *If* $\mathbb{C}\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}}\mathbb{D}$ *or* $\mathbb{C}\overset{\mathrm{hdl}}{\mapsto}_{\mathrm{ci}}\mathbb{D}$*, then* $\Phi^{\mathrm{ci}}_{\mathrm{op}}(\mathbb{C})=\Phi^{\mathrm{ci}}_{\mathrm{op}}(\mathbb{D})$.

Proof. We proceed by case analysis knowing that $\mathbb{C}=\langle\mathtt{I}_p;\ \gamma_p;\ \xi_p\dashv\vdash(\mathtt{P}(\alpha)\mid\mathtt{I}_a);\ \gamma_a;\ \delta_a\rangle$ and that $\mathtt{P}(\alpha)$ is in normal form. □

Lemma C.8. *Taking a configuration* $\mathbb{C}\in\mathrm{Confs}_{\mathrm{ci}}$*, there exists a configuration* $\mathbb{D}$ *such that* $\mathbb{C}(\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}})^*\mathbb{D}$ *and* $\mathbb{D}\not\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}}$.

Proof. Let's write $\mathbb{C}$ as $\langle\mathtt{I}_p;\ \gamma_p;\ \xi_p\dashv\vdash(\mathtt{M}\mid\mathtt{I}_a);\ \gamma_a;\ \delta_a\rangle$ and $\gamma=[\alpha_0\mapsto\mathtt{v}_0,\dots,\alpha_{\mathtt{n}}\mapsto\mathtt{v}_{\mathtt{n}}]$ for $\gamma_a\cdot\gamma_p$.

If $\mathtt{M}$ is not in normal form then $\mathbb{C}\not\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}}$, thus $\mathbb{D}=\mathbb{C}$. Otherwise, there exists an index $k\in\{0,\dots,\mathtt{n}\}$ and a copattern, whose shape is either $\langle\Box\_\mid\mathtt{S}\rangle$ or $\langle\_\mid\mathtt{S}[\Box]\circ\_\rangle$, such that $\mathtt{M}=\mathtt{P}(\alpha_k)$.
We proceed by induction on k and on the length of $\mathtt{S}$.

case $\mathtt{M}=\langle f\,\mathtt{v}\mid\mathtt{S}\rangle$ with $f\in\mathrm{dom}(\gamma)$.
  Subcase $\gamma(f)\in\mathrm{dom}(\gamma)$.
    Since γ is a telescoped substitution, there exists $\ell\in\{k+1,\dots,\mathtt{n}\}$ such that $\gamma(f)=\alpha_\ell$ Thus there exists a configuration $\mathbb{D}$ such that $\mathbb{C}\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}}\mathbb{C}'$ and whose active component can be written as $\langle d[\alpha_\ell\,\mathtt{V}];\ \gamma_p\cdot\gamma_{\mathtt{V}};\ \varnothing\rangle$ By induction hypothesis, there exists $\mathbb{D}$ such that $\mathbb{C}'(\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}})^*\mathbb{D}$, and in particular $\mathbb{C}'(\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}})^*\mathbb{D}$, and $\mathbb{D}\not\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}}$.
  Subcase $\gamma(f)\notin\mathrm{dom}(\gamma)$.
    $\gamma(f)$ is a λ-abstraction. Then there exists a configuration $\mathbb{D}$ such that $\mathbb{C}\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}}\mathbb{D}$ and whose active component can be written as $\langle d[(\lambda x.\mathtt{t})\,\mathtt{V}];\ \gamma_p\cdot\gamma_{\mathtt{V}};\ \varnothing\rangle$. From which we can deduce that $\mathbb{D}\overset{\mathrm{op}}{\mapsto}_{\mathrm{ci}}$ and in particular that $\mathbb{D}\not\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}}$.
case $\mathtt{M}::=\langle\mathbf{ret}\,\mathtt{v}\mid\mathtt{S}[\kappa[]]\rangle\mid\langle\mathtt{e}\mid\mathtt{S}[\kappa[]]\circ\mathtt{S}\rangle$ with $\kappa\in\mathrm{dom}(\gamma)$.
  Subcase $\gamma(\kappa)=\mathtt{T}[\mathtt{E}]$.
    Then there exists a configuration $\mathbb{D}$ such that $\mathbb{C}\overset{\mathrm{dual}}{\mapsto}_{\mathrm{ci}}\mathbb{D}$ and whose active term $\mathtt{N}$ can be written as $\langle\mathbf{ret}\,\mathtt{V}\mid\mathtt{S}[\mathtt{T}[\mathtt{E}]]\rangle$ in which case $\mathtt{N}\mapsto_{\mathrm{eval}}$, or as $\langle\mathtt{e}\{\delta_p\}\mid\mathtt{S}[\mathtt{T}[\mathtt{E}]]\circ\mathtt{S}\rangle$ in which

case $\mathtt{N} \mapsto_{\mathsf{fwd}}$. In either case, we have $\mathbb{D} \overset{\mathsf{op}}{\mapsto}_{\mathsf{ci}}$ and in particular that $\mathbb{D} \overset{\mathsf{dual}}{\not\mapsto}_{\mathsf{ci}}$.

Subcase $\gamma(\kappa) = \Box$.

Suppose $\mathtt{M} = \langle \mathbf{ret}\, \mathtt{v} \mid \mathtt{S}[\kappa[]] \rangle$, we have

$$\begin{array}{lll} \mathbb{C} & \overset{\mathsf{dual}}{\mapsto}_{\mathsf{ci}} & \langle \mathtt{I}_a;\, \gamma_a;\, \varnothing \dashv \langle \mathbf{ret}\, \mathtt{V} \mid d[] \mid \mathtt{I}_p \rangle;\, \gamma_p \cdot \gamma_{\mathtt{V}} \cdot [d \mapsto \mathtt{S}];\, \varnothing \rangle \\ & \overset{\mathsf{dual}}{\mapsto}_{\mathsf{ci}} & \langle \mathtt{I}_p;\, \gamma_p \cdot \gamma_{\mathtt{V}} \cdot \gamma_d;\, \varnothing \dashv \langle \mathbf{ret}\, \mathtt{W} \mid \mathtt{S} \mid \mathtt{I}_a \rangle;\, \gamma_a \cdot \gamma_{\mathtt{W}};\, \varnothing \rangle = \mathbb{C}' \end{array}$$

We conclude by applying the induction hypothesis on $\mathbb{C}'$. The same reasoning applied to the other case.

case $\mathtt{M} ::= \langle \mathbf{ret}\, \mathtt{v} \mid c[]\, \rangle \mid \langle \mathtt{e} \mid c[]\, \circ \mathtt{S} \rangle$.

Subcase $c \notin \mathrm{dom}(\gamma)$ Then $\mathbb{D} = \mathbb{C}$ since $\mathbb{C} \overset{\mathsf{dual}}{\not\mapsto}_{\mathsf{ci}}$.

Subcase $\gamma(c) ::= \gamma(c) = \alpha_\ell[] \mid \mathtt{T}[\alpha_\ell[]]$ with $\ell \in \{k+1, \ldots, \mathtt{n}\}$.

There exists a configuration $\mathbb{C}'$ such that $\mathbb{C} \overset{\mathsf{dual}}{\mapsto}_{\mathsf{ci}} \mathbb{C}'$ and whose active component can be written as $\langle (\mathtt{Q}(\alpha_\ell) \mid \mathtt{I}_p);\, \gamma_p \cdot \gamma_{\mathtt{Q}};\, \varnothing \rangle$ where $\mathtt{Q}$ is a copattern. We conclude using the induction hypostesis on $\mathbb{C}'$.

□

Lemma C.9. *Let* $\mathtt{I}; \Gamma \Vdash_{\mathsf{c}} \mathtt{M}$ *be an interactive computation and let* $\mathtt{I}; \Delta \vdash \gamma : \Gamma$ *be a name assignment.*
If $(\mathtt{M} \mid \mathtt{I}) \mapsto_{\mathsf{op}} (\mathtt{N} \mid \mathtt{I}')$ *then* $(\mathtt{M}\{\gamma\} \mid \mathtt{I}) \mapsto_{\mathsf{op}} (\mathtt{N}\{\gamma\} \mid \mathtt{I}')$.

Lemma C.10. *Let* $\mathtt{I}; \Gamma \Vdash_{\mathsf{c}} \mathtt{M}$ *be a computation s.t.* $(\mathtt{M} \mid \mathtt{I}) \mapsto_{\mathsf{op}}$ *and let* $\mathtt{I}; \Delta \vdash \gamma : \Gamma$ *be a name assignment.*
If $(\mathtt{M}\{\gamma\} \mid \mathtt{I}) \mapsto_{\mathsf{op}} (\mathtt{N}\{\gamma\} \mid \mathtt{I}')$ *and* $\mathrm{supp}(\mathtt{M}) = \mathrm{supp}(\mathtt{N})$, *then* $(\mathtt{M} \mid \mathtt{I}) \mapsto_{\mathsf{op}} (\mathtt{N} \mid \mathtt{I}')$.

Proof. By case analysis knowing that any such term $\mathtt{M}$ is given by the syntax:

$$\begin{array}{lcl} \mathtt{Nf}_{\mathsf{fwd}} & ::= & \langle \mathbf{ret}\, \mathtt{v} \mid \mathtt{S}[\mathtt{E}] \rangle \mid \langle (\lambda x.\mathtt{t})\, \mathtt{v} \mid \mathtt{S} \rangle \\ & \mid & \langle \iota \# \mathbf{op}\, \mathtt{v} \mid \mathtt{S}[\{[]\}\, \mathbf{with}\, \mathtt{h}] \circ \mathtt{S} \rangle \quad (\iota \in \mathbf{hdl}(\mathtt{h})) \\ & \mid & \langle e \mid \mathtt{S}[\mathtt{E}] \circ \mathtt{S} \rangle \end{array}$$

□

Lemma C.11. *Taking a configuration* $\mathbb{C} \in \mathrm{Confs}_{\mathsf{ci}}$ *s.t.* $\Phi^{\mathsf{ci}}_{\mathsf{op}}(\mathbb{C}) \mapsto_{\mathsf{op}} (\mathtt{N} \mid \mathtt{I})$, *then there exists a configuration* $\mathbb{D} \in \mathrm{Confs}_{\mathsf{ci}}$ *s.t.* $\mathbb{C} \Longmapsto_{\mathsf{ci}} \mathbb{D}$ *and* $\Phi^{\mathsf{ci}}_{\mathsf{op}}(\mathbb{D}) = (\mathtt{N} \mid \mathtt{I})$.

Proof. Let's write $\mathbb{C} = \langle \mathtt{I}_p; \gamma_p; \xi_p \dashv (\mathtt{M} \mid \mathtt{I}_a); \gamma_a; \delta_a \rangle$.

Case $\mathbb{C} \overset{\mathsf{dual}}{\not\mapsto}_{\mathsf{ci}}$; that is $\mathtt{M} \notin \mathtt{Nf}(\Lambda_{\mathsf{eff}})$.

By hypothesis, we have $\Phi^{\mathsf{ci}}_{\mathsf{op}}(\mathbb{C}) = (\mathtt{M}\{\gamma\} \mid \mathtt{I}_p \cdot \mathtt{I}_a) \mapsto_{\mathsf{op}} (\mathtt{N}'\{\gamma\} \mid \mathtt{I})$ where $\gamma = (\gamma_p \cdot \gamma_a)^*$ and $\mathtt{N}'$ is such that $\mathrm{supp}(\mathtt{M}) = \mathrm{supp}(\mathtt{N}')$ and $\mathtt{M} = \mathtt{N}'\{\gamma\}$. Then from lemma C.10, we can deduce that $\mathbb{C} \overset{\mathsf{op}}{\mapsto}_{\mathsf{ci}} \langle \mathtt{I}_p; \gamma_p; \varnothing \dashv (\mathtt{N}' \mid \mathtt{I}'_a); \gamma_a; \varnothing \rangle = \mathbb{D}$ where $\mathtt{I} = \mathtt{I}_p \cdot \mathtt{I}'_a$, which entails that $\Phi^{\mathsf{ci}}_{\mathsf{op}}(\mathbb{D}) = (\mathtt{N}'\{\gamma\} \mid \mathtt{I}_p \cdot \mathtt{I}'_a) = (\mathtt{N} \mid \mathtt{I})$.

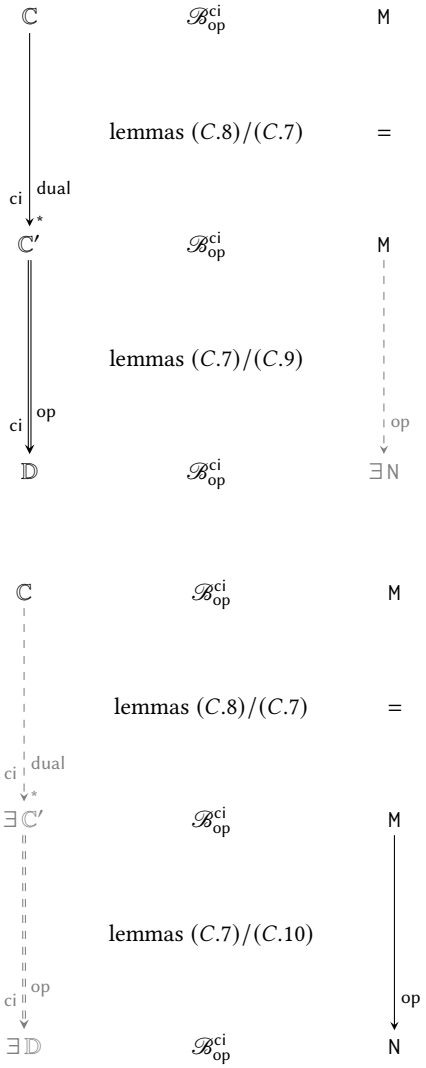
Case $\mathbb{C} \overset{\mathsf{dual}}{\mapsto}_{\mathsf{ci}}$.

By lemma C.8, there exists $\mathbb{C}'$ s.t. $\mathbb{C} (\overset{\mathsf{dual}}{\mapsto}_{\mathsf{ci}})^* \mathbb{C}'$ and $\mathbb{C}' \overset{\mathsf{dual}}{\not\mapsto}_{\mathsf{ci}}$. Since $\Phi^{\mathsf{ci}}_{\mathsf{op}}(\mathbb{C}) = \Phi^{\mathsf{ci}}_{\mathsf{op}}(\mathbb{C}')$ (by lemma C.7), then by applying the above argument to $\mathbb{C}'$ we can prove what we want.

□

Lemma C.12 (bisimulation). *Taking* $\overset{\mathsf{op}}{\Longmapsto}_{\mathsf{ci}} := (\overset{\mathsf{hdl}}{\mapsto}_{\mathsf{ci}})^* \overset{\mathsf{op}}{\mapsto}_{\mathsf{ci}}$ *and* $\Longmapsto_{\mathsf{ci}} := (\overset{\mathsf{dual}}{\mapsto}_{\mathsf{ci}})^* \overset{\mathsf{op}}{\Longmapsto}_{\mathsf{ci}}$, *there exists a bisimulation relating* $(\mathrm{Confs}_{\mathsf{ci}}, \Longmapsto_{\mathsf{ci}})$ *and* $(\Lambda_{\mathsf{eff}}, \mapsto_{\mathsf{op}})$, *i.e there exists a relation* $\mathscr{B}^{\mathsf{ci}}_{\mathsf{op}} \in \wp(\mathrm{Confs}_{\mathsf{ci}} \times \Lambda_{\mathsf{eff}})$ *s.t. the following diagram holds.*

Proof. We show that $\mathscr{B}^{\mathsf{ci}}_{\mathsf{op}}$ and its converse are simulations.



□

Corollary C.13. *Taking a nominal term* $\mathtt{I}; \Gamma; \emptyset \vdash_{\mathsf{c}} \mathtt{t} : \tau$, *a compatible evaluation context* $\mathtt{I}'; \Delta \vdash \mathtt{E} : \tau \rightsquigarrow \mathbb{1}$ *and a assignment*

$\mathtt{I}';\Delta \vdash \gamma : \Gamma$, *then:*

$$\langle \mathtt{I}'_1, \gamma \cdot [c \mapsto d[\mathtt{E}]]; \bot \dashv\vdash (c[\mathtt{t}] \mid \mathtt{I}_1); \varepsilon, \bot\rangle \Downarrow_{\mathsf{ci}} \langle \mathtt{I}_2; \gamma_2; \bot \dashv\vdash (d[\mathbf{ret}\,\langle\rangle] \mid \mathtt{I}'_2); \gamma_1; \bot\rangle \iff (\mathtt{E}[\mathtt{t}\{\gamma\}] \mid \mathtt{I}'_1 \cdot \mathtt{I}_1) \Downarrow_{\mathsf{op}} (d[\mathbf{ret}\,\langle\rangle] \mid \mathtt{I}'_2 \cdot \mathtt{I}_2)$$

D Observation (proof of lemma B.2)

D.1 Relating $(\mathcal{L}_{\mathcal{T}}, \rightarrow_{\mathcal{T}})$ to $(\mathcal{L}_{\mathsf{AI}} \rightarrow_{\mathsf{I}})$

Lemma D.1. *Taking a well-typed interactive term in normal form* $\mathtt{I};\Gamma \Vdash_{\mathsf{c}} \mathtt{Nf}$ *such that* $\mathbf{abstract}(\mathtt{Nf}, \delta) \nearrow\!\!\nearrow (\boldsymbol{\alpha}.\mathsf{q}, \gamma, \delta\cdot\xi)$ *and* $\Gamma(\alpha) = \tau$ *and* $\Gamma \vdash_{\mathsf{p}} \mathsf{q} : \neg\tau \triangleright \Gamma_{\mathsf{q}}; \Delta_{\mathsf{q}}$, *we have:* $\Gamma \vdash \gamma : \Gamma_{\mathsf{q}}$ *and* $\Gamma \vdash \xi : \Delta_{\mathsf{q}}$.

Lemma D.2. *Given a well-typed interactive term* $\mathtt{I};\Delta\cdot\Gamma \Vdash_{\mathsf{c}} \mathtt{M}$ *and a substitution* $\Gamma' \vdash \delta : \Delta$ *such that* $\Gamma' \subseteq \Gamma$ *and* $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\delta)$, *then:* $\mathtt{I};\Gamma \Vdash_{\mathsf{c}} \mathtt{M}\{\delta\}$

Lemma D.3 (Subject reduction). *Taking* $\mathbb{I}, \mathbb{J} \in \mathcal{A}$ *and* $\mathbb{T} \in \mathcal{T}$ *such that* $\mathbb{I} \between \mathbb{T}$ *and* $\mathbb{I} \xrightarrow{\alpha}_{\mathsf{I}} \mathbb{J}$, *then there exists* $\mathbb{T} \in \mathcal{T}$ *such that* $\mathbb{T} \xrightarrow{\alpha}_{\mathcal{T}} \mathbb{S}$ *and* $\mathbb{J} \between \mathbb{S}$.

Proof. We write $\mathbb{T} = \langle \Gamma_o; \Delta_o \mid \Gamma_p; \Delta_p\rangle$ and proceed by case analysis on α.

Case $\alpha = hdl^{\oplus}$.
Then if we take $\mathbb{I} = \langle(\mathtt{M} \mid \mathtt{I}); \gamma; \mathbf{!}\delta\rangle$, we have $\mathbb{I} \xrightarrow{hdl^{\oplus}}_{\mathsf{I}} \langle(\mathtt{M} \mid \mathtt{I}); \gamma; \varnothing\rangle = \mathbb{J}$. It is immediate to verify that $\mathbb{S} = \langle \Gamma_o\cdot\Delta_o; \emptyset \mid \Gamma_p; \emptyset\rangle$ is s.t. $\mathbb{T} \xrightarrow{hdl^{\oplus}}_{\mathcal{T}} \mathbb{S}$ and $\mathbb{J} \between \mathbb{S}$.

Case $\alpha = hdl^{\ominus}$.
Then if we take $\mathbb{I} = \langle \mathtt{I}; \gamma; \mathbf{?}\xi\rangle$ where $\xi = (\_, \_, \delta)$, we have $\mathbb{I} \xrightarrow{hdl^{\ominus}}_{\mathsf{I}} \langle \mathtt{I}; \gamma\cdot\delta; \varnothing\rangle = \mathbb{J}$. Similarily, it is immediate to verify that $\mathbb{S} = \langle \Gamma_o; \emptyset \mid \Gamma_p\cdot\Delta_p; \emptyset\rangle$ is s.t. $\mathbb{T} \xrightarrow{hdl^{\ominus}}_{\mathcal{T}} \mathbb{S}$ and $\mathbb{J} \between \mathbb{S}$.

Case $\alpha = \boldsymbol{\alpha}.\mathsf{p}^{\oplus}$.
Then if we write $\mathbb{I} = \langle(\mathtt{Nf} \mid \mathtt{I}); \gamma; \delta\rangle$, we have $\mathbb{I} \xrightarrow{\boldsymbol{\alpha}.\mathsf{p}^{\oplus}}_{\mathsf{I}} \langle \mathtt{I}; \gamma\cdot\gamma_{\mathsf{abs}}; \xi\rangle = \mathbb{J}$ whith $\mathbf{abstract}(\mathtt{Nf}, \delta) \nearrow\!\!\nearrow (\gamma_{\mathsf{abs}}, \delta\cdot\xi)$.
Since $\mathbb{I} \between \mathbb{T}$, we have $\mathtt{I};\Gamma_o\cdot\Delta_o \Vdash_{\mathsf{c}} \mathtt{Nf}$, and since $\alpha \in \mathrm{supp}(\mathtt{Nf})$ then there exists τ, Γ_{p}, Δ_{p} such that $\Gamma_o(\alpha) = \tau$ and $\Gamma_o\cdot\Delta_p\cdot\Delta_o \vdash_{\mathsf{p}} \mathsf{p} : \neg\tau \triangleright \Gamma_{\mathsf{p}}; \Delta_{\mathsf{p}}$, so by lemma D.1, we have $\Gamma_o \vdash \gamma_{\mathsf{abs}} : \Gamma_{\mathsf{p}}$ and $\Gamma_o\cdot\Delta_o \vdash \xi : \Delta_{\mathsf{p}}$.
Thus by taking $\mathbb{S} = \langle \Gamma_o; \Delta_o \mid \Gamma_p\cdot\Gamma_{\mathsf{p}}; \Delta_p\cdot\Delta_{\mathsf{p}}\rangle$, it is immediate that $\mathbb{T} \xrightarrow{\boldsymbol{\alpha}.\mathsf{p}^{\oplus}}_{\mathcal{T}} \mathbb{S}$ and $\mathbb{J} \between \mathbb{S}$, knowing that $\mathtt{I};\Gamma_o \vdash \gamma : \Gamma_p$, and $\mathtt{I};\Gamma_o \vdash \delta : \Delta_p$.

Case $\alpha = \boldsymbol{\alpha}.\mathsf{p}^{\ominus}$.
Then if we take $\mathbb{I} = \langle \mathtt{I}; \gamma; \xi\rangle$ where $\xi = (\_, \_, \delta)$, we have $\mathbb{I} \xrightarrow{\boldsymbol{\alpha}.\mathsf{p}^{\ominus}}_{\mathsf{I}} \langle(\mathtt{M} \mid \mathtt{I}); \gamma; \delta\rangle = \mathbb{J}$ where $\mathtt{M} = \mathsf{p}\{\delta\}(\gamma(\alpha))$.
Let τ, Γ_{p}, Δ_{p} such that $\Gamma_p(\alpha) = \tau$ and $\Delta_p\cdot\Delta_o \vdash_{\mathsf{p}} \mathsf{p} : \neg\tau\triangleright\Gamma_{\mathsf{p}}; \Delta_{\mathsf{p}}$, then we have $\mathbb{T} \xrightarrow{\boldsymbol{\alpha}.\mathsf{p}^{\oplus}}_{\mathcal{T}} \langle \Gamma_o\cdot\Gamma_{\mathsf{p}}; \Delta_o\cdot\Delta_{\mathsf{p}} \mid \Gamma_p; \Delta_p\rangle = \mathbb{S}$.
Since $\mathbb{I} \between \mathbb{T}$, we have $\mathtt{I};\Gamma_o \vdash \gamma : \Gamma_p$ and $\mathtt{I};\Gamma_o \vdash \delta : \Delta_p$ and in particular $\mathtt{I};\Gamma_o\cdot\Gamma_{\mathsf{p}} \vdash \gamma : \Gamma_p$ and $\mathtt{I};\Gamma_o\cdot\Gamma_{\mathsf{p}} \vdash \delta : \Delta_p$.
Moreover, since $\mathtt{I};[\alpha \mapsto \tau]\cdot\Gamma_{\mathsf{p}}\cdot\Delta_p\cdot\Delta_o\cdot\Delta_{\mathsf{p}} \Vdash_{\mathsf{c}} \mathsf{p}(\alpha)$ and $\mathrm{dom}(\Delta_o) = \mathrm{dom}(\delta)$, then by lemma D.2 we have $\mathtt{I};[\alpha \mapsto \tau]\cdot\Gamma_{\mathsf{p}}\cdot\Delta_o\cdot\Delta_{\mathsf{p}} \Vdash_{\mathsf{c}} \mathsf{p}\{\delta\}(\alpha)$. Finally, given that $\mathtt{M} = \mathsf{p}\{\delta\}(\alpha)$ and $\Gamma_o \vdash \gamma(\alpha) : \tau$ then we can deduce that $\mathtt{I};\Gamma_o\cdot\Gamma_{\mathsf{p}}\cdot\Delta_o\cdot\Delta_{\mathsf{p}} \Vdash_{\mathsf{c}} \mathtt{M}$, and by extent $\mathbb{J} \between \mathbb{S}$.

□

D.2 Compatibility of $(\mathcal{L}_{\mathsf{AI}}, \rightarrow_{\mathsf{I}})$ and $(\mathcal{L}_{wb}, \rightarrow_{wb})$

Definition D.4. We define the predicate $\wedge_{\square}$ on $\{\mathbf{?}, \mathbf{!}\}\times\mathcal{N}\times\mathcal{F}$, relating an *abstract forward stack* to a *call frame*, depending on which *player*

has performed the effect.

$$\frac{\cdot \wedge_? \ [] \qquad r \wedge_? \ f}{r :: \kappa' :: \kappa \wedge_? \kappa :: f} \qquad \frac{\cdot \wedge_! \ [] \qquad r \wedge_! \ f}{r :: \kappa' :: \kappa \wedge_! \kappa' :: f}$$

Definition D.5 ($(\mathcal{L}_{\mathrm{Al}}, \mathcal{L}_{wb})$-compatibility). Let $\mathbb{I}, \mathbb{J} \in \mathcal{A}$ and $\mathbb{W} \in \mathcal{W}$ s.t. $\mathbb{I} = \langle \gamma;\ \xi \rangle$, $\mathbb{J} = \langle \langle \mathtt{t} \mid \mathtt{S} \circ \mathtt{S} \mid \mathtt{I} \rangle; \gamma'; \delta \rangle$ and $\mathbb{W} = \langle \sigma \mid \eta \mid \phi \rangle$.

We say that $\mathbb{I}$ and $\mathbb{W}$ are compatible when $\mathbf{Compat}(\mathbb{I}, \mathbb{W})$ holds, where

$$\mathbf{Compat}(\mathbb{I}, \mathbb{W}) :\Longleftrightarrow \forall \mathtt{f} \in \phi.\ \mathrm{set}(\mathtt{f}) \subseteq \mathrm{dom}(\gamma) \ \wedge \ \xi \sim_{\mathrm{fwd}} \eta$$

Similarily, we say that $\mathbb{J}$ and $\mathbb{W}$ are compatible when $\mathbf{Compat}(\mathbb{J}, \mathbb{W})$ holds, where

$$\mathbf{Compat}(\mathbb{J}, \mathbb{W}) :\Longleftrightarrow \forall \mathtt{f} \in \phi.\ \mathrm{set}(\mathtt{f}) \subseteq \mathrm{dom}(\gamma') \ \wedge \ \delta \sim_{\mathrm{fwd}} \eta$$

with

$$\varnothing \sim_{\mathrm{fwd}} \varnothing \qquad \mathbf{!}(e, \_, \_) \sim_{\mathrm{fwd}} ([], e) \qquad \mathbf{?}(e, r, \_) \sim_{\mathrm{fwd}} (f, e) \text{ when } r \wedge_? f$$

$$\mathbf{!}\delta^e \sim_{\mathrm{fwd}} ([], e) \qquad \mathbf{?}\delta \sim_{\mathrm{fwd}} (\mathtt{f}, e) \text{ when } r \wedge_? \mathtt{f} \text{ and } \mathtt{S}\{\delta^{-1}\} = \mathtt{T} \oplus r$$

Lemma D.6 (preservation of compatibility). *Let $\mathbb{I} \in \mathcal{A}$ and $\mathbb{W} \in \mathcal{W}$ s.t.* $\mathbf{Compat}(\mathbb{I}, \mathbb{W})$.

If $\mathbb{I} \xrightarrow{\alpha}_{\mathrm{I}} \mathbb{J}$ and $\mathbb{W} \xrightarrow{\alpha}_{wb} \mathbb{U}$, then $\mathbf{Compat}(\mathbb{J}, \mathbb{U})$.

Proof. We write $\mathbb{W} = \langle \sigma \mid \eta \mid \phi \rangle$ with $\eta = (e, f)$ then proceed by case analysis on $\mathbb{I}$ and α.

Case $\mathbb{I} = \langle \mathtt{I};\ \gamma;\ \blacksquare \xi \rangle$ with $\blacksquare \in \{\mathbf{?}, \mathbf{!}\}$ and $\xi = (e, r, \delta)$.

Subcase $\alpha = hdl^{\ominus}$.

By hypothesis, we have $r \wedge_? f$ and $\forall f' \in \phi.\ \mathrm{set}(f') \subseteq \mathrm{dom}(\gamma)$. Moreover, we have $\mathbb{J} = \langle \mathtt{I};\ \gamma \cdot \delta;\ \varnothing \rangle$ and $\mathbb{U} = \langle \sigma \mid \varnothing \mid \phi \cup \{f\} \rangle$, thus, since $\mathrm{set}(f) \subseteq \mathrm{dom}(\delta)$ we can deduce that $\forall f' \in \phi \cup \{f\}.\ \mathrm{set}(f') \subseteq \mathrm{dom}(\gamma \cdot \delta)$ and $\varnothing \sim_{\mathrm{fwd}} \varnothing$, from which we can conclude that $\mathbf{Compat}(\mathbb{J}, \mathbb{U})$.

Subcase $\alpha ::= \boldsymbol{c}.\langle \mathbf{ret}\, \mathtt{V} \mid \square \rangle^{\ominus} \mid \boldsymbol{\kappa}.\langle \square[\mathbf{ret}\, \mathtt{V}] \mid d \rangle^{\ominus} \mid \boldsymbol{f}.\langle \square\, \mathtt{V} \mid d \rangle^{\ominus}$.

In either subcase, there exist $\mathtt{M}$ and σ' s.t. $\mathbb{J} = \langle (\mathtt{M} \mid \mathtt{I});\ \gamma;\ \varnothing \rangle$ and $\mathbb{U} = \langle \sigma' \mid \varnothing \mid \phi \rangle$. Moreover, by defintion $\varnothing \sim_{\mathrm{fwd}} \varnothing$, and by hypothesis we have $\forall f \in \phi.\ \mathrm{set}(f) \subseteq \mathrm{dom}(\gamma)$, thus $\mathbf{Compat}(\mathbb{J}, \mathbb{U})$.

Subcase $\alpha ::= \boldsymbol{c}.\langle \kappa[e] \mid \square \rangle^{\ominus} \mid \boldsymbol{\kappa'}.\langle \square[\kappa[e]] \mid d \rangle^{\ominus}$ (*performing an effect*).

In either subcase, there exist $\mathtt{S}, \sigma'$ s.t. $\mathbb{J} = \langle \langle e \mid \mathtt{S} \circ \kappa \mid \mathtt{I} \rangle;\ \gamma;\ \varepsilon \rangle$ and $\mathbb{U} = \langle \sigma' \mid (e, []) \mid \phi \rangle$.

We have then, by definition, $\kappa \wedge_? []$ and $\mathbf{?}\varepsilon \sim_{\mathrm{fwd}} (e, [])$, and by hypothesis, we have that $\forall f \in \phi.\ \mathrm{set}(f) \subseteq \mathrm{dom}(\gamma)$. Thus, we can conclude that $\mathbf{Compat}(\mathbb{J}, \mathbb{U})$.

Subcase $\alpha ::= \boldsymbol{c}.\langle \kappa :: r[e] \mid \square \rangle^{\ominus} \mid \boldsymbol{\kappa'}.\langle \square[\kappa :: r[e]] \mid d \rangle^{\ominus}$ (*forwarding an effect*).

In either subcase, there exist $\mathtt{S}, \sigma'$ such that $\mathbb{J} = \langle \langle e\{\delta\} \mid \mathtt{S} \circ \kappa[r\{\delta\}] \mid \mathtt{I} \rangle;\ \gamma;\ \blacksquare \delta \rangle$ and $\mathbb{U} = \langle \sigma' \mid \eta \mid \phi \rangle$.

By hypothesis, we have that $\forall f \in \phi.\ \mathrm{set}(f) \subseteq \mathrm{dom}(\gamma)$ and $r \wedge_{\blacksquare} f$.

If $\blacksquare = \mathbf{?}$, then r is even-length, and by taking $\mathtt{S} = \kappa[r\{\delta\}]$, we have $\mathtt{S}\{\delta^1\} = \kappa :: r$ and $\kappa :: r \wedge_? f$. If otherwise $\blacksquare = \mathbf{!}$, then $\eta = \varnothing$.

Thus in both cases, we have $\blacksquare\ \delta \sim_{\mathrm{fwd}} \eta$, and we can conclude that $\mathbf{Compat}(\mathbb{J}, \mathbb{U})$.

Case $\mathbb{I} = \langle \langle \mathtt{t} \mid \mathtt{S} \circ \mathtt{S} \mid \mathtt{I} \rangle;\ \gamma;\ \delta \rangle$.

Subcase $\alpha ::= \boldsymbol{d}.\langle \mathbf{ret}\, \mathtt{V} \mid \square \rangle^{\oplus} \mid \boldsymbol{\kappa}.\langle \square[\mathbf{ret}\, \mathtt{V}] \mid c \rangle^{\oplus} \mid \boldsymbol{f}.\langle \square\, \mathtt{V} \mid c \rangle^{\oplus}$.

In either subcase, there exist γ' and σ' s.t. $\mathbb{J} = \langle \mathtt{I};\ \gamma \cdot \gamma';\ \varnothing \rangle$ and $\mathbb{U} = \langle \sigma' \mid \varnothing \mid \phi \rangle$. By defintion, we have $\varnothing \sim_{\mathrm{fwd}} \varnothing$, and by hypothesis we have $\forall f \in \phi.\ \mathrm{set}(f) \subseteq \mathrm{dom}(\gamma)$, and in particular $\forall f \in \phi.\ \mathrm{set}(f) \subseteq \mathrm{dom}(\gamma \cdot \gamma')$, thus $\mathbf{Compat}(\mathbb{J}, \mathbb{U})$.

Subcase $\alpha ::= \boldsymbol{d}.\langle \kappa :: r[e] \mid \square \rangle^{\oplus} \mid \boldsymbol{\kappa'}.\langle \square[\kappa :: r[e]] \mid c \rangle^{\oplus}$.

In either subcase, there exist $\gamma', \sigma', \eta', \mathtt{S}'$ and $\mathtt{T}$ s.t. $\mathbb{J} = \langle \mathtt{I};\ \gamma \cdot \gamma';\ \blacksquare \xi \rangle$ and $\mathbb{U} = \langle \sigma' \mid \eta' \mid \phi \rangle$, where $\mathtt{S} = \mathtt{T} \oplus \mathtt{S}'$, $\mathtt{S}\{\delta^{-1}\} = r$ and $\xi = (e, r, \delta \cdot [\kappa \mapsto \mathtt{T}])$.

By hypothesis, we have that $\forall f \in \phi.\ \mathrm{set}(f) \subseteq \mathrm{dom}(\gamma)$ and $r \wedge_{\blacksquare} f$.

If $\blacksquare = \mathbf{?}$, then r is odd-length and $\eta' = (f +\!\!+ [\kappa], e)$, thus $\kappa :: r \wedge_? f +\!\!+ [\kappa]$. If otherwise $\blacksquare = \mathbf{!}$, then $\eta = \eta' = \varnothing$.

Thus in both cases, we have $\blacksquare\ \xi \sim_{\mathrm{fwd}} \eta$, and we can conclude that $\mathbf{Compat}(\mathbb{J}, \mathbb{U})$.

□

D.3 Semantic ciu-observation

Given a term $\mathtt{I}; \Gamma; \emptyset \vdash_{\mathrm{c}} \mathtt{t} : \tau$, an *observer* of $\mathtt{t}$ is an environment given by an evaluation context and a name valuation $(\mathtt{S}, \upsilon)$ such that $\mathtt{I}'; \emptyset \vdash \mathtt{S} : \tau \rightsquigarrow \mathbb{1}$ and $\mathtt{I}'; \emptyset \vdash \upsilon : \Gamma$.

We want to give a semantic characterization of "observing $\langle\rangle$" but first we need to introduce the notion of *closed composability* of OGS configurations, which is the semantic counterpart of $\mathtt{t}$ being composable with $(\mathtt{S}, \upsilon)$ and $\mathtt{S}[\mathtt{t}\{\upsilon\}]$ being a closed term of type $\mathbb{1}$.

D.3.1 Closed composability of OGS *configurations.* If *complementarity* guarantees the "closedness" part of *closed composability*, we still require non-intial *program* and *environment* configurations to agree on their history of interaction. We will capture this *coherence of trajectory* by the following *synchronicity relation.*

First, we will the notion of *return frame* and auxiliary definitions.

Definition D.7 (return frames). We will call a *return frame* any control-flow stack of the shape $f +\!\!+ [c]$;*i.e.* that it is made up of a call frame ($f \in \mathtt{List}(\mathcal{K})$) to which is appended a single abstract undelimited continuation ($c \in C$).

We will also define the function F that, given an evaluation stack, computes the associated *return frame.*

$$F(c[\mathtt{S}]) := F(\mathtt{S}) +\!\!+ [c] \qquad F(\mathtt{T}[\kappa[\mathtt{S}]]) := F(\mathtt{S}) +\!\!+ [\kappa] \qquad F(\mathtt{T}) := []$$

Definition D.8 (return frames decomposition). For any a non-empty control-flow stack σ generated by $\mathcal{L}_{wb}$, there corresponds a natural number $k \in \mathbb{N}^*$ and k *return frames* $\sigma_1, \cdots, \sigma_k$ such that $\sigma = \sigma_1 +\!\!+ \cdots +\!\!+ \sigma_k$.

We will write $\sigma_1 \oplus \cdots \oplus \sigma_k$ for this unique decomposition of σ.

Definition D.9 (Synchronicity). We define synchronicity as a quarternary relation $\mathcal{S} \subseteq \mathcal{A} \times \mathcal{W} \times \mathcal{A} \times \mathcal{W}$.

Let $\mathbb{I} = \langle (\mathtt{M} \mid \mathtt{I});\ \gamma_{\mathbb{I}};\ \blacksquare \delta_{\mathbb{I}} \rangle$ in $\mathrm{Confs}_{\mathrm{act}}$, $\mathbb{J} = \langle \gamma_{\mathbb{J}},\ \square(\_, r, \_) \rangle$ in $\mathrm{Confs}_{\mathrm{pas}}$ where $\mathtt{M} = \langle \mathtt{t} \mid c_0[\mathtt{S}_0] \circ \mathtt{S} \rangle$ and $\blacksquare, \square \in \{?, !\}$, Let $\mathbb{W} = \langle \sigma_{\mathbb{W}} \mid (f_{\mathbb{W}}, e_{\mathbb{W}}) \mid \phi_{\mathbb{W}} \rangle$, $\mathbb{U} = \langle \sigma_{\mathbb{U}} \mid (f_{\mathbb{U}}, e_{\mathbb{U}}) \mid \phi_{\mathbb{U}} \rangle$ in $\mathcal{W}$ where $\sigma_0 \oplus \cdots \oplus \sigma_k$ and $\sigma'_0 \oplus \cdots \oplus \sigma'_p$ (for some $k, p \in \mathbb{N}^*$) are the respective decompositions of $\sigma_{\mathbb{W}}$ and $\sigma_{\mathbb{U}}$ into *return frames.*

We say that $\langle \mathbb{I}, \mathbb{W} \rangle$ and $\langle \mathbb{J}, \mathbb{U} \rangle$ are *in sync*, and write $\langle \mathbb{I}, \mathbb{W} \rangle\ \mathcal{S}\ \langle \mathbb{U}, \mathbb{J} \rangle$ when the following holds:

- $\mathbb{I} \dashv\vdash \mathbb{J}$ and for all $i \geq 0$, we can write $\gamma_{\mathbb{I}}(c_i) = d_{i+1}[\mathsf{S}'_i]$ and $\gamma_{\mathbb{I}}(d_{i+1}) = c_{i+1}[\mathsf{S}_{i+1}]$.
- $r \wedge \blacksquare f_{\mathbb{U}}$ and $r \wedge \square f_{\mathbb{J}}$
- $\forall 0 \leq i.\ F(c_i[\mathsf{S}_i]) \simeq_{\phi_{\mathbb{U}}} \sigma'_i$ and $F(\gamma_{\mathbb{I}}(c_i)) \simeq_{\phi_{\mathbb{W}}} \sigma_i$

where $\simeq_\phi$ is given by

$$\textsc{Refl}\ \frac{\pi = \pi' \qquad \phi \in K}{\pi \simeq_\phi \pi'} \qquad \textsc{Up-to}\ \frac{\pi = \pi' \qquad \phi \in K \qquad f \in \phi}{\pi \simeq_\phi f \Rightarrow \pi'}$$

Next we show how the *synchronicity* relation indeed captures the *coherence* of both the *program's* and the *environment's* trajectories, in the sense that whatever interaction is triggered by the *active* player is in fact compatible with what the *passive* player is expecting, and that this property is preserved by interaction.

This is formalised in the next lemma, which proves that $\mathcal{S}$ is a simulation.

Lemma D.10 (O-simulation). *Given $\mathbb{K} \in \mathrm{Confs}_{\mathsf{pas}}$, $\mathbb{I} \in \mathrm{Confs}_{\mathsf{act}}$, $\mathbb{W}$, $\mathbb{U} \in \mathcal{W}$ such that $\langle \mathbb{I}, \mathbb{W}\rangle \mathcal{S} \langle \mathbb{K}, \mathbb{U}\rangle$, we have:*

$$\frac{\mathbb{I} \xrightarrow{\mathbf{m}}_{\mathsf{I}} \mathbb{J} \qquad \mathbb{W} \xrightarrow{\mathbf{m}}_{wb} \mathbb{W}'}{\exists \mathbb{J}, \mathbb{U}'.\ \mathbb{K} \xrightarrow{\mathbf{m}^\perp}_{\mathsf{I}} \mathbb{J} \qquad \mathbb{U} \xrightarrow{\mathbf{m}^\perp}_{wb} \mathbb{U}' \qquad \langle \mathbb{J}, \mathbb{W}'\rangle \mathcal{S} \langle \mathbb{J}, \mathbb{U}'\rangle}$$

Proof. We write $\mathbb{I} = \langle\!\langle \mathsf{t} \mid c_0[\mathsf{S}_0] \circ \mathsf{S} \mid \mathtt{I}_{\mathbb{I}} \rangle\!\rangle$; $\gamma_{\mathbb{I}}$, $\delta_{\mathbb{I}}\rangle$, $\mathbb{W} = \langle \sigma_{\mathbb{W}} \mid \eta_{\mathbb{W}} \mid \phi_{\mathbb{W}}\rangle$, $\mathbb{W}' = \langle \sigma_{\mathbb{W}'} \mid (\_, r_{\mathbb{W}'}, \_) \mid \phi_{\mathbb{W}'}\rangle$, $\mathbb{U} = \langle \sigma_{\mathbb{U}} \mid \eta_{\mathbb{U}} \mid \phi_{\mathbb{U}}\rangle$ and $\mathbb{K} = \langle \mathtt{I}_{\mathbb{K}}; \gamma_{\mathbb{K}}; \xi_{\mathbb{K}}\rangle$ We will also write $\sigma_0 \oplus \cdots \oplus \sigma_k$ and $\sigma'_0 \oplus \cdots \oplus \sigma'_p$ (for some $k, p \in \mathbb{N}^*$) for the respective decompositions of $\sigma_{\mathbb{W}}$ and $\sigma_{\mathbb{U}}$, as well as $\gamma_{\mathbb{K}}(c_i) = d_{i+1}[\mathsf{S}'_i]$ and $\gamma_{\mathbb{I}}(d_{i+1}) = c_{i+1}[\mathsf{S}_{i+1}]$ for $i \geq 0$.

We proceed by case analysis on $\mathbf{m}$.

Case $\mathbf{m} = \boldsymbol{c_0}.\langle \mathbf{ret}\, \mathsf{V} \mid \square\rangle^{\oplus}$.

We have $\mathsf{S}_0 = c_0[[]]$, $\mathsf{S} = []$, $\mathsf{t} = \mathbf{ret}\, \mathsf{V}$, $\delta_{\mathbb{I}} = \varnothing$ and $\eta_{\mathbb{W}} = \varnothing$, $r_{\mathbb{W}'} = \cdot$.

$\mathbb{I},\ \mathbb{W} \xrightarrow{\mathbf{m}} \langle \mathtt{I}_{\mathbb{I}};\ \gamma_{\mathbb{I}};\ \varnothing\rangle,\ \langle \sigma_{\mathbb{W}'} \mid \varnothing \mid \phi_{\mathbb{W}'}\rangle$

Since $\langle \mathbb{I}, \mathbb{W}\rangle \mathcal{S} \langle \mathbb{K}, \mathbb{U}\rangle$, then $\sigma'_0 = F(c_0[[]]) = [c_0]$ and $\xi_{\mathbb{K}} = \varnothing$.

Thus $\mathbb{U} \xrightarrow{\mathbf{m}^\perp}_{wb} \langle \sigma_{\mathbb{U}'} \mid \varnothing \mid \phi_{\mathbb{U}}\rangle = \mathbb{U}'$ and $\mathbb{K} \xrightarrow{\mathbf{m}^\perp}_{\mathsf{I}} \langle\!\langle \mathbf{ret}\, \mathsf{V} \mid \gamma_{\mathbb{K}}(c_0) \mid \mathtt{I}_{\mathbb{K}} \rangle\!\rangle;\ \gamma_{\mathbb{K}};\ \varnothing\rangle = \mathbb{J}$ with $\sigma_{\mathbb{U}'} = \sigma'_1 \oplus \cdots \oplus \sigma'_p$.

First, it is clear that $r_{\mathbb{W}'} \wedge f'_{\mathbb{U}}$, $r_{\mathbb{W}'} \wedge f'_{\mathbb{W}}$. Second, by synchronicity, we have $F(\gamma_{\mathbb{K}}(c_\ell)) \simeq_{\phi_{\mathbb{W}}} \sigma_\ell$ and $F(c_\ell[\mathsf{S}_\ell]) \simeq_{\phi_{\mathbb{U}}} \sigma'_\ell$, then if we take, forall $\ell \geq 0$, $c'_\ell = d_{\ell+1}$ and write $\gamma_{\mathbb{K}}(c_0) = c'_0[\mathsf{S}'_0]$, then forall $\ell \geq 0$, we get $F(c'_\ell[\mathsf{S}'_\ell]) \simeq_{\phi_{\mathbb{W}'}} \sigma_\ell$ and $F(\gamma_{\mathbb{I}}(c'_\ell)) \simeq_{\phi_{\mathbb{U}'}} \sigma'_{\ell+1}$, i.e . $\langle \mathbb{J}, \mathbb{W}'\rangle \mathcal{S} \langle \mathbb{J}, \mathbb{U}'\rangle$ since $\sigma_{\mathbb{W}} = \sigma_{\mathbb{W}'}$ and $\sigma_{\mathbb{U}'} = \sigma'_1 \oplus \cdots \oplus \sigma'_p$.

Case $\mathbf{m} = \boldsymbol{\kappa}.\langle \square[\mathbf{ret}\, \mathsf{V}] \mid d_0\rangle^{\oplus}$.

In this case, S_0 must be of the shape $\mathsf{S}[\kappa[[]]]$, $\mathsf{t} = \mathbf{ret}\, \mathsf{V}$, $\xi_{\mathbb{I}} = \varnothing$ and $\eta_{\mathbb{W}} = \varnothing$ and

$\mathbb{I},\ \mathbb{W} \xrightarrow{\mathbf{m}} \langle \mathtt{I}_{\mathbb{I}};\ \gamma_{\mathbb{I}} \cdot [d_0 \mapsto c_0[\mathsf{S}]];\ \varnothing\rangle,\ \langle [d_0] \oplus \sigma_{\mathbb{W}} \mid \varnothing \mid \phi_{\mathbb{W}}\rangle$

Since $\langle \mathbb{I}, \mathbb{W}\rangle \mathcal{S} \langle \mathbb{K}, \mathbb{U}\rangle$, then $F(c_0[\mathsf{S}[\kappa[[]]]]) \simeq_{\phi_{\mathbb{U}}} \sigma'_0$ and $\xi_{\mathbb{K}} = \varnothing$.

Subcase $F(c_0[\mathsf{S} \circ \kappa]) = \sigma'_0$.

There exists $\sigma''_0 = F(c_0[\mathsf{S}])$ s.t. $\sigma'_0 = \kappa :: \sigma''_0$,

We have $\mathbb{U} \xrightarrow{\mathbf{m}^\perp}_{wb} \langle \sigma''_0 \Rightarrow \sigma'_{\mathbb{U}} \mid \varnothing \mid \phi_{\mathbb{U}}\rangle = \mathbb{U}'$ that is $\sigma_{\mathbb{U}'} = \sigma''_0 \oplus \sigma'_1 \oplus \cdots \oplus \sigma'_p$ and $\mathbb{K} \xrightarrow{\mathbf{m}^\perp}_{\mathsf{I}} \langle\!\langle \mathbf{ret}\, \mathsf{V} \mid d_0[\gamma_{\mathbb{K}}(\kappa)] \mid \mathtt{I}_{\mathbb{K}} \rangle\!\rangle;\ \gamma_{\mathbb{K}};\ \varnothing\rangle = \mathbb{J}$.

Subcase $F(c_0[\mathsf{S} \circ \kappa]) \neq \sigma'_0$.

There exists $f \in \phi_{\mathbb{U}}$ s.t. $F(c_0[\mathsf{S} \circ \kappa]) = f \Rightarrow \sigma'_0$.

We have $\mathbb{U} \xrightarrow{\mathbf{m}^\perp}_{wb} \langle \sigma''_0 \Rightarrow \sigma'_{\mathbb{U}} \mid \varnothing \mid \phi_{\mathbb{U}}\rangle = \mathbb{U}'$ that is $\sigma_{\mathbb{U}'} = \sigma''_0 \oplus \sigma'_1 \oplus \cdots \oplus \sigma'_p$ where $f = \kappa :: f'$ and $\sigma''_0 = f' \Rightarrow \sigma'_0$ and $\mathbb{K} \xrightarrow{\mathbf{m}^\perp}_{\mathsf{I}} \langle\!\langle \mathbf{ret}\, \mathsf{V} \mid d_0[\gamma_{\mathbb{K}}(\kappa)] \mid \mathtt{I}_{\mathbb{K}} \rangle\!\rangle;\ \gamma_{\mathbb{K}};\ \varnothing\rangle = \mathbb{J}$.

In either subcase, let's write $d_0[\mathsf{S}''_0]$ for $d_0[\gamma_{\mathbb{K}}(\kappa)]$ and, forall $\ell \geq 0$, $\mathsf{S}''_{\ell+1} = \mathsf{S}'_\ell$.

First, it is clear that $r_{\mathbb{W}'} \wedge f'_{\mathbb{U}}, r_{\mathbb{W}'} \wedge f'_{\mathbb{W}}$ and that $F(d_0[\mathsf{S}''_0]) \simeq_{\phi_{\mathbb{W}'}} [d_0]$ and that $F(\gamma_{\mathbb{I}}(d_0)) \simeq_{\phi_{\mathbb{U}'}} \sigma''_0$, since $\phi_{\mathbb{U}} = \phi_{\mathbb{U}'}$, $\phi_{\mathbb{W}} = \phi_{\mathbb{W}'}$ and $\gamma_{\mathbb{I}}(d_0) = \gamma_{\mathbb{I}}(d_0) = c_0[\mathsf{S}]$.

Then we have, by synchronicity, forall $\ell \geq 0, F(\gamma_{\mathbb{K}}(c_\ell)) \simeq_{\phi_{\mathbb{W}}} \sigma_\ell$ and $F(c_\ell[\mathsf{S}_\ell]) \simeq_{\phi_{\mathbb{U}}} \sigma'_\ell$, that is, forall $\ell \geq 1, F(d_\ell[\mathsf{S}''_\ell]) \simeq_{\phi_{\mathbb{W}'}} \sigma_\ell$ and $F(\gamma_{\mathbb{I}}(d_\ell)) \simeq_{\phi_{\mathbb{U}'}} \sigma'_\ell$,

Thus we can conclude that $\langle \mathbb{J}, \mathbb{W}'\rangle \mathcal{S} \langle \mathbb{J}, \mathbb{U}'\rangle$.

Case $\mathbf{m} = \boldsymbol{c_0}.\langle \kappa :: r[e] \mid \square\rangle^{\oplus}$.

In this case, $\langle\!\langle \mathsf{t} \mid c_0[\mathsf{S}_0] \circ \mathsf{S} \rangle\!\rangle$ must be of the shape $\langle\!\langle \mathsf{e} \mid c_0[[]] \circ \mathsf{T} \oplus \mathsf{S} \rangle\!\rangle$,

Subcase $\mathsf{e} = \iota \# \mathbf{op}\, \mathsf{V}$.

$\mathbb{I},\ \mathbb{W} \xrightarrow{\mathbf{m}} \langle \mathtt{I}_{\mathbb{I}};\ \gamma_{\mathbb{I}};\ \mathbf{!}(e, \kappa :: r, \delta_{\mathbb{I}} \cdot [\kappa \mapsto \mathsf{T}])\rangle,\ \langle \sigma_{\mathbb{W}} \mid (f_{\mathbb{W}} \Rightarrow [\kappa], e) \mid \phi_{\mathbb{W}}\rangle$.

Since $\langle \mathbb{I}, \mathbb{W}\rangle \mathcal{S} \langle \mathbb{K}, \mathbb{U}\rangle$, then $F(c_0[[]]) \simeq_{\phi_{\mathbb{U}}} \sigma'_0$, i.e $\sigma'_0 = [c_0]$ and $\xi_{\mathbb{K}} = \mathbf{?}(e,\ r,\ \delta_{\mathbb{K}})$.

We have then $\mathbb{K} \xrightarrow{\mathbf{m}^\perp}_{\mathsf{I}} \langle\!\langle e \mid \gamma_{\mathbb{K}}(c_0) \circ \kappa[r\{\delta_{\mathbb{K}}\}] \mid \mathtt{I}_{\mathbb{K}} \rangle\!\rangle;\ \gamma_{\mathbb{K}};\ \mathbf{?}(e, \kappa :: r, \delta_{\mathbb{K}})\rangle = \mathbb{J}$ and $\mathbb{U} \xrightarrow{\mathbf{m}^\perp}_{wb} \langle \sigma'_1 \oplus \cdots \oplus \sigma'_p \mid (f_{\mathbb{U}}, e) \mid \phi_{\mathbb{U}}\rangle = \mathbb{U}'$

By synchronicity, we have $F(\gamma_{\mathbb{K}}(c_\ell)) \simeq_{\phi_{\mathbb{W}}} \sigma_\ell$ and $F(c_\ell[\mathsf{S}_\ell]) \simeq_{\phi_{\mathbb{U}}} \sigma'_\ell$, $r \wedge\ !f_{\mathbb{W}}$ and $r \wedge\ ?f_{\mathbb{U}}$ then if we take, forall $\ell \geq 0$, $c'_\ell = d_{\ell+1}$ and write $\gamma_{\mathbb{K}}(c_0) = c'_0[\mathsf{S}'_0]$, we get $\kappa :: r \wedge\ !f'_{\mathbb{U}}$, $\kappa :: r \wedge\ ?f_{\mathbb{W}'}$ and forall $\ell \geq 0$ we get $F(c'_\ell[\mathsf{S}'_\ell]) \simeq_{\phi_{\mathbb{W}'}} \sigma_\ell$ and $F(\gamma_{\mathbb{I}}(c'_\ell)) \simeq_{\phi_{\mathbb{U}'}} \sigma'_{\ell+1}$, i.e . $\langle \mathbb{J}, \mathbb{W}'\rangle \mathcal{S} \langle \mathbb{J}, \mathbb{U}'\rangle$ since $\sigma_{\mathbb{W}} = \sigma_{\mathbb{W}'}$ and $\sigma_{\mathbb{U}'} = \sigma'_1 \oplus \cdots \oplus \sigma'_p$.

Subcase $\mathsf{e} = e.\,\mathbb{I},\ \mathbb{W} \xrightarrow{\mathbf{m}} \langle \mathtt{I}_{\mathbb{I}};\ \gamma_{\mathbb{I}};\ \mathbf{?}(e, \kappa :: r, \delta_{\mathbb{I}} \cdot [\kappa \mapsto \mathsf{T}])\rangle,\ \langle \sigma_{\mathbb{W}} \mid (f_{\mathbb{W}} \Rightarrow [\kappa], e) \mid \phi_{\mathbb{W}}\rangle$.

Since $\langle \mathbb{I}, \mathbb{W}\rangle \mathcal{S} \langle \mathbb{K}, \mathbb{U}\rangle$, then $F(c_0[[]]) \simeq_{\phi_{\mathbb{U}}} \sigma'_0$, i.e $\sigma'_0 = [c_0]$ and $\xi_{\mathbb{K}} = \mathbf{!}(e, r, \delta_{\mathbb{K}})$.

We have then $\mathbb{K} \xrightarrow{\mathbf{m}^\perp}_{\mathsf{I}} \langle\!\langle \mathsf{e}\{\delta_{\mathbb{K}}\} \mid \gamma_{\mathbb{K}}(c_0) \circ \kappa[r\{\delta_{\mathbb{K}}\}] \mid \mathtt{I}_{\mathbb{K}} \rangle\!\rangle;\ \gamma_{\mathbb{K}};\ \mathbf{?}(e, \kappa :: r, \delta_{\mathbb{K}})\rangle = \mathbb{J}$ and $\mathbb{U} \xrightarrow{\mathbf{m}^\perp}_{wb} \langle \sigma'_1 \oplus \cdots \oplus \sigma'_p \mid (f_{\mathbb{U}}, e) \mid \phi_{\mathbb{U}}\rangle = \mathbb{U}'$

By synchronicity, we have $F(\gamma_{\mathbb{K}}(c_\ell)) \simeq_{\phi_{\mathbb{W}}} \sigma_\ell$ and $F(c_\ell[\mathsf{S}_\ell]) \simeq_{\phi_{\mathbb{U}}} \sigma'_\ell$, $r \wedge\ !f_{\mathbb{W}}$ and $r \wedge\ ?f_{\mathbb{U}}$ then if we take, forall $\ell \geq 0$, $c'_\ell = d_{\ell+1}$ and write $\gamma_{\mathbb{K}}(c_0) = c'_0[\mathsf{S}'_0]$, we get $\kappa :: r \wedge\ ?f'_{\mathbb{U}}$, $\kappa :: r \wedge\ !f_{\mathbb{W}'}$ and forall $\ell \geq 0$ we get $F(c'_\ell[\mathsf{S}'_\ell]) \simeq_{\phi_{\mathbb{W}'}} \sigma_\ell$ and $F(\gamma_{\mathbb{I}}(c'_\ell)) \simeq_{\phi_{\mathbb{U}'}} \sigma'_{\ell+1}$, i.e . $\langle \mathbb{J}, \mathbb{W}'\rangle \mathcal{S} \langle \mathbb{J}, \mathbb{U}'\rangle$ since $\sigma_{\mathbb{W}} = \sigma_{\mathbb{W}'}$ and $\sigma_{\mathbb{U}'} = \sigma'_1 \oplus \cdots \oplus \sigma'_p$.

Case $\mathbf{m} = \boldsymbol{\kappa}.\langle \square[r[e]] \mid d_0\rangle^{\oplus}$. Similar reasoning applies.

□

Corollary D.11. *Given $\mathbb{J} \in \mathrm{Confs}_{\mathsf{pas}}$, $\mathbb{I} \in \mathrm{Confs}_{\mathsf{act}}$, $\mathbb{W}$, $\mathbb{U} \in \mathcal{W}$ and a trace t such that $\langle \mathbb{J}, \mathbb{W}\rangle \mathcal{S} \langle \mathbb{I}, \mathbb{U}\rangle$, $\mathsf{t} \in \mathbf{Tr}_{\mathcal{L}_{\mathsf{AI}}}(\mathbb{I})$ and $\mathbf{Tr}_{\mathcal{L}_{\mathsf{AI}}}(\mathbb{J})$ then:*

$$\mathsf{t} \in \mathbf{Tr}_{\mathcal{L}_{wb}}(\mathbb{W}) \wedge \mathsf{t}^\perp \in \mathbf{Tr}_{\mathcal{L}_{wb}}(\mathbb{U})$$

Proof. By induction on the length of t using Lemma D.10. □

Definition D.12 (Composability). Given a passive configuration $\mathbb{G} \in O^{\ominus}$ and an active one $\mathbb{H} \in O^{\oplus}$ such that $\mathbb{G} = \langle \mathbb{I} \parallel \mathbb{T} \parallel \mathbb{W}\rangle$ and $\mathbb{H} = \langle \mathbb{J} \parallel \mathbb{S} \parallel \mathbb{U}\rangle$, we say that $\mathbb{G}$ and $\mathbb{H}$ are *closed composable* and write $\mathbb{G} \perp\!\!\!\perp \mathbb{H}$ when:

$$\frac{\mathbb{G} = \langle \mathbb{I} \parallel \mathbb{T} \parallel \mathbb{W}\rangle \qquad \mathbb{H} = \langle \mathbb{J} \parallel \mathbb{S} \parallel \mathbb{U}\rangle \qquad \langle \mathbb{I}, \mathbb{W}\rangle \mathcal{S} \langle \mathbb{J}, \mathbb{U}\rangle}{\mathbb{G} \perp\!\!\!\perp \mathbb{H}}$$

Remark 4. *The preservation of* closed composability *by configuration synchronization follows from lemmas A.2 and D.10.*

D.3.2 OGS *Observation.* Now we can semantically capture $\mathtt{S}[\mathtt{t}\{v\}] \Downarrow_{\mathrm{op}} \langle\rangle$ by the following predicate.

Definition D.13 (Observation predicate). Taking a passive configuration $\mathbb{G} \in O^{\ominus}$ and an active one $\mathbb{H} \in O^{\oplus}$ such that $\mathbb{G} \perp\!\!\!\perp \mathbb{H}$, we define the observation of $\mathbb{G}$ on $\mathbb{H}$ as the predicate:

$$\langle \mathbb{G} \perp\!\!\!\perp \mathbb{H} \rangle \Downarrow [c]\,\mathbf{ret}\,\langle\rangle \;:\Longleftrightarrow\; \exists \mathbb{K}, \gamma.\, \langle \mathbb{J} \dashv\vdash \mathbb{I} \rangle \Downarrow_{\mathrm{ci}} \langle \mathbb{K} \dashv\vdash c[\mathbf{ret}\,\langle\rangle]; \gamma; \bot \rangle$$

Now, that we have given a sensible notion of observation at the level of OGS configurations, it remains to show what does this predicate mean at the level of OGS interpretations.

Theorem D.14 (full observation). *Given two composable configurations $\mathbb{G}$ and $\mathbb{H}$, we have:*

$$\langle \mathbb{H} \perp\!\!\!\perp \mathbb{G} \rangle \Downarrow [c]\,\mathbf{ret}\,\langle\rangle$$

if and only if

$$\exists \mathtt{t} \in \mathbf{CTr}_{\mathrm{ogs}}(\mathbb{G}).\ \mathtt{t}^{\perp}\, \boldsymbol{c}.\langle \mathbf{ret}\,\langle\rangle \mid \square \rangle^{\oplus} \in \mathbf{CTr}_{\mathrm{ogs}}(\mathbb{H})$$

Proof. We write $\mathbb{G} = \langle \mathbb{I} \parallel \mathbb{T} \parallel \mathbb{W} \rangle$ and $\mathbb{H} = \langle \mathbb{J} \parallel \mathbb{S} \parallel \mathbb{U} \rangle$ and $\mathbb{I} = \langle (\mathtt{t} \mid \mathtt{I}); \gamma_{\mathbb{I}}; \xi \rangle$

Left to right. Suppose there exist $\mathbb{K}, \mathbb{J} \in \mathcal{A}$ and γ such that $\langle \mathbb{J} \dashv\vdash \mathbb{I} \rangle \Downarrow_{\mathrm{ci}} \langle \mathbb{K} \dashv\vdash \mathbb{J} \rangle$ with $\mathbb{J} = \langle (c[\mathbf{ret}\,\langle\rangle] \mid \mathtt{I}); \gamma; \varnothing \rangle$.

By corollary A.5 and definition A.3, there exists an odd-length trace $\mathtt{t}$ such that $\mathbb{I} \overset{\mathtt{t}}{\Rightarrow}_{\mathrm{I}} \mathbb{K}$ and $\mathbb{J} \overset{\mathtt{t}^{\perp}}{\Longrightarrow}_{\mathrm{I}} \mathbb{J} \xrightarrow{\boldsymbol{c}.\langle \mathbf{ret}\,\langle\rangle \mid \square \rangle^{\oplus}}_{\mathrm{I}} \langle \gamma; \varnothing \rangle = \mathbb{J}'$. By lemma D.3, there exist $\mathbb{T}', \mathbb{S}', \mathbb{S}'' \in \mathcal{T}$ s.t. $\mathbb{T} \xrightarrow{\mathtt{t}}_{\mathcal{T}} \mathbb{T}'$, $\mathbb{S} \xrightarrow{\mathtt{t}^{\perp}}_{\mathcal{T}} \mathbb{S}' \xrightarrow{\boldsymbol{c}.\langle \mathbf{ret}\,\langle\rangle \mid \square \rangle^{\oplus}}_{\mathcal{T}} \mathbb{S}''$, $\mathbb{K} \between \mathbb{T}'$, $\mathbb{J}' \between \mathbb{S}'$ and $\mathbb{J}' \between \mathbb{S}''$. Similarly, corollary D.11 ensures the existence of $\mathbb{W}', \mathbb{U}', \mathbb{U}'' \in \mathcal{W}$ s.t. $\mathbb{W} \xrightarrow{\mathtt{t}}_{wb} \mathbb{W}'$, $\mathbb{U} \xrightarrow{\mathtt{t}^{\perp}}_{wb} \mathbb{U}' \xrightarrow{\boldsymbol{c}.\langle \mathbf{ret}\,\langle\rangle \mid \square \rangle^{\oplus}}_{wb} \mathbb{U}''$ whereas lemma D.10 ensures that synchronicity is preserved and that in particular $\langle \mathbb{J}, \mathbb{U}' \rangle \mathrel{\mathcal{S}} \langle \mathbb{K}, \mathbb{W}' \rangle$.

Thus we have shown that $\mathbb{G} \overset{\mathtt{t}}{\underset{\mathrm{ogs}}{\Rightarrow}} \langle \mathbb{K} \parallel \mathbb{T}' \parallel \mathbb{W}' \rangle$ and $\mathbb{H} \overset{\mathtt{t}^{\perp}}{\underset{\mathrm{ogs}}{\Longrightarrow}} \langle \mathbb{J} \parallel \mathbb{S}' \parallel \mathbb{U}' \rangle \overset{\boldsymbol{c}.\langle \mathbf{ret}\,\langle\rangle \mid \square \rangle^{\oplus}}{\underset{\mathrm{ogs}}{\Longrightarrow}} \langle \mathbb{J}' \parallel \mathbb{S}'' \parallel \mathbb{U}'' \rangle$, from which we can deduce that $\mathtt{t} \in \mathbf{Tr}_{\mathrm{ogs}}(\mathbb{G})$ and $\mathtt{t}^{\perp}\, \boldsymbol{c}.\langle \mathbf{ret}\,\langle\rangle \mid \square \rangle^{\oplus} \in \mathbf{Tr}_{\mathrm{ogs}}(\mathbb{H})$.

Right to left. Let $\mathtt{t} \in \mathbf{Tr}_{\mathrm{ogs}}$ such that $\mathtt{t} \in \mathbf{Tr}_{\mathrm{ogs}}(\mathbb{G})$ and $\mathtt{t}^{\perp}\, \boldsymbol{c}.\langle \mathbf{ret}\,\langle\rangle \mid \square \rangle^{\oplus} \in \mathbf{Tr}_{\mathrm{ogs}}(\mathbb{H})$.

If we write $\mathbb{G} \overset{\mathtt{t}}{\underset{\mathrm{ogs}}{\Rightarrow}} \langle \mathbb{K} \parallel \mathbb{T}' \parallel \mathbb{W}' \rangle$ and for some $\mathbb{W}' \in \mathcal{W}$ with $\mathbb{K} = \langle \mathtt{I}_{\mathbb{K}}; \gamma_{\mathbb{K}}; \varnothing \rangle \in \mathcal{A}$, and write $\mathbb{H} \overset{\mathtt{t}^{\perp}}{\underset{\mathrm{ogs}}{\Longrightarrow}} \langle \mathbb{J} \parallel \mathbb{S}' \parallel \mathbb{U}' \rangle \xrightarrow{\boldsymbol{c}.\langle \mathbf{ret}\,\langle\rangle \mid \square \rangle^{\oplus}}_{\mathrm{ogs}} \mathbb{E}$ for some $\mathbb{S}' \in \mathcal{T}$, $\mathbb{U}' \in \mathcal{W}$ and $\mathbb{E} \in O$ with $\mathbb{J} = \langle (c[\mathbf{ret}\,\langle\rangle] \mid \mathtt{I}); \gamma; \varnothing \rangle$, then by corollary A.5, we have $\langle \mathbb{J} \dashv\vdash \mathbb{I} \rangle \mapsto^{*}_{\mathrm{ci}} \langle \mathbb{J} \dashv\vdash \mathbb{K} \rangle$ and again by lemma A.2, we have $\mathbb{J} \dashv\vdash_{c} \mathbb{K}$, and in particular $c \notin \mathrm{dom}(\gamma_{\mathbb{J}})$, thus $\langle \mathbb{J} \dashv\vdash \mathbb{K} \rangle \not\mapsto_{\mathrm{ci}}$. Finally, we have shown that $\langle \mathbb{J} \dashv\vdash \mathbb{I} \rangle \Downarrow_{\mathrm{ci}} \langle \mathbb{J} \dashv\vdash \mathbb{K} \rangle$ and consequently $\langle \mathbb{H} \perp\!\!\!\perp \mathbb{G} \rangle \Downarrow [c]\,\mathbf{ret}\,\langle\rangle$.

□

An interactive type-checker for dependent types with general recursion (System Description)

Valentin Pasquale
valentin.pasquale@cea.fr
CEA List, Université Paris-Saclay
France

Álvaro García-Pérez
alvaro.garciaperez@cea.fr
CEA List, Université Paris-Saclay
France

ABSTRACT

In this system description, we present an interactive type-checker for Hutchins' Pure Subtype Systems (PSS). By blurring the distinction between types and terms, PSS features dependent types and general recursion, but the resulting theory is highly impredicative and not strongly normalising, which poses a number of known challenges for practical implementations of type-checking. We propose an interactive type-checker where the programmer drives the type-checking process by performing different actions on the program term. Our interactive type-checker is based on Emacs and is written in Emacs Lisp. We provide a motivating example involving the type-checking of a safe division function, and we show how our type-checker certifies this function is safe by implementing a rudimentary abstract interpretation on top of PSS type-checking.

ACM Reference Format:
Valentin Pasquale and Álvaro García-Pérez. 2025. An interactive type-checker for dependent types with general recursion (System Description) . In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In this system description, we present an interactive type-checker for Hutchins' Pure Subtype Systems (PSS for short), which are based on a novel and intriguing typing discipline that blurs the distinction between types and terms [12].[1] PSS (hereafter in the singular) features higher-order subtyping, dependent types, and general recursion. These features combined allow for great expressivity and enable a number of advanced language features for genericity and extensibility. However, PSS is highly impredicative and not strongly normalising, which poses a number of known metatheoretical challenges, the most relevant for our work here being that type-checking in PSS is undecidable in general.[2]

Our type-checker circumvents this undecidability by letting the programmer drive the type-checking process. Our system allows one to type-check PSS terms, where some parts of the algorithm are automatic, and other parts rely on the interactions with the programmer, who likely knows how to type-check their program.

Consider the following program:

$$\begin{aligned} fact(0) &= 1 \\ fact(n+1) &= (n+1) * fact(n) \\ safeDiv(n,m) &= n/fact(m) \end{aligned}$$

With PSS, our type-checker could certify that *safeDiv* never incurs a division-by-zero error, since the factorial function always returns a result greater than one. Certifying this static guarantee is not direct in conventional type systems (like Haskell or ML). We provide suitable encodings for the naturals and show how to implement a rudimentary abstract interpretation in PSS in a straightforward fashion. Thanks to PSS impredicativity, we can apply the successor function to the type of naturals, thus obtaining a domain of interpretation that represents all the naturals produced by the successor function, which are exactly those that are greater than zero. Then we show that the value returned by the factorial function is a subtype of the said domain of interpretation.

Our focus on recursive implementations of factorial, which is itself primitive recursive, showcases the ability of our approach to be generalised to general recursive functions, and functions that may not terminate. Consider the program below:

$$\begin{aligned} Syracuse(0) &= 1 \\ Syracuse(1) &= 1 \\ Syracuse(n) &= Syracuse(n/2) && \text{(if } n \text{ is even)} \\ Syracuse(n) &= Syracuse((3*n)+1) && \text{(otherwise)} \\ safeDiv2(n,m) &= n/Syracuse(m) \end{aligned}$$

It is a variation of the Syracuse function, which is traditionally not defined for 0. It is not known whether *Syracuse* terminates on every natural number or not. However, if the function terminates, then it always returns 1. Our type-checker can also certify that *safeDiv2* above is safe.

We focus our discussion mainly on the recursive implementation of *fact*—rather than on the implementation of *Syracuse*—because the former involves applying arithmetic operations on its recursive call, which is not the case for the type-checking derivation of the latter.[3]

[1] Our implementation is open source and available on GitHub at the following address: https://github.com/valentin-ppp/interactive-dependenttypes-typechecker.
[2] In a related ongoing work, the authors study PSS meta-theory, notably its type safety and the conditions on which type-checking is decidable.

[3] We cover the type-checking of *Syracuse* in Section 4.5, but this is not central to the bulk of our paper

PPDP '25, September 10–11, 2025, Rende, Italy

ACM ISBN 979-8-4007-2085-7…$15.00
https://doi.org/XXXXXXX.XXXXXXX

In a superficial way, our type-checker resembles an interactive theorem prover. However, due to the lack of strong normalisation, whether PSS corresponds to some logic by the Curry-Howard isomorphism is still an open problem, and as a consequence our type-checker is not suited for doing mechanised mathematics right away.

The remainder of the paper is organised as follows. Section 2 recalls PSS (with minor modifications that we comment on in detail) and the encodings of different objects including the naturals, which we dub "integers" as conventionally in programming. Section 3 comments on our design and implementation choices, and describes how our interactive type-checker is used. Section 4 showcases the type-checking of different programs, including those in the motivating examples above. Section 5 concludes by enumerating related work and by discussing avenues for future work.

2 PRELIMINARIES

In this section, we present the theory we use, which is a slight variation of the original PSS theory. We then present how we encode objects in PSS, and finally, we detail the changes we made to the original PSS theory.

2.1 The PSS theory

PSS blurs the distinction between types and terms and has numerous interesting features explained at length in Hutchins' PhD thesis [12], which we do not explore in this paper. Type-checking is known to be challenging. In this paper, we focus only on practical approaches to type-checking PSS programs. PSS is an extension of the pure lambda calculus with a constant symbol Top for the most general supertype, and with subtype annotations in the formal parameters of abstractions. Assume a countably infinite set of variables V ranged over by $x, y, \ldots$; the syntax of terms and contexts of PSS, extended with a fixed point combinator and a variant type (which we describe in a moment), is given by the following grammar:

$$\begin{array}{rcl} t, u, v, \ldots \in \Lambda & ::= & x \mid \mathsf{Top} \mid (\lambda x \leq t.u) \mid (u\,v) \mid (\mathsf{Y}\,u) \mid \mathsf{Or}(u, v) \\ \Gamma & ::= & \varepsilon \mid \Gamma, x \leq t \end{array}$$

The type-checking process of PSS ensures that for every application $(u\,v)$ of a given term, v is a subtype of the type annotation of a function of which u is a subtype.

In our theory, the program we wish to type-check is the following:

$$\begin{array}{rcl} \mathit{fact} & = & \mathsf{Y}\lambda s \leq (\lambda x \leq \mathit{int}.\mathit{int}).\, \lambda n \leq \mathit{int}. \\ & & \mathit{match}\, n\, \mathit{one}\, (\lambda p \leq \mathit{int}.\mathit{mult}\, n\, (s\, p)) \\ \mathit{safediv} & = & \lambda n \leq \mathit{int}.\lambda m \leq \mathit{int}.\mathit{div}\, n\, (\mathit{fact}\, m) \\ \mathit{div} & = & \lambda n \leq \mathit{int}.\lambda m \leq (\mathit{succ}\, \mathit{int}).\ \cdots \end{array}$$

where *one* and *int* are respectively the term for the integer 1 and for the integer type, *mult* and *succ* stand for the arithmetic operations of the same names, and *match* implements the pattern matching on an integer being either 0 or a successor. The expressions of these terms are defined in Section 2.2.

In this theory, we consider a static condition of *well-formedness* that prescribes that every application in a term is well-typed, and it is an open problem to show it ensures type-safety.[4] Verifying the well-formedness of a term is akin to type-checking a term in other theories. In the rest of the paper, we refer to the process of verifying the well-formedness of a term as type-checking.

Well-formedness of a term is defined by the rules in Figure 1. Rule Wf-Var states that a variable x is well-formed if it is well-scoped. Rule Wf-Top states that the term Top is always well-formed. Rule Wf-Fun states that a function $\lambda x \leq t.u$ is well-formed if and only if both its body u and its type annotation t are well-formed. Rule Wf-App states that an application $u\,v$ is well-formed if u is a subtype of a function $\lambda x \leq t.\mathsf{Top}$, such that v is a subtype of t. There are two challenging things to do when type-checking an application: (i) finding a suitable term t, and (ii) proving the subtype relations prescribed by the rules. As for (i), finding such t is challenging since the rule Wf-App is not syntax-directed. The algorithm described in [12] finds such t under certain conditions, but in general, this algorithm does not terminate. As for (ii), proving the subtype relation is undecidable in general, since it is undecidable to check if two terms are equivalent, which is a subproblem of checking if they are subtypes.

Term well-formedness $\boxed{\Gamma \vdash t\ \mathsf{wf}}$

$$\frac{x \leq t \in \Gamma}{\Gamma \vdash x\ \mathsf{wf}}\ \textsc{Wf-Var} \qquad \frac{}{\Gamma \vdash \mathsf{Top}\ \mathsf{wf}}\ \textsc{Wf-Top}$$

$$\frac{\Gamma \vdash t\ \mathsf{wf} \qquad \Gamma, x \leq t \vdash u\ \mathsf{wf}}{\Gamma \vdash \lambda x \leq t.u\ \mathsf{wf}}\ \textsc{Wf-Fun}$$

$$\frac{\Gamma \vdash u\ \mathsf{wf} \qquad \Gamma \vdash v\ \mathsf{wf} \qquad \Gamma \vdash t\ \mathsf{wf} \qquad \Gamma \vdash u \leq \lambda x \leq t.\mathsf{Top} \qquad \Gamma \vdash v \leq t}{\Gamma \vdash u\,v\ \mathsf{wf}}\ \textsc{Wf-App}$$

Figure 1: Well-formedness rules in PSS

Now we describe subtyping in PSS, which is collected by the rules in Figure 2. These rules describe two reduction relations: a *subtype reduction*, which models promotion steps, and an *equivalence reduction*, which models β-reduction.

Rule Leq-Def defines the subtyping relation ($\leq$) such that a is a subtype of b if and only if there exists a term c such that a is a subtype of c and b is equivalent to c. We represent in the rest of the paper the subtype reduction vertically and the equivalence reduction horizontally, such that the diagram below represents the subtyping relation $a \leq b$:

$$\begin{array}{ccc} b & \xrightarrow{\equiv\!\!\gg} & c \\ & & \leq \uparrow\!\uparrow \\ & & a \end{array}$$

We now describe the subtype reduction and the equivalence reduction. Rule Eq-Refl ensures that the equivalence reduction

[4] The fact that well-formedness implies type-safety is an open problem since the introduction of PSS by Hutchins [12], which we are actively investigating in a separate work.

relation is reflexive. Rule Eq-Beta instruments the β-reduction that can happen in any context $C_{\equiv}$ thanks to rule Eq-Ctx. Rule Sub-Top states that any term is a subtype of Top. Rule Sub-Prom states that if $x \leq t \in \Gamma$ (meaning x has subtype t in context Γ), then x is a subtype of t. Rule Sub-Eq ensures that any equivalence reduction is also a subtype reduction. Rule Sub-CtxFun states that a subtype reduction can happen in the body of a function, with its variable x linked to the subtype annotation t in the context Γ. Rules Sub-Top, Sub-Prom, Sub-Eq and Sub-CtxFun also happen in a context $C_{\leq}$ thanks to rule Sub-Ctx.

$$\frac{\Gamma \vdash a \xrightarrow{\leq} c \qquad \Gamma \vdash b \xrightarrow{\equiv} c}{\Gamma \vdash a \leq b}\ \textsc{Leq-Def}$$

$$\frac{}{\Gamma \vdash u \xrightarrow{\equiv} u}\ \textsc{Eq-Refl} \qquad \frac{}{\Gamma \vdash (\lambda x \leq t.u)\, v \xrightarrow{\equiv} u[x \backslash v]}\ \textsc{Eq-Beta}$$

$$\frac{\Gamma \vdash u \xrightarrow{\equiv} v}{\Gamma \vdash C_{\equiv}[u] \xrightarrow{\equiv} C_{\equiv}[v]}\ \textsc{Eq-Ctx} \qquad \frac{}{\Gamma \vdash u \xrightarrow{\leq} \mathrm{Top}}\ \textsc{Sub-Top}$$

$$\frac{x \leq t \in \Gamma}{\Gamma \vdash x \xrightarrow{\leq} t}\ \textsc{Sub-Prom} \qquad \frac{\Gamma \vdash u \xrightarrow{\equiv} v}{\Gamma \vdash u \xrightarrow{\leq} v}\ \textsc{Sub-Eq}$$

$$\frac{\Gamma, x \leq t \vdash u \xrightarrow{\leq} v}{\Gamma \vdash \lambda x \leq t.u \xrightarrow{\leq} \lambda x \leq t.v}\ \textsc{Sub-CtxFun}$$

$$\frac{\Gamma \vdash u \xrightarrow{\leq} v}{\Gamma \vdash C_{\leq}[u] \xrightarrow{\leq} C_{\leq}[v]}\ \textsc{Sub-Ctx}$$

$$C_{\equiv} ::= \square \mid (\lambda x \leq C_{\equiv}.t) \mid (\lambda x \leq t.C_{\equiv}) \mid (C_{\equiv}\ t) \mid (t\ C_{\equiv}) \mid$$
$$(\mathrm{Y}\ C_{\equiv}) \mid \mathrm{Or}(C_{\equiv}, t) \mid \mathrm{Or}(t, C_{\equiv})$$

$$C_{\leq} ::= \square \mid (C_{\leq}\ t) \mid (t\ C_{\leq}) \mid (\mathrm{Y}\ C_{\leq}) \mid \mathrm{Or}(C_{\leq}, t) \mid \mathrm{Or}(t, C_{\leq})$$

Figure 2: Subtyping rules in PSS

Our system is slightly different from the original PSS theory in the following ways: In the original PSS theory, the type system allows terms to be promoted only in the following context: $C_{PSS} ::= \square \mid (\lambda x \leq t.C_{PSS}) \mid (C_{PSS}\ t)$. In this paper, we extend the original PSS contexts with $(t\ C_{\leq})$, on which we will elaborate in a moment.

We disallow the following context, $(\lambda x \leq C_{\leq}.t)$. The original theory is invariant; therefore, we keep the invariant property in our theory.

The new allowed context is $(t\ C_{\leq})$. We allow promotions to happen in this new allowed context, under certain conditions, to allow more expressivity for the programmer. The condition is the following: for this new allowed context to be used in a promotion step, there must be no instances of variables in any subtype annotation anywhere in the term t. The above condition is an over-approximation of the actual set of terms for which it is problematic to promote in this new allowed context. The above condition is also a major change to the theory, as simple terms such as $\lambda x \leq \mathrm{Top}.\lambda y \leq x.y$ are disallowed in some promotion steps. Under the constraint described above, no forbidden promotions can happen; as a result, we conjecture that the metatheory is still correct. All the examples of Section 4 do follow the constraint mentioned above, where this new allowed context is used extensively, which is crucial for the expressivity of the theory.

Moreover, because the programmer drives the subtyping process, allowing more contexts and therefore making these derivation rules less algorithmic is not problematic; yet it is useful for the programmer to have increased expressivity.

2.2 How we represent objects in this theory

To program in this theory, we need to represent some objects. We either add primitive operators to the theory or encode the objects within the calculus when possible. This section presents all the different objects.

General recursion can be implemented with the aid of a fixed point combinator. Such a fixed point combinator could be written in our calculus thanks to a family of looping terms [12, p. 46]. Such a family results in a term 40 pages long [10]. Therefore, in order to simplify the implementation, we add a primitive fixed point combinator to the theory whose typing and reduction rules are the following:

$$\frac{}{\Gamma \vdash \mathrm{Y}\, f \xrightarrow{\equiv} f\ (\mathrm{Y}\, f)}\ \textsc{Equiv-Y}$$

$$\frac{\Gamma \vdash f\ \mathrm{wf} \qquad \Gamma \vdash t\ \mathrm{wf} \qquad \Gamma \vdash f \leq \lambda x \leq t.\mathrm{Top} \qquad \Gamma \vdash \mathrm{Y}f \leq t}{\Gamma \vdash \mathrm{Y}f\ \mathrm{wf}}\ \textsc{Wf-Y}$$

We add a primitive variant type to the theory, which represents a possible choice between two or more terms. Given two terms a and b, the variant of these two terms is written $\mathrm{Or}(a, b)$ and indicates a choice between a and b. We add the above construction to model a domain of interpretation of possible values, in the spirit of abstract interpretation. An area of future work would be to add more similar constructs to enrich the expressivity of the language, for instance a set difference operator. Figure 3 presents our encoding of integers with our variant operator being crucial to the encoding.

The subtyping and well-formedness rules of variants are the following:

$$\frac{\Gamma \vdash a \leq b}{\Gamma \vdash a \leq \mathrm{Or}(b, c)}\ \textsc{Leq-OrR1} \qquad \frac{\Gamma \vdash a \leq c}{\Gamma \vdash a \leq \mathrm{Or}(b, c)}\ \textsc{Leq-OrR2}$$

$$\frac{\Gamma \vdash a \leq c \qquad \Gamma \vdash b \leq c}{\Gamma \vdash \mathrm{Or}(a, b) \leq c}\ \textsc{Leq-OrLeft}$$

$$\frac{\Gamma \vdash f\ a \xrightarrow{\equiv} a' \qquad \Gamma \vdash f\ b \xrightarrow{\equiv} b'}{\Gamma \vdash f\ \mathrm{Or}(a, b) \xrightarrow{\equiv} \mathrm{Or}(a', b')}\ \textsc{Eq-Or}$$

$$\frac{\Gamma \vdash a\ \mathrm{wf} \qquad \Gamma \vdash b\ \mathrm{wf}}{\Gamma \vdash \mathrm{Or}(a, b)\ \mathrm{wf}}\ \textsc{Wf-Or}$$

To preserve the expressivity of variants, in the implementation, we disallow any equivalence reduction that would duplicate a variant. For instance, consider the term $(\lambda x \leq \mathsf{Top}.x\ x)\ (\mathsf{Or}(a, b))$, which represents the choice between $(\lambda x \leq \mathsf{Top}.x\ x)\ a$ (which is $(a\ a)$) and $(\lambda x \leq \mathsf{Top}.x\ x)\ b$ (which is $(b\ b)$). If we β-reduce the term $(\lambda x \leq \mathsf{Top}.x\ x)\ (\mathsf{Or}(a, b))$ to $(\mathsf{Or}(a, b)\ \mathsf{Or}(a, b))$, the resulting expression can lead to terms like $(a\ b)$, which were not possible outcomes of the original expression that only allowed $(a\ a)$ or $(b\ b)$. In fact, the term obtained after the β-reduction $(\mathsf{Or}(a, b)\ \mathsf{Or}(a, b))$ is a supertype of the original term $(\lambda x \leq \mathsf{Top}.x\ x)\ (\mathsf{Or}(a, b))$. Therefore, because the duplication of a variant is similar to a promotion in our calculus, we disallow any duplication of a variant during equivalence steps: if an equivalence reduction step duplicates a variant object, then it is a subtype reduction step, and not an equivalence reduction step.

For all other objects, we use a slight variation of traditional Scott encoding for the terms [8, 18], and we use an encoding reminiscent of algebraic data types for the types [4]. We first present these encodings, and then discuss our choices.

For terms, other encodings exist, but we use the Scott encoding to let the programmer pattern match on the objects they manipulate, which makes practical programming with these objects easier. For instance, the integers 0 and 1 are represented in Figure 3.

$$\begin{array}{rcl} \mathit{int} & = & \mathsf{Y}\lambda f \leq \mathsf{Top}.\lambda z \leq \mathsf{Top}.\lambda s \leq (\lambda m \leq f.\mathsf{Top}).\mathsf{Or}(z, s\ f) \\ \mathit{zero} & = & \lambda z \leq \mathsf{Top}.\lambda s \leq (\lambda m \leq \mathit{int}.\mathsf{Top}).z \\ \mathit{one} & = & \lambda z \leq \mathsf{Top}.\lambda s \leq (\lambda m \leq \mathit{int}.\mathsf{Top}).s\ \mathit{zero} \end{array}$$

Figure 3: Scott encoding of integers

With the above encoding, the function *match*, which takes an integer and two choices, one for the case where n is "zero" and one for the other case, is defined as follows:

$$\mathit{match} \quad = \quad \lambda n \leq \mathit{int}.\lambda z \leq \mathsf{Top}.\lambda f \leq (\lambda m \leq \mathit{int}.\mathsf{Top}).n\ z\ f$$

The successor function is defined as follows:

$$\mathit{succ} \quad = \quad \lambda n \leq \mathit{int}.\lambda z \leq \mathsf{Top}.\lambda s \leq (\lambda m \leq \mathit{int}.\mathsf{Top}).s\ n$$

To write the "type" of our objects, we use an algebraic data type encoding to exactly model the possible values of the object: only terms that follow this algebraic data type encoding will be subtypes of this type. For instance, integers are defined recursively as either 0 or a successor of themselves. As a result, our type *int* models all these possible values thanks to our variant construct described in Section 2.2. Our approach is similar to what an iso-recursive type system [16] would be, in the sense that in practical use, the programmer needs to indicate to the type-checker where to fold and unfold the recursive type definition. Figure 4, for instance, shows that *one* is a subtype of *int* thanks to one unfolding, which in our system is an expansion of the fixed point combinator. Our encoding of recursive types resembles the conventional algebraic data type definitions, where $\mu\alpha.\lambda z.\lambda s.z + s\ \alpha$ turns into $\mathsf{Y}\lambda f \leq \mathsf{Top}.\lambda z \leq \mathsf{Top}.\lambda s \leq (\lambda m \leq f.\mathsf{Top}).\mathsf{Or}(z, s\ f)$.

We now explain our differences from the traditional Scott encoding.

The first difference is that the traditional encoding takes a carrier, and ours does not. For instance, the term *zero* is often represented as follows:

$$\lambda c \leq \mathsf{Top}.\lambda z \leq c.\lambda s \leq (\lambda m \leq \mathit{int}.c).z$$

While we acknowledge that letting the programmer provide a carrier is useful for automatic type-checking, we are not interested in automatic type-checking, and as such, our terms do not take a carrier as a parameter as it lengthens every term, which makes the process of driving the type-checker harder. Removing the carrier has other advantages, especially with booleans, one of which is described at the end of this section.

The second difference from the traditional Scott encoding is our type *int*. In typed lambda calculus, when the Scott encoding is used, the integer type (with a carrier) is usually represented as follows:

$$\mathit{wrongint} = \mathsf{Y}\lambda f \leq \mathsf{Top}.\lambda c \leq \mathsf{Top}.\lambda z \leq c.\lambda s \leq (\lambda m \leq f.c).c$$

The above Scott integer type *wrongint* is pathological for our use case for three reasons: First, some terms, which are not integers, are subtypes of this term. For instance, consider the following term:

$$\mathit{notanint} = \lambda c \leq \mathsf{Top}.\lambda z \leq c.\lambda s \leq (\lambda m \leq \mathit{wrongint}.c).s\ \mathit{wrongint}$$

By promoting the variable s to its subtype annotation, the body of the function becomes $(\lambda m \leq \mathit{wrongint}.c)\ \mathit{wrongint}$ which β-reduces to *wrongint*, and as such we have that *notanint* is a subtype of *wrongint*.

Second, addition is not commutative on these "incorrect" integers. For instance, consider the standard definition of addition for Scott integers, which takes two parameters, pattern matches on the first one to either return the second parameter if the first parameter is 0, or return the successor of self applied to some arguments. Because of the Scott encoding, the way to pattern match on the first argument is to apply it to both what should be done for the 0 case, and for the successor case. Consider therefore the term *plus wrongint one*, this term is actually *wrongint* applied to some other terms specifying what to do if it is 0 or a successor, but in any case the term *wrongint* always returns the carrier c when applied. The term *plus wrongint one* hence reduces to *wrongint*. Whereas, in contrast, the term *plus one wrongint* pattern matches on *one* to then return the successor of *wrongint*, which is *notanint*. This example of non-commutativity can cause confusion for the programmer because these terms do not have the expected behaviour that we expect from integers.

Third, with the above integer type *wrongint*, unexpected behaviours arise when *wrongint* is used as a term. Consider for instance the following function:

$$\lambda n \leq \mathit{wrongint}.\mathit{match}\ n\ \mathit{one}\ (\lambda m \leq \mathit{wrongint}.\mathit{one})$$

This function matches on the value of n, and then returns *one* both if n is "zero" or if n is a successor. However, applied to *wrongint*, for the same reason as the non-commutativity of the addition, it returns its carrier instead of *one*, which is not the expected behaviour. However, if we consider the following term, which is the same term but requires *int* as a parameter instead of *wrongint*:

$$\lambda n \leq \mathit{int}.\mathit{match}\ n\ \mathit{one}\ (\lambda m \leq \mathit{int}.\mathit{one})$$

Here, the use of *int* instead of *wrongint*, does not have this issue and with which we can prove with our type-checker that the result is always *one*.

$$int \xrightarrow{\equiv}\!\!\!\!\rightarrow \mathrm{Or}(zero, succ\,\mathrm{Or}(zero, succ\,int)) \xrightarrow{\equiv} \mathrm{Or}(zero, \mathrm{Or}(succ\,zero, succ\,succ\,int)) \xrightarrow{\equiv} \mathrm{Or}(zero, \mathrm{Or}(one, succ\,succ\,int))$$
$$\uparrow \leq$$
$$one$$

Figure 4: *one* is a subtype of *int*

For all these reasons, the encoding we use is the one depicted in Figure 3.

Boolean terms are represented similarly as follows:

$$\begin{aligned} true &= \lambda t \leq \mathrm{Top}.\lambda f \leq \mathrm{Top}.t \\ false &= \lambda t \leq \mathrm{Top}.\lambda f \leq \mathrm{Top}.f \end{aligned}$$

The type for booleans is initially defined in a style similar to integers:

$$notyetbool = \mathrm{Y}\lambda f \leq \mathrm{Top}.\lambda t \leq \mathrm{Top}.\lambda f \leq \mathrm{Top}.\mathrm{Or}(t, f)$$

Because the term *notyetbool* is lengthy, in the implementation, we use the following type, which is equivalent, simpler, and more practical:

$$bool = \mathrm{Or}(true, false)$$

The lack of a carrier in booleans allows more expressivity; one can therefore build an if-expression where each branch returns a different type (for instance, one branch returns *int* whereas the other returns *succ*), and as long as the program ultimately type-checks, the program is considered valid.

It is left to future work to define other usual objects such as lists, pairs, etc. We now present the differences between the original PSS theory introduced in [12] and the theory we use.

Our theory enables a limited form of abstract interpretation. For instance, if a function f is a subtype of $\lambda n \leq int.succ\,int$, then for any input x with $x \leq int$, the result $f\,x$ is a subtype of $(\lambda n \leq int.succ\,int)\,x$ which reduces to $succ\,int$. In essence, the type of a term is a superset of the set of possible values of the term.

The subtyping relation in our system provides some information about the term structure. For instance, if we establish that $u \leq succ\,int$, then we can deduce that u is in essence a successor when it comes to the usual pattern matching of integers. Consider for instance a function $f = \lambda n \leq int.match\,n\,zero\,(\lambda n \leq int.one)$ which takes an integer, matches on its value, returns *zero* if the integer is 0, and else returns *one*. Then we know that $f\,u$ is a subtype of $f\,(succ\,int)$, which returns *one*. This reasoning means that there is no possible value for u such that the call of f on u leads to the case "zero", which means that in practice u is a successor. Therefore once the factorial function has been proven be a subtype of $succ\,int$, it is safe to divide by its result.

It is unclear how general this method is to represent set of objects. For instance, we believe it is impossible to represent the set of prime number in such fashion. We can represent the set of non-prime, but as the calculus doesn't have a "difference" operator this doesn't lead to a construction of the set of prime numbers.

3 THE INTERACTIVE TYPE-CHECKER

In this section, we discuss the technical choices related to the implementation, and we discuss how to use the type-checker.

3.1 Why Emacs

In order to write our type-checker, we had to choose a language and a framework. To prove a subtype relation, the programmer selects a subterm of a program and chooses some action on it. The way for the programmer to do so is to select a subterm on which they want to take action, and then select the desired action. This requires exposing to the programmer the exact AST of the program. Emacs and Lisp are technological solutions that adapt very well to this interaction scheme. We now describe the syntax of our calculus in Lisp.

3.2 Syntax of our calculus in Lisp

We represent PSS terms by the following grammar:

```
'x           ; A variable "x"
top          ; The term Top
(FUN x u v)  ; A function with type u and body v
(u v)        ; An application: u applied to v
(Y u)        ; Fixed point combinator applied to u
(Or u v)     ; The term Or(u, v)
```

We use the traditional Lisp conventions. Terms are symbols, written within asterisks, for instance the term *zero* is written `*0*`, and the addition is written `*plus*`. Variables in terms are always quoted, otherwise they are considered as symbols that may be defined elsewhere, for instance the term `(FUN n *int* 'n)` is valid, but the term `(FUN n *int* n)` is not.

We now write with these conventions some of the Scott integers we described in Section 2.2 in Figure 3:

```
(define-term '*0* ; The integer 0
  '(FUN zero-case top
     (FUN succ-case (FUN n *int* top)
       'zero-case)))

(define-term '*1* ; The integer 1
  '(FUN zero-case top
     (FUN succ-case (FUN n *int* top)
       ('succ-case *0*))))

(define-term '*int* ; The type int
  '(y (FUN self top
     (FUN zero-case top
       (FUN succ-case (FUN n 'self top)
         (OR 'zero-case ('succ-case 'self)))))))

(define-term '*succ* ; The successor function
 '(FUN n *int*
    (FUN zero-case top
      (FUN succ-case (FUN n top top)
        ('succ-case 'n)))))
```

3.3 Type-checking programs

To type-check a given program, there are two phases:

(1) Check that the program is well-formed with the rules described in Figure 1 in an automatic process that generates a list of *proof obligations* to resolve.
(2) Address these proof obligations with the help of commands described in Figure 5.

To check that a program is well-formed, we use the rules described in Figure 1. Our algorithm traverses the program to look for all applications of the form $u\,v$. When we encounter such an application, the type-checking process generates two proof obligations with a *placeholder* that stands for the t in the premises of the rule Wf-App. The two proof obligations are to prove that $u \leq \lambda x \leq t.\mathrm{Top}$ and that $v \leq t$.[5] These proof obligations to resolve are displayed in the buffer `*Obligations*`, with the following format, where a is the term to prove as a subtype of the objective b:

```
(<=p a b) ; A proof obligation that a <= b
```

In Emacs, when the programmer addresses a proof obligation, we spawn the following buffers:

- `*Term*` which displays the current status of the term a.
- `*Objective*` which displays the current status of the *objective* b.
- `*Goal*` which displays the list of current *goals*, which are proof obligations, to prove.
- `*Temp-Proof*` which holds the current interactive proof done so far, the programmer's actions are logged to `*Temp-Proof*` until a commit is made to allow exploration and rollbacks in the proof process.
- `*Proof*` which holds the consolidated proof, which thanks to the commit command, `C-c c`, pastes all the content of the buffer `*Temp-Proof*` to the buffer `*Proof*`. At any time the programmer can recompute the current state of the buffers `*Term*`, `*Objective*` and `*Goal*` according to the consolidated proof displayed in the buffer `*Proof*`, with the command `C-c r`.

Then, the programmer needs to pick a subterm of either a or b on which they want to perform an action, and which action they wish to execute. Only equivalence reductions are allowed on the term b, whereas on the term a the programmer can do both equivalence and subtype reductions. The proof ends when both terms are equal. The proof obtained is the path in the form of a diagram as the one in Section 2.1 from the term a to the term b.

3.4 Identification of subterms

An action made by the programmer acts on a subterm of a term. A *path* is a list of integers that describes the position of the subterm on which the action is done, from the main term, which can be used to traverse the AST to find the wanted subterm. At each node of the tree, the child to take is indicated by the integer at the current position in the path, where the first child is at position 0, the second child is at position 1, etc. For instance, take the term $((a\,b)\,c)$, then the path `nil` indicates the whole term, the path `(0)` indicates the term $(a\,b)$, and the path `(0 1)` indicates the term b.

[5] We know that if u is already a function, then the term t to use is its type annotation thanks to the inversion result in [12]. This improvement is implemented.

- **Proof mode commands:**
  - `C-c S`: Compile a term
  - `C-c s`: Start to address a proof obligation
  - `C-c m`: Merge `*Temp-Proof*` in `*Proof*`
  - `C-c r`: Recompute `*Term*`, `*Objective*` and `*Goal*` according to `*Proof*`
  - `C-c q`: Qed command, indicate the proof as ended
- **Action commands (Manipulation of term that modifies the proof, available on both `*Term*` and `*Objective*` unless specified otherwise):**
  - `C-c b`: Equivalence reduction at point
  - `C-c B`: Beta substitution (recursive) at point
  - `C-c e`: Macro expand at point
  - `C-c p`: Promote at point only in `*Objective*`
  - `C-c o`: Split "or" at point only in `*Term*`
  - `C-c C-o n`: Replace the "or" variant by its n choice at point only in `*Objective*`

Figure 5: List of Emacs commands of our type-checker

3.5 Normalisation of terms

The command `C-c B` normalises a term with only β-substitution steps. Indeed, it is important that the `C-c B` command does not make unwanted actions such as duplicating a variant as explained in Section 2.2, or reducing a fixed point combinator which would likely lead to an infinite reduction sequence. For these reasons, the command `C-c B` is limited to only β-reductions.

When the programmer wishes to reduce a fixed point combinator, or to duplicate a variant in a controlled manner, they can use the `C-c b` command, which is a single step of equivalence reduction with no limitation.

3.6 How to deal with variants

Two different actions can be made for variants, depending on whether the variant occurs in the term, or in the objective.

If a programmer wishes to prove that a term $\mathrm{Or}(a, b)$ is a subtype of a term c, the programmer can split the left variant with the command `C-c o`, which spawns two subgoals, one for each element of the variant, that the programmer must prove separately.

If a programmer wants to prove that a term a is a subtype of a term $\mathrm{Or}(b, c)$, then they can choose whether they wish to prove that the term a is a subtype of b or of c, with either the command `C-c C-o 0` to choose b, or `C-c C-o 1` to choose c.

3.7 Recursive programs

In our system, recursive terms are built with our fixed point combinator, that has a well-formedness rule stated in Section 2.2. According to this rule, in order for a recursive term to be well-formed, one should provide a term t as the rule is not syntax-directed. The term t is provided by the programmer when they write a recursive term. For instance, the factorial function is:

```
(define-term '*factorial*
```

```
  '(y (FUN self (FUN n *int* (*succ* *int*))
    ...
```

In the above function, the term t required by the well-formedness rule is $\lambda n \leq int.succ\ int$.

In contrast to other dependent type theories, recursive terms do not have to be terminating. The Syracuse function defined in Section 4.5, for instance, potentially is not terminating, yet is well-formed.

There can be alternative definitions of recursive terms, for instance the factorial function can be defined with the term $t' = \lambda n \leq int.int$ and also be well-formed. However, the term t of the rule WF-Y is the set of values that the function returns: indeed, one of the premises of the rule WF-Y is $\Gamma \vdash Yf \leq t$ which when t is badly chosen (for instance $t = \text{Top}$), provides no useful information about the result of the function. For the factorial case, the motivating example of the paper is proving that its result is greater than one, hence the choice of t being $\lambda n \leq int.succ\ int$ which ensures the result is greater than one.

4 EXAMPLE OF ADDRESSING PROOF OBLIGATIONS

This section presents different examples to showcase how the programmer addresses different proof obligations.

We begin by proving that `*1*` is a subtype of `*int*`. This example serves both as an example of our recursive types, as well as showing how to deal with simple type-checking of integers.

Then we prove that the addition of an integer greater than 1, with any integer, results in an integer greater than 1. This example showcases that reasoning about integers and their sets of values is possible with our system.

Next, we consider the factorial function, and prove it always returns a result greater than 1. This example showcases both reasoning about a recursive function, and some arithmetic reasoning.

Finally, we arrive at our initial example of safe division. This example showcases the generation of proof obligations when a program is type-checked.

Additionally, we prove the Syracuse function always returns 1. This example showcases that our approach works for non-primitive recursive functions.

4.1 Proving that *1* is a subtype of *int*

The term `*1*` is defined as follows:

```
(define-term '*1*
  '(FUN zero-case top
    (FUN succ-case (FUN n *int* top)
      ('succ-case *0*))))
```

And the term `*int*` is defined as follows:

```
(define-term '*int*
  '(y (FUN self top
    (FUN zero-case top
     (FUN succ-case (FUN n *int* top)
       (OR 'zero-case ('succ-case 'self)))))))
```

In order to unify both terms, we first expand the term `*0*` in the term `*1*`. We obtain the following term:

```
'(FUN zero-case top
  (FUN succ-case (FUN n *int* top)
    ('succ-case
      (FUN zero-case top
        (FUN succ-case (FUN n *int* top)
          'zero-case)))))
```

Now we reduce the term `*int*` to obtain:

```
(FUN zero-case top
  (FUN succ-case (FUN n *int* top)
    (OR 'zero-case
      ('succ-case
        (y (FUN self top (FUN zero-case top
          (FUN succ-case (FUN n *int* top)
            (OR 'zero-case ('succ-case 'self))))))
              ))))
```

We can now pick the second choice in the first Or disjunction, which selects the successor choice, and then pick the first choice in the second Or disjunction, which selects the "zero" choice. All in all, the result should be the successor of "zero", which is the term `*1*`. Once we select these choices and β-reduce, we obtain:

```
(FUN zero-case top
  (FUN succ-case (FUN n top top)
    ('succ-case
      (FUN zero-case top
        (FUN succ-case (FUN n top top)
          'zero-case)))))
```

The term above is exactly the reduced term `*1*`, which concludes the proof.

The proof trace, where β-reduction and macro expansion are removed for readability, is the following:

```
(start-term *1*)
(start-goal *int*)
;; reduce fixed point combinator in *int*
(rewrite beta-substitute :on-objectives t
  :path nil)
;; choice: the successor
(rewrite (objective-or-split 1) :on-objectives t
  :path (3 3))
;; choice: zero
(rewrite (objective-or-split 0) :on-objectives t
  :path (3 3 1 1 3 3 3))
;; reflexivity
(qed)
```

4.2 The addition of a successor and anything, is a successor

In this section, we address the following proof obligation: `(<=p ((*plus* (*succ* *int*))*int*)(*succ* *int*))`.

Our starting term is:

```
((*plus* (*succ* *int*)) *int*)
```

We first discharge the case where n is "zero". To do so, we expand the term `*succ*` and execute a step of the function `*plus*`. We now obtain:

```
(*succ*
  (((y (FUN self (FUN a *int* (FUN b *int* *int*))
    (FUN m *int*
      (FUN n *int*
        (('m 'n) (FUN a *int* (*succ* (('self 'a)
            'n)))))))))
    *int*)
  *int*))
```

The above term is the expected result; the function `*plus*` was applied to a successor, so according to its definition the result is the successor applied to itself with some arguments. In the term above, the recursive call is `(('self 'a)'n)`, and we wish here to only prove that the result is a successor. We can therefore promote the induction call to only an integer, and still obtain the correct result.

To do so, we promote the variable `'self`, and we can then apply the β-reduction to obtain:

```
(*succ*
  ((*int*
    *int*)
    (FUN a *int* (*succ* *int*))))
```

The above term is in fact a pattern matching on the value of `*int*`. If the value of `*int*` is `*0*`, then the result is `*int*`, else the result is also `(*succ* *int*)`. Once we expand the term `*int*`, we see more clearly the two options, as we obtain after some β-reduction:

```
(*succ*
  (OR *int* (*succ* *int*)))
```

We now split the two choices, thanks to Rule Leq-OrLeft, and conclude each case easily.

4.3 Factorial always returns a result greater than one

In this section, we prove that the factorial function always returns a result greater than 1.

The factorial function is defined as follows:

```
(define-term '*factorial*
  (y (FUN self (FUN n *int* (*succ* *int*))
    (FUN n *int*
      (('n
        ;; n == 0 -> 1
        *1*)
        ;; n == S(a) -> n * (fact a)
        (FUN a *int*
          ((*mult* 'n) ('self 'a)))))))))
```

In this section, we prove that `*factorial*` is a subtype of `(*succ* *int*)`. To prove the required subtyping derivation, we split the value of `*int*` into either `*0*`, or a successor, thanks to Rule Leq-OrLeft. For the case `*0*`, once we expand and β-reduce, we obtain directly the term `*1*`, which we can prove as in Section 4.1 is a subtype of `(*succ* *int*)`. Otherwise, we obtain the following term:

```
((y (FUN self (FUN n *int* (*succ* *int*))
  (FUN n *int*
    (('n
      *1*)
      (FUN a *int* ((*mult* 'n) ('self 'a)))))))
  (...))
```

We have omitted the term with ... because the term is long; the term is beta-equivalent to `(*succ* *int*)`. The term obtained above is the application of the factorial function to the term `(*succ* *int*)`. To conclude, we first promote the recursive call of the factorial, in order to mimic the induction principle needed to prove the wanted result. We therefore promote the variable `'self` to its type annotation `FUN n *int* (*succ* *int*)`, after which the term `('self 'a)` reduces to `(*succ* *int*)`. After these changes, and some β-reduction, we obtain the following term:

```
((*mult*
  (...))
  (*succ* *int*))
```

Here, the term ... designates the same term as above. The proof trace, up to this point is the following:

```
(start-term
  (*factorial* *int*))
(start-goal
  (*succ* *int*))

;; split *int*
(rewrite split-or-all :on-objectives nil
  :path (1 3 3))
;; case *int* = 0
(focus 0)
;; choice "zero" in the objective
(rewrite (objective-or-split 0) :on-objectives t
  :path (1 1 3 3 3))
;; reflexivity
(qed)

;; case *int* = succ(...)
(focus 0)
;; "by induction" - promote self to its objective
(rewrite promote :on-objectives nil
  :path (0 1 3 3 1 3 1 0))
```

To then conclude this proof, one should resolve the obligation `(<=p ((*mult* (*succ* *int*))(*succ* *int*))(*succ* *int*))` as in Section 4.2.

4.4 Type-checking the safe division

In this section, we type-check the safe division program described at the start of the paper. The program we wish to type-check is the following:

```
'(FUN p *int*
  (FUN n *int*
    ((*safe-div* 'p) (*factorial* 'n))))
```

When we type-check this program, a list of obligations is generated. Most of them are immediate to prove, except for `(<=p *factorial* (FUN n *int* (*succ* *int*)))`, which we have already addressed in Section 4.3.

Once the above non-trivial proof obligation has been addressed, and once all the other proof obligations are addressed, the program is type-checked and can be executed without any error.

In the next section, we highlight that our approach works with any recursive function, including potentially non-terminating ones, by proving that the Syracuse function returns 1.

4.5 The Syracuse function always returns 1

One important example that highlights the difference between our type-checker and traditional proof assistants is the Syracuse function defined in the introduction. It is an open problem to know whether the Syracuse function terminates, yet it is straightforward to see that if the Syracuse function terminates then the result must be 1. We define the Syracuse function as follows:

```
(define-term '*syracuse*
  (y (FUN self (FUN n *int* *1*)
    (FUN n *int*
      (((*isBelowOne* 'n)
        *1*)
        (((*isEven* 'n)
          ('self ((*div* 'n) *2*)))
          ('self (*succ* ((*mult* *3*) 'n)))))))))
              )
```

In our implementation, the functions `*isBelowOne*`, `*isEven*` and `*div*` are not yet implemented, which does not impact this example. To type-check the Syracuse function, we need to address the following proof obligation (`<=p *syracuse* (FUN n *int* *1*)`).

To do so, we promote each recursive instance of `'self` to the objective, both conditions (`*isBelowOne* 'n`) and (`*isEven* 'n`) to a boolean (which in this case depends on their implementation), to obtain the following term:

```
((((*bool* *1*) *bool*) *1*) *1*)
```

We can now split both booleans, thanks to Rule Leq-OrLeft, and conclude each case. The proof trace, up until proving that each term `*1*` is indeed `*1*` in each boolean case is the following:

```
(start-term
  (*syracuse* *int*))
(start-goal
  *1*)

;; Promote the first recursive call
(rewrite promote :on-objectives nil
  :path (0 1 3 3 1 1 0))
;; Promote the second recursive call
(rewrite promote :on-objectives nil
  :path (0 1 3 3 1 0 1 0))
;; Split the first boolean
(rewrite split-or-all :on-objectives nil
  :path (0 0))
;; 1st boolean case *true*
(focus 0)

;; Split the second boolean
(rewrite split-or-all :on-objectives nil
  :path (1 0 0))

;; Then, the rest of the proof
```

This example highlights the difference between our system and traditional dependent-type systems that have strong normalisation requirements.

5 CONCLUSION

In this section, we present the related work, the future work, and a conclusion.

5.1 Related work

Pure type systems [15] allow a finer control over the kinds used for terms and types, which generalise the lambda calculi with types in the Barendregt Cube [3]. These systems, however, use a typing relation instead of a subtyping one.

Other attempts try to unify typing and subtyping, like $\lambda I_{\leq}$ [22], a dependently typed generalisation of System $F_{\leq}$. In a different approach, Aspinall studied the combination of subtyping with singleton types [1].

Some systems, like Cedille, use pure lambda calculus to encode types, similar to our approach, instead of using primitive operators for algebraic data types like in other theories [19]. Cedille combines dependent types with a restricted notion of recursion. Our system, however, blurs completely the distinction between terms and types.

Proof assistants like Rocq [7], Agda [5] leverage dependent types to prove properties about programs or mathematical properties in general. In these systems, the programmer specifies a type that captures the desired specification, and then proceeds to construct a term, either via tactics for Rocq, or directly inputting the term for Agda. All of these approaches require strong normalisation, which is not required in our theory.

These proof assistants can be used in Emacs with Proof General [2], which is also written in Emacs Lisp, and lets the programmer interact with a variety of proof assistants in Emacs. The interface exposed to the programmer is similar: multiple buffers are spawned with the relevant information, and are updated in sync when the programmer interacts with them.

Interactive typecheckers for undecidable type theories were also considered in the past, for instance the NuPRL proof development system [13] which is a proof assistant based on the undecidable Martin-Löf intuitionistic type theory [14].

Proof assistants that allow, as our approach, reasoning about non-terminating terms have been studied, with for instance LCF [11, 17], with the drawbacks of the complexity of the reasoning.

Other programming languages with dependent types exist, such as Idris [6] and F* [20, 21], we believe our approach is more versatile as it does not have any strong normalisation requirement on either terms or types.

5.2 Limitations and future work

In order to increase the expressivity of our type-checker, multiple features that are already implemented in the traditional proof assistants could be added to our type-checker. We enumerate some of them.

We are unable to deal with refinement types [9]. One simple example with integers is the set of primes. While we can build the

set of non-primes, which is in our calculus `((FUN x *int* ((*mult* 'x)'x))*int*)`, it is unclear in our system how to build the set of primes.

For the type *int*, the only case disjunction we are allowed to do is to be either 0 or a successor. An area of future work is to allow the programmer to do more advanced case disjunctions, for instance to define a case disjunction on the evenness of an integer.

An area of future work is bringing more automation to our type-checker. Proof assistants usually have a lot of automation features that are very handy for programmers. Bringing automation to our type-checker is an area of future work.

Even if there is no Curry-Howard correspondence in PSS, some induction principles could be extracted from recursive functions. The recursive function extracted would take an additional *fuel* parameter which decreases at each recursive call, and the extracted induction principle is the strict decrease of this fuel parameter at each recursive call.

5.3 To conclude

In this system description, we described our experimental type-checker based on the PSS theory. We believe the PSS theory and our type-checker have some interesting properties. Our system is more versatile than other type systems, as it allows the programmer to write and reason about non-terminating terms.

REFERENCES

[1] David Aspinall. 1994. Subtyping with singleton types. In *International Workshop on Computer Science Logic*. Springer, 1–15.

[2] David Aspinall. 2000. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25–April 2, 2000 Proceedings 6*. Springer, 38–43.

[3] Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. 1992. Lambda Calculi with Types. (1992), 117–309.

[4] Richard Bird and Oege De Moor. 1996. The algebra of programming. *NATO ASI DPD* 152 (1996), 167–203.

[5] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*. Springer, 73–78.

[6] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming* 23, 5 (2013), 552–593.

[7] Thierry Coquand and Gérard Huet. 1986. *The calculus of constructions*. Ph. D. Dissertation. INRIA.

[8] Haskell B Curry, J Roger Hindley, and Jonathan P Seldin. 1972. *Combinatory Logic. Volume II*. North-Holland Publishing Company.

[9] Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 268–277.

[10] Herman Geuvers and Joep Verkoelen. 2009. On fixed point and looping combinators in type theory.

[11] Michael J Gordon, Arthur J Milner, and Christopher P Wadsworth. 1979. *Edinburgh LCF: a mechanised logic of computation*. Springer.

[12] DeLesley Hutchins. 2009. *Pure subtype systems: A type theory for extensible software*. Ph. D. Dissertation. The University of Edinburgh.

[13] Christoph Kreitz. 2002. The Nuprl Proof Development System, Version 5: Reference Manual and User's Guide. *Department of Computer Science, Cornell University* (2002), 10.

[14] Per Martin-Löf. 1984. Intuitionistic type theory vol. 1. (1984).

[15] James McKinna and Robert Pollack. 1993. Pure type systems formalized. In *Typed Lambda Calculi and Applications: International Conference on Typed Lambda Calculi and Applications TLCA'93 March, 16–18, 1993, Utrech, The Netherlands Proceedings 1*. Springer, 289–305.

[16] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

[17] Gordon D. Plotkin. 1977. LCF considered as a programming language. *Theoretical computer science* 5, 3 (1977), 223–255.

[18] Dana Scott. 1962. A system of functional abstraction (1968). Lectures delivered at University of California, Berkeley. *Cal* 63 (1962), 1095.

[19] Aaron Stump. 2017. The calculus of dependent lambda eliminations. *Journal of Functional Programming* 27 (2017), e14.

[20] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. *ACM SIGPLAN Notices* 46, 9 (2011), 266–278.

[21] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the Dijkstra monad. *ACM SIGPLAN Notices* 48, 6 (2013), 387–398.

[22] Yanpeng Yang and Bruno C d S Oliveira. 2017. Unifying typing and subtyping. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

Abstract Machines and Small-step Semantics: a Winning Ticket for Proof Automation?

Alain Delaët
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
alain.delaet@inria.fr

Sandrine Blazy
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
sandrine.blazy@irisa.fr

Denis Merigoux
Inria
Paris, France
denis.merigoux@inria.fr

Abstract

Verifying the correctness of a program using an interactive proof assistant involves first defining in the proof assistant the semantics of the involved programming language. Once mechanized, the semantics serves as the basis for the whole proof development, as it is referred to by all subsequent theorems. Traditionally, semantic judgments are mechanized using recursive inductive predicates, matching the pen-and-paper inference rules of operational semantics. However, the shape of these judgments may make proof engineering and maintenance tedious, requiring complex proof automation frameworks. This problem is especially acute when iterating during the language design period.

In this paper, we highlight a different style of writing and mechanizing semantics that follows the CEK abstract machine. We explain how exactly this style interacts better with basic proof automation tactics than the traditional structural operational semantics (SOS) style for the λ-calculus. As such, we explicitly detail why this alternative semantic style allows for scaling up more efficiently the mechanized proof development, and hint at future improvements based on the same principles. After detailing the inner workings of the interactions between the semantic style and proof automation of common theorems, we perform a case study for the medium-sized Catala domain-specific programming language. All of our examples and case studies are mechanized using the Rocq proof assistant and our development is available as a supplementary material [21].

CCS Concepts

• **Software and its engineering → Software verification**; **Semantics**; **Context specific languages**.

Keywords

verified compilation, operational semantics of programming languages

ACM Reference Format:
Alain Delaët, Sandrine Blazy, and Denis Merigoux. 2025. Abstract Machines and Small-step Semantics: a Winning Ticket for Proof Automation?. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

PPDP '25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Proof assistants have come a long way in the last twenty years, to the point where software as complex as realistic compilers has been proven correct [2, 3, 9, 13, 31]. A verified piece of software consists of the software itself, which runs outside of the proof tool (as any other software), together with a machine-checked proof that the software is correct with respect to its specification. When the software implements a program transformation (*e.g.*, a compilation pass), its specification expresses its semantic preservation (and possibly other semantic properties), thus involving the formal semantics of the underlying programming languages.

Semantic reasoning on realistic languages generates large proofs and requires a proof tool. Automatic proof tools discharge logical formulas to logic solvers (that may fail to prove the formulas), but they are not very well adapted to proving the correctness of sophisticated program transformations operating over realistic languages. Indeed, to be able to reason about the semantics, it is necessary to somehow deeply embed it inside the proof tool. Hence, proofs about semantics require tricky reasoning. For this reason, proof assistants (a.k.a. interactive proof tools) are rather used in these situations where user interaction is still needed at each proof step to decide on what to reason about. The price to pay is that proof assistants require writing these elementary proof steps (called *tactics*), which makes the proofs difficult to maintain over time as new features are added or statements are reformulated. What is especially tricky is defining and proving the semantic invariants that hold during the proof of a given theorem, as this may necessitate inventing whole new "phantom" data structures for the sole purpose of the proof.

Nearly all theorems in a mechanized development for verified software will refer back to the semantics of the language, whether one is building a verified compiler [2, 3, 9, 13, 31], a verified abstract interpreter [27], or a verified bug-finder [16]. As such, the encoding of the language semantics inside a proof assistant is a critical and central software component of the mechanized development. From a software engineering point of view, the design patterns and coding style of the language semantics as a program in the proof assistant heavily influence the rest of the mechanized development, as it is a direct dependency to everything else.

Compared to other semantic styles (mainly denotational and axiomatic), operational semantics defines precisely the low-level execution of a program and is a de facto standard for mechanizing semantic reasoning in a proof assistant [28, 44, 46]. For instance, the correctness theorem of a program transformation (from a source to a target program) states that any execution of one of the programs

is *simulated* by the execution of the other one, while ensuring some invariants between execution states (which can be tricky to define and mechanize). In compiler verification, because of optimizations, bisimulations do not hold, and forward or backward simulations are rather used to simulate one execution step by zero, one or several execution steps, given an *invariant* between semantic states.

Such subtle proofs require a great deal of precision from the operational semantics w.r.t. details like execution order. While certain styles of operational semantics such as big-step are easier to manipulate inside a proof assistant, they may not always contain all the information necessary for proofs involving complex features (*e.g.*, concurrency, weak memory models), forcing the proof engineer to default to a more tedious small-step operational semantics.

The proof may require iterating between proof development and theorem statement refactoring, in order to define new semantic invariants and intermediate lemmas. Moreover, it may be useful to debug the semantics, which requires replaying the proof many times over. There are other reasons why proofs need to be replayed (and updated): firstly, programming languages evolve to satisfy users' needs, and secondly, new intermediate representations may be added to refactor the proofs (and split them in smaller and more maintainable pieces).

Furthermore, automating these proofs is not a simple task, as the sequence of elementary proof steps is not straightforward; it depends on the property to prove as well as a precise combination of inductive and case-based reasonings. If the individual proof steps are well understood, as they correspond to specific tactics (mainly induction or case-based reasoning, inversions, or formula simplification), there are a number of ways of combining them. This raises the question of what we can do to improve proof engineering and especially proof automation in mechanized developments involving language semantics, which is the subject of this paper.

The main thesis of this paper is that *using abstract machines defined syntactically to model small-step operational semantics allows for a simple but powerful proof automation, leveraging the unification capabilities of the proof assistant.* If this result has already been hinted at multiple times in the literature and is part of the folklore amongst the connoisseurs, we explain in depth and going back to the basic mechanisms of proof automation *why* abstract machines work better in mechanized developments. We believe this extensive description of the phenomenon might guide future improvements in proof automation leveraging clever encoding of semantics, and bring more scientific method into the quest for scaling up mechanized proofs.

1.1 Related Work

Before detailing our contributions, it is necessary to give some context about abstract machines, semantics, and encoding choices inside proof assistants in major mechanized developments over time.

Abstract machines for describing operational semantics have been the subject of extensive study since the inception of the λ-calculus and subsequent languages. Starting with Landin's SECD machine [32], many theoretical abstract machines have been proposed, including Felleisen and Friedman's CEK machine [23] and Krivine's machine [30] that model the different reduction strategies. These machines then influenced the implementation of modern functional programming languages, for instance through the CAM [18] and later the ZAM [34] from which the initial implementation of Caml was derived.

While being a popular foundation for language implementation, we notice that abstract machines have not become the default style for encoding language semantics inside proof assistants. Notably, the famous Software Foundations manual [44] overwhelmingly used to teach Rocq and language theory describes the small-step operational semantics of the λ-calculus using a recursive judgment and substitution. Indeed, this style of semantics encoding, which we will call structural operational semantics throughout this paper, is the closest to the pen-and-paper presentation of small-step operational semantics, making it the best teaching vehicle. We suppose that, because of its absence in standard teaching material, abstract machine semantics have been neglected in mechanized proof developments presented in the literature. To substantiate this claim, we will survey the semantic encoding choices of major works striving to scale up mechanized program verification.

CakeML [31] is a verified compiler for a substantial subset of Standard ML, large enough to bootstrap itself. The semantics of the CakeML languages use mainly functional big-step style [42] (*i.e.*, functions are used instead of relations to define the semantic rules), but interestingly an abstract machine (small step) semantics in the style of CEK was used in earlier versions of the compiler "for the type soundness proof" [42, p. 2]. In general, the use of big-step semantics avoids the bookkeeping of structural operational semantics and scales up better, but they are not adapted to define in a natural way some semantic features such as unstructured control, diverging and concurrent executions. More generally, "[a]s a rule-of-thumb, the more complex the language's features, or the more semantically intricate the desired theorem, the more likely it is that small-step semantics will be needed" [42, p. 1].

Another classic twist on the structural operational semantics mechanization is the factorization of the recursive part of the judgment inside a context, with an unique contextual rule. This twist has been used for a long time [12], and is reused inside a recent survey of mechanization inside Rocq of reduction strategies for structural operational semantics [7]. This twist, as we will explain later in the paper, does not avoid very manual proofs with inversion lemmas and explicit lemma applications, following the style popularized in [44]. It is not an issue as [7] implements a simple pure λ-calculus, but the syntactic proof of type safety [57] is quadratic in the number of constructors and scales poorly. This pitfall is pointed out by [50] which proposes a fix, extended to non-deterministic languages by [15], but the fix requires a clever re-stating of the type safety theorem, which might be difficult to port to the proof of other properties.

Another formally verified compiler for a functional language is CertiCoq [3], a compiler for the specification language of Rocq. Its intermediate languages use big-step semantics and avoid abstract machines. To end our tour of verified compilers, we can also consider Vélus [14], which proves the correctness of a large subset of Lustre using CompCert. Vélus uses a mix of inductive and coinductive objects to express relations between streams of values. Its main correctness proofs work by performing induction on an

object, inversion on other inductive hypotheses, and reconstructing the proof in each case. Moreover, [14, Section 3.2] highlights the difficulty of discovering the details required to mechanize a proof. These details are typically found while writing the proof and often require modifications to the theorem's hypotheses and semantic definitions.

Outside of verified compilers, domain-specific mechanized developments using language semantics also report difficulties for scaling up their proofs. Benzaken et al. [6] prove the correctness of a SQL to IMP scheduler by defining their semantics using big-step semantics. Similar to Vélus, their proof is not automated and requires keeping the language small to remain tractable. For example, they avoid using `if` in one of their intermediate languages to keep the number of inductive variants low, expressing it through other constructs, though at the cost of increasing the length of the generated code. Another key application of mechanized proofs for software verification is cryptography. The lines of work around Fiat [20, 22] or the recent Noise* [26] rely on synthesis and meta-programming inside the proof assistant to generate verified code from a high-level specification. These techniques, like other recent works [25], allow for a very high degree of proof automation, at the expense of a large proof infrastructure that may be tedious to maintain and expand.

Finally, we can also look at specification efforts for semantics of large, real-world languages. Extensive work has been recently put into formalizing WebAssembly, in Isabelle [55] and in Rocq [47]. Both mechanizations use small-step semantics, but, to Watt's admission [56], "[B]ecause WebAssembly was deliberately designed to be small, simple, and amenable to formalisation, mechanizing the full semantics [...] is more easily achievable". On the JavaScript side, JSCert [10] uses big-step semantics. The K framework [52] has been extensively used to model several high-profile real-world programming languages, but the framework is not a proof assistant and does not allow for the mechanized proofs of meta-properties involving the semantics. Overall, the related work presented above tends to flee from mechanizing large small-step semantics to avoid manual proofs that scale and repair poorly.

Nonetheless, it is sometimes impossible to avoid small-step semantics because the proof relies on the execution order or details missing from big-step semantics. Then, mechanized developments tend to flock back to some kind of abstract machines to support the scaling up, although some fail to mention their use of abstract machines properly. For instance, Appel and Blazy [4] present a small-step semantics using an abstract machine (which they name "*continuations*") for the statements of the `Cminor` language of CompCert. They then define a sound separation logic for their imperative language. They report that the use of this abstract machine "allow an uniform representation of statement execution", without elaborating. It is worth noting that these "continuations" are first-order syntactic data structures representing the control stack of an abstract machine, distinct from the higher-order continuations of continuation-passing style in denotational semantics. More recently, syntactically encoded abstract machines were incorporated in the Skeletal semantics framework [11]. They provide general tooling for defining and manipulating operational semantics. Skeletal semantics were used on toy languages to derive interpreters or static analyzers automatically from operational semantics. To improve automation in skeletal semantics, Khayam and Schmitt [29] define a purely functional meta-language that captures target language features and introduces control stacks to derive abstract machine semantics.

Additionally, Courant et al. [17] use abstract machine semantics for the paper proof of the correctness of their code generator, and claim that they could be "easily mechanized". Going beyond semantics, using syntactic artefacts to efficiently encode judgments in a proof assistant seems to have a broader application scope. Li and Appel [35] encode program transformations in a similar fashion, using MetaCoq [54] to enable proof sharing between different transformations. However, the correctness proof for their transformations still requires significant work: they focus more on extracting efficient implementations of transformations rather than easing their mechanized proof development.

These hints from the literature pointed us to recognize that behind these semantics using "continuations" (i.e., control stacks), there is a revamp of abstract machines for the λ-calculus, inside proof assistants. Their syntactic features seem to help mechanized proof scale up, and lower the barrier to using small-step semantics in large developments. *Why? Can we generalize this insight and leverage it to obtain better proof automation for more theorems involving semantics?* We will answer these questions in the paper by showing the fine-grained interactions between semantic encodings and proof tactics, using first a call-by-value λ-calculus with both a structural operational semantics and a abstract machine semantics following the CEK machine.

This subject arose for us during the development of a new verified compiler for the Catala domain-specific programming language [39]. Catala is a language designed for situations where a given set of laws or regulations has to be turned into computer code for automatic enforcement. In practice, this situation concerns mostly tax or social benefits computations. These strong requirements make these sometimes-thousands-of-lines-of-code-long programs [40] hard to write and maintain, as the source of truth for their behavior is determined by legal interpretation performed by government lawyers. Catala has a set of syntactic and semantic features that facilitates the review of the program source code by lawyers. First, Catala allows for literate programming, locally linking each snippet of computer code to the article of statutes or regulations that specifies it. Second, Catala's semantics is based on default logic [33], a logic that matches how the law lays out its instructions for how to compute things. Because tax- and social-benefits-computing programs are critical for the smooth operation of state apparatus, a high level of assurance is also required of the Catala compiler that translates the source Catala code to target programming languages such as C or Python.

1.2 Contributions

We defined a new intermediate representation λ^δ for Catala, improving upon the original default calculus from [39], and mechanically verified an alternative compilation scheme using an option monad instead of try/catch exceptions. These efforts will thus be used as an empirical evaluation of the proof engineering and automation techniques discussed in this paper. While Catala is not as complex as real-world general-purpose languages like C [36], it is based on

a simply typed λ-calculus with classical extensions, enriched with a default calculus involving default terms written as $\langle t^* \mid t \ : - \ t \rangle$. For instance, the simple default term $\langle t_{ex1}, t_{ex2} \mid t_{just} \ : - \ t_{cons} \rangle$ has two exceptions t_{ex1} and t_{ex2}, that defaults to a base case consequence t_{cons} when the base case justification t_{just} is true. This logical structure posits that exceptional cases should apply first, and only by default the base case should apply. It fits very-well to define law and regulations. Moreover, default terms have been evaluated empirically as powerful enough to accurately formalize the entirety of the real-world French housing benefits [38]. The use of small-step for the Catala semantics helps to pin down precisely the execution order and error semantics of the default term. While it may not be completely necessary as the semantics could be simple enough to be converted into big-step, the small-step semantics have been useful during the design phase, when iterating over the semantics of the default term. While later works like [24, Section 4.1] and [5] have proposed optimizations or simplifications for the semantics, we will stick in this paper to the original small-step style of the Catala semantics that also matches the compiler implementation.

Adding new features (incl. the default calculus) to Catala resulted in the λ^δ language, which exhibits the key characteristics where abstract machine semantics shine for proof automation: syntactic ambiguity arising when multiple reduction rules can apply to a single language construct. Unless noted otherwise, all results presented in this paper have been mechanically verified using the Rocq proof assistant. The complete development is available as supplementary material. Specifically, we claim the following novel contributions.

- **Semantics styles:** We mechanize the proof of equivalence between a structural operational semantics and a abstract machine semantics equivalent to a CEK machine for the call-by-value λ-calculus (prior mechanizations of the CEK machine exist [51], but we have not yet found a mechanization of this equivalence proof with the structural operational semantics).
- **Semantics and tactics:** We explain the different interactions between basic proof automation tactics and semantic judgments, depending on the style of semantics.
- **Proof engineering:** We rely heavily on state-of-the-art proof automation techniques and present powerful but low-tech meta-interpreters based on a couple of primitive tactics, leveraging the proof assistant's syntactic unification.
- **Application:** We illustrate how our findings scale on the proof of a novel compilation pass for the λ^δ domain-specific programming language of Catala.

This paper is organized as follows. First, Section 2 presents two small-step semantics for a call-by-value λ-calculus. Then, Section 3 details how both styles interact with the basic blocks of proof automation in a proof assistant. Finally, Section 4 illustrates how abstract machine semantics allows for better scaling up of proofs in mechanized developments. It introduces λ^δ, a new mechanized semantics for Catala's core intermediate representation, the default calculus, as well as the proofs of various theorems around λ^δ, including the correctness of a non-trivial compilation pass.

2 Mechanizing Small-step Semantics: Choose Your Style

There are many ways to mechanize semantics inside a proof assistant. Coupled with the criticality of the semantics as a central proof component, the choice of semantic styles makes for a hard decision for the proof engineer, as this choice will determine the shape and details of all future proofs, as well as how easy they will be to write, debug and repair. Admittedly, the syntax of terms quite naturally maps to an inductive type inside the proof assistant, without much room for user fantasy. But the semantic judgments can be mechanized in different ways. Indeed, one can choose to mechanize typing and reduction as total or partial functions over terms, thus building an executable typer or interpreter [42]. But this encoding differs quite significantly from semantics written on paper, with standard inference rules.

As the semantics encoding is part of the trusted computing base of the mechanized development, it is better for human review [37, Section 1.2.1] if the encoding matches closely the pen-and-paper rules. Also, since functions have to be proven total, encoding semantics as functions quickly becomes cumbersome. Hence, the recommended way [41, 44] to mechanize semantics is to define them as inductive types, where each type constructor corresponds to an inference rule, which ensures that the inductive reasoning and the case analysis closely follow the pen-and-paper ones.

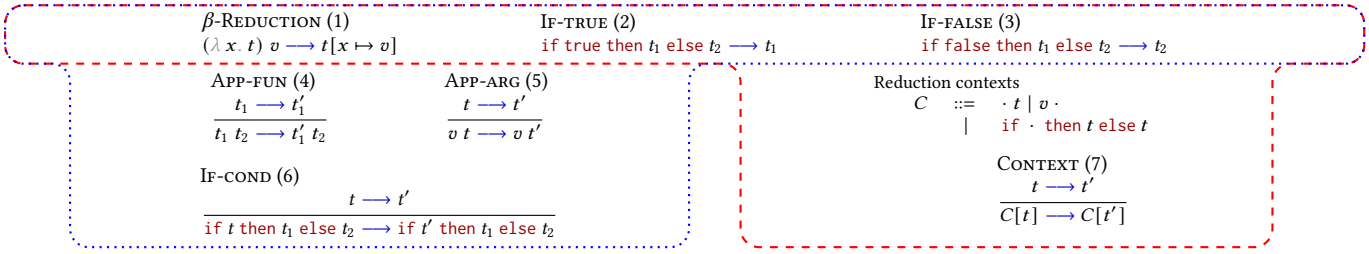
But even when encoding the semantic judgment inductively, the user is still left with some degrees of freedom, like the choice between small-step or big-step semantics. They are equivalent but they will be expressed with very different rules, and mechanized differently. As stated in the introduction, this paper will focus only on small-step semantics. But additionally, there are several ways of defining small-step semantics, depending on the way semantic states are defined. This is precisely the point of this paper, and we will show two such small-step semantics: first, *structural operational semantics*, using recursive premises in the reduction judgment, and second a *abstract machine semantics*, using syntactically-defined abstract machines to encode the control stack that describe the context in which we step.

This section presents both styles: the first one feels familiar as it represents the most taught and presented style. On the contrary, the second one can feel more artificial and verbose, while very precise and exhaustive. Why bother with the second style? We answer in Section 3, where we dive deep into the issues of proof automation and proof repair, and compare the styles. This section presents the two styles for a simply-typed λ-calculus augmented with conditionals that simplifies the more realistic language λ^δ we detail later in Section 4. A value is a Boolean value or a function abstraction. A term is a variable x, a value v, a function application $t\ t$ or a conditional. Its (standard call-by-value) semantics is defined by the judgments $t \longrightarrow t'$ for traditional small-step semantics, and $s \rightsquigarrow s'$ for abstract machine semantics (see Figure 1).

Values	v	::=	$\texttt{true} \mid \texttt{false} \mid \lambda\, x.\, t$
Terms	s	::=	$x \mid v \mid t\ t \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t$

2.1 Traditional Small-step Semantics

There are two kinds of rules defined in Figure 1a: contextual rules (4), (5) and (6) have recursive premises, and computation rules (1), (3)



(a) Structural operational semantics **(b) Context-based small-step semantics**

$$\mathcal{S}(x,\kappa,\sigma) \rightsquigarrow \mathcal{C}(v,\kappa) \quad \text{when } x \mapsto v \in \sigma \qquad \text{Compute-Var}$$
$$\mathcal{S}(\texttt{true},\kappa,\sigma) \rightsquigarrow \mathcal{C}(\texttt{true},\kappa) \qquad \text{Compute-Val-True}$$
$$\mathcal{S}(\texttt{false},\kappa,\sigma) \rightsquigarrow \mathcal{C}(\texttt{false},\kappa) \qquad \text{Compute-Val-False}$$
$$\mathcal{S}(\lambda\ x.\ t,\kappa,\sigma) \rightsquigarrow \mathcal{C}([x,t,\sigma],\kappa) \qquad \text{Compute-Abs}$$
$$\mathcal{S}(t_1\ t_2,\kappa,\sigma) \rightsquigarrow \mathcal{S}(t_1,\mathsf{app}(t_2,\sigma)::\kappa,\sigma) \qquad \text{Focus-App}$$
$$\mathcal{S}(\texttt{if } t \texttt{ then } t_a \texttt{ else } t_b,\kappa,\sigma) \rightsquigarrow \mathcal{S}(t,\mathtt{if}(t_a,t_b,\sigma)::\kappa,\sigma) \qquad \text{Focus-If}$$
$$\mathcal{C}([x,t_{\text{cl}},\sigma_{\text{cl}}],\mathsf{app}(t,\sigma)::\kappa) \rightsquigarrow \mathcal{S}(t,\mathsf{fun}([x,t_{\text{cl}},\sigma_{\text{cl}}])::\kappa,\sigma) \qquad \text{Resume-App}$$
$$\mathcal{C}(v,\mathsf{fun}([x,t_{\text{cl}},\sigma_{\text{cl}}])::\kappa) \rightsquigarrow \mathcal{S}(t_{\text{cl}},\kappa,x\mapsto v::\sigma_{\text{cl}}) \qquad \text{Resume-Beta}$$
$$\mathcal{C}(\texttt{true},\mathtt{if}(t_a,t_b,\sigma)::\kappa) \rightsquigarrow \mathcal{S}(t_a,\kappa,\sigma) \qquad \text{Compute-If-true}$$
$$\mathcal{C}(\texttt{false},\mathtt{if}(t_a,t_b,\sigma)::\kappa) \rightsquigarrow \mathcal{S}(t_b,\kappa,\sigma) \qquad \text{Comp-If-false}$$

(c) Abstract machine semantics

Figure 1: Three small-step semantics for a simply-typed λ-calculus with conditionals

and (2). Rule (1) defines the semantics of function application ; the substitution mechanism here is assumed to be capture-avoiding; in practice our mechanized development uses de Bruijn variables and `autosubst` [53]. While computation rules actually encapsulate the core of the semantics, contextual rules encode the recursive nature of the evaluation and its order. In pen-and-paper presentations, it is common to define more compact semantics using reduction contexts to factor out the contextual rules as shown in Figure 1b. However, the actual effects of the context are more difficult to parse. In mechanized semantics, the context C is an inductive type defining different contexts including a hole. Then, a function, named the context application function, is defined to take a term and a context, and return the new term corresponding to the context where the hole has been replaced with the argument term. The context reduction rule uses this function to state that if a term reduces, then any context applied to this term also reduces.

Both semantics (with or without explicit context) share key characteristics. First, their judgment is recursive. Second, this recursiveness concretely adds a (recursive) premise to rules (4), (5) and (6). In the contextual version of the structural operational semantics, the recursive premise only appears in rule (7). These recursive premises heavily impact the manipulation of the rules during a meta-theoretical proof. Indeed, suppose we want to know which rules apply to a term t to determine t' such that $t \longrightarrow t'$, which is a very frequent proof-pattern. We case analyze each rule to determine whether it can apply or not. The shape of t allows us to discard many rules. However, suppose t is $\texttt{if true then } t_2 \texttt{ else } t_3$. Then, we can narrow down eligible rules to If-true, but also the condition-related contextual rule (If-cond or Context), which might apply because we cannot rule out immediately that the premise $\texttt{true} \longrightarrow t'$ might hold for some t'. Of course, we can further deduce that there is no such t', but this requires an extra proof step. Hence, the determination of which rule can be applied for t requires more than a purely syntactic analysis of t; it also requires a recursive examination of which rules apply for its sub-terms. While this extra consideration is not an issue for pen-and-paper proofs, it matters when automating mechanized proofs. Of course, given suitable lemmas, these steps can be shortened, but in turn these lemmas have to be maintained over updates in the semantics, making the codebase heavier and more tedious to evolve.

2.2 Abstract Machine Semantics

Operational semantics may rely on an abstract machine to model the executions between semantic states. We defined our toy λ-calculus using substitution, with no need for such a machine, as its semantic states are only defined from the syntax of terms. However, operational semantics of realistic languages rely on environments, *e.g.*, to map variables to values. Pushing this idea further, the reduction judgment sometimes features complex semantic elements that encode information about the state.

Our abstract machine semantics relies on a variant of the CEK abstract machine [1, 23]. Its semantic states rely on a control stack that tell the reduction where to resume when the current subterm is fully reduced. This makes the semantics less readable and elegant than traditional small-step semantics, which are a more human-readable and pleasant form of semantics, ideally suited to a pen-and-paper presentation and reviewing. However, we will later argue that it may not be the best way to mechanize semantics.

The judgment is $s \rightsquigarrow s'$, where states s and s' augment the term t being reduced with extra information about the state of its reduction in control stack κ, that represents the remaining steps that need to be done once t is reduced. We distinguish between evaluation states $\mathcal{S}(t,\kappa,\sigma)$ stepping from t, and return states $\mathcal{C}(v,\kappa)$ when a step returns a value v, namely Boolean values and closures. Contrary to the structural operational semantics presented above, we chose to avoid substitutions in our abstract machine semantics and use environments instead. This may appear to bias the comparison, as we could have chosen to base our abstract machine semantics on the CK abstract machine that uses substitution. But using syntactically-defined environments is more aligned with the

philosophy of this paper, and helps leverage the same proof automation techniques as the rest of the abstract machines in the semantics. Furthermore, our mechanized development for abstract machine semantics handles variables and environments natively, which we compare to our structural operational semantics mechanized development that uses `autosubst` to discharge substitution-related lemmas, hence levelling the playing field between the two styles.

The environment σ maps variables to values. A control stack κ is a list of frames k together with their environment σ. A frame k remembers where a sub-term is being reduced in a larger term. So, there are as many frames as reduction contexts, and σ is a mapping from the variables in the sub-terms of k. For instance, $\texttt{if}(t_1, t_2, \sigma)$ corresponds to $\texttt{if}\ \cdot\ \texttt{then}\ t_1\ \texttt{else}\ t_2$, then $\mathsf{app}(t, \sigma)$ corresponds to $\cdot\ t$, and $\mathsf{fun}\,([x, t, \sigma])$ corresponds to $(\lambda\ x.\ t)\ \cdot$.

Values	v	::=	$[x, t, \sigma] \mid \texttt{true} \mid \texttt{false}$
Environment	σ	::=	$x \mapsto v, \sigma \mid \cdot$
Control Stack	κ	::=	k^*
Frame	k	::=	$\mathsf{app}(t, \sigma) \mid \mathsf{fun}\,([x, t, \sigma]) \mid \texttt{if}(t, t, \sigma)$
State	s	::=	$\mathcal{S}(t, \kappa, \sigma) \mid C(v, \kappa)$

There are three kinds of rules in our semantics, depending on whether a rule *computes* a result (and steps to a C state), or *focuses* on a sub-term (and steps from a $\mathcal{S}$ state to another $\mathcal{S}$), or *resumes* a computation from a result (and steps from a C state to a $\mathcal{S}$ state). For instance, Focus-App *focuses* the step from $t_1\ t_2$ to t_1, keeping t_2 along with the current environment σ in the control stack κ. The value of t_1 is a closure $[x, t_{\text{cl}}, \sigma_{\text{cl}}]$, computed by Compute-Abs. Then, the evaluation of $t_1\ t_2$ resumes by popping from the control stack, and switching to the next frame corresponding to the reduction of t_2 along with the saved environment. When t_2 evaluates to v, then Resume-Beta performs the equivalent of the substitution by updating the environment σ with $x \mapsto v$ and resumes the evaluation of the function body. By switching alternatively between $\mathcal{S}$ and C states, the rules carefully decompose the reduction steps in a very predictable manner.

The rules (see Figure 1c) are easy to read, mainly as they do not update σ; they are not recursive and easy to mechanize inductively. In the end, there are more rules (to focus on a sub-term or resume a step). However, in the structural operational semantics, determining which rule can be applied for a given term t requires more than a purely syntactic analysis of t (which is enough here); it also requires a recursive examination of which rules apply for some sub-terms of t. Thanks to control stack, this extra requirement disappears: the syntactic shape of the initial state completely and uniquely determines which rule to apply. In a nutshell, switching to control stack adds new rules, but their structure is harmonized and more regular, leading to a better proof automation.

3 Proof Ergonomics and Repair: Comparing Styles

This section explains why the abstract machine semantics is easier to mechanize than the structural operational semantics. Indeed, the rules of the former do not have recursive premises stepping on sub-terms, thanks to control stack. So, the behavior of common elementary proof-patterns (that are general enough not to be specific to Rocq) becomes much simpler. In Rocq, it relies on tactics such as `constructor`, aimed at proving a conclusion defined inductively by applying one of its constructors (and in particular applying the adequate semantic rule), and `inversion` aimed at exploiting unused hypotheses. Thus, abstract machine semantics enables us to automate large parts of our proofs, as hinted in CompCert [8].

Before diving into the proof engineering comparison, we emphasize that all results presented in this section are backed by a complete mechanization in Rocq comprising approximately 8,789 lines of code (4,635 lines of specifications and 4,154 lines of proofs). Our formalization includes both semantic styles for the simply-typed λ-calculus with and without conditionals presented here (1,277 lines for basic λ-calculus and 2,317 lines for λ-calculus with conditionals), as well as the more realistic language λ^δ detailed in Section 4 (2,654 lines for the Catala implementation). The development itself is available here [21], with detailed correspondences between paper statements and mechanized proofs documented in the accompanying `CLAIMS.md` file.

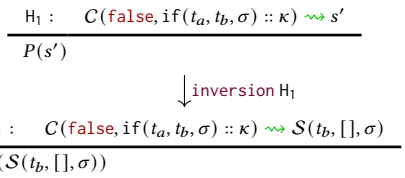
We gathered the insights presented in this section during the development of the mechanized formalization; but rather than presenting the formalization itself here, we preferred to take a step back and look for the root causes of our claims about proof automation, tactics behavior, and maintenance overhead.

3.1 The Effect of Semantic Style on Common Proof Patterns

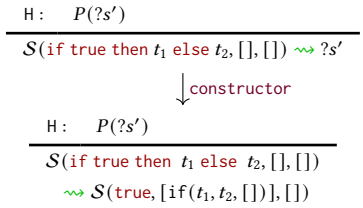
The absence of recursive premises in abstract machine semantics improves proof automation in three different ways. First, it makes hypotheses more precise, and avoids introducing inconsistent proof goals when inverting a hypothesis, which structural operational semantics inevitably does (see 3.1.1). Second, it prevents `constructor` from introducing goals that turn out to be false further on in the proof. This makes automation harder as we need to know which rule leads into false proof goals (see 3.1.2). Third, it allows us to switch the target of inductive reasoning away from recursive judgments and towards the very simple list structure of the control stack, for which we can reuse standard-generated induction principles *as is* (see 3.1.3). We illustrate relevant proof goals in Figure 2. The proof goals are a simplified view of the Rocq interface when conducting proofs using one of our semantics. Each proof goal consists of one or more hypotheses at the top, and a conclusion at the bottom separated by a line. The goal is to show the conclusion from the hypotheses using tactics.

3.1.1 Inverting Judgments. Inverting judgments is a staple of semantic proofs to infer required information, *e.g.*, in Figure 2a, from $C(\texttt{false}, \texttt{if}(t_a, t_b, \sigma) :: \kappa) \rightsquigarrow s'$, we can deduce that $s' = \mathcal{S}(t_b, \kappa, \sigma)$. The `inversion` tactic unifies a hypothesis with every constructor of its type. But we could also have multiple constructors, and `inversion` would generate a new proof goal for each unification, which may trigger further inversions, as shown in Figure 2c when inverting `H1`: this produces both cases $\texttt{true} \longrightarrow t'$ and $t' = t_1$. But inverting a structural operational semantics hypothesis often generates irrelevant proof goals; in our previous examples, $\texttt{true} \longrightarrow t'$ is irrelevant because values cannot reduce.

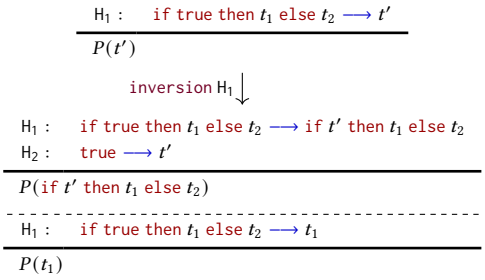
When `inversion` is applied many times in a row, and since each application can generate proof goals, this leads to an exponential growth of the proof cases to analyze and makes the tactic practically unusable in the context of automation. It should be possible to



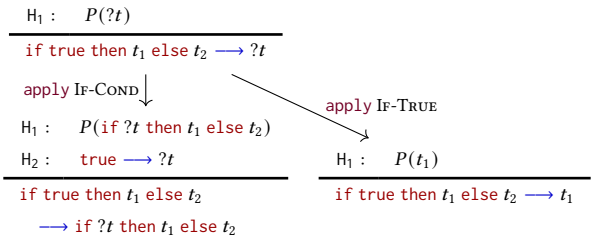
(a) Applying `inversion` on abstract machine semantics. A single proof goal is generated.



(b) Applying `constructor` on abstract machine semantics. The tactic selects the expected Focus-If rule.



(c) Applying `inversion` on structural operational semantics. Two proof goals are generated (separated by the dotted line).



(d) Applying `constructor` on structural operational semantics. It may choose one of the two applicable rules If-Cond or If-True depending on the order of the rules.

Figure 2: Proof goals of the behavior of the `inversion` and `constructor` tactics with both semantics.

modify the rules of a structural operational semantics to avoid this issue by syntactically separating values from terms. However, this is not satisfactory, as values are still terms, so we would need one more inversion to realize that no rule can be applied.

A solution is to add a side condition that t_1 should not be a value, which can be syntactically checked. However, adding non-value premises overloads the semantics in a inconvenient way, and does not fix further proof automation problems. A better solution is to rely on the distinction between evaluation and return states. Hence, `inversion` does not introduce new incoherent proof goals. For instance, inverting $\mathcal{S}(\texttt{if true then } t_1 \texttt{ else } t_2, [], []) \rightsquigarrow s'$, will only give $s' = \mathcal{S}(\texttt{true}, [\texttt{if}(t_1, t_2, \sigma)], [])$. This helps greatly when scaling the proof to more complex constructs.

Such a unique result of an inversion is a meta-property of the reduction judgment, which we check empirically by trying to invert every possible instance, and assert that at most one unique proof goal is generated. It could also be expressed as a MetaCoq [54] statement and proved accordingly. We leave the illustration of this MetaCoq case study as future work.

3.1.2 Introducing Judgments. A common pattern when trying to prove that a step holds is to use `constructor` to choose the possible (and hopefully single) semantic rule. We need to figure out its outcome, which can be achieved using existential variables, noted $?t$, that represent unknown terms that need to be instantiated. So, we generate a step to an existential variable $?t$ and apply `constructor` to instantiate $?t$ when it finds a fitting rule. For instance, rule If-True matches $\texttt{if true then } t_1 \texttt{ else } t_2 \longrightarrow ?t$ and $?t$ becomes t_1.

Contrary to abstract machine semantics, using `constructor` on a structural operational semantics may generate unprovable proof goals. For example, in Figure 2d, when we use `constructor` on $\texttt{if true then } t_1 \texttt{ else } t_2 \longrightarrow ?t$, both If-cond and If-true may be applied. The first one will generate $?t'$ (and $\texttt{true} \longrightarrow ?t'$) and instantiate $?t$ as $\texttt{if } ?t' \texttt{ then } t_1 \texttt{ else } t_2$. The proof goal then becomes $\texttt{if true then } t_1 \texttt{ else } t_2 \longrightarrow \texttt{if } ?t' \texttt{ then } t_1 \texttt{ else } t_2$. This goal holds only if $\texttt{true} \longrightarrow ?t'$, which cannot happen. So, `constructor` can pick from two matching rules but only one (If-true) allows us to conclude the proof. More generally, the fix of adding non-value premises to contextual rules leads to more instances of the problem of picking the right match for `constructor`. There are two workarounds but none is satisfactory. The first one is to reorder the semantic rules, so that `constructor` will select the most precise rule. Sadly, this approach is very brittle, as this hinders proof repair [48]. The second workaround is to avoid using `constructor` and apply manually the correct constructor, but this leads to long and manual proofs.

In contrast, in abstract machine semantics, `constructor` almost always either fails, or selects the only applicable rule, without introducing unprovable goals, as for instance in Figure 2b. Still, some rules have premises (*e.g.*, to check whether a variable belongs to σ). But, thanks to the control stack, they are not recursive for the semantics, contrary to the contextual rules in structural operational semantics. It thus becomes possible to automate their proofs. Furthermore, we know that at most one rule is applicable without considering the premises. Therefore, if a premise is unsolvable, the step itself is unsolvable. So, `constructor` does not introduce any unprovable goal. Overall, `constructor` becomes a very powerful yet simple proof automation tool for abstract machine semantics.

3.1.3 Generating and Applying Induction Principles. The last advantage of abstract machine semantics is the improved handling of induction principles, which is paramount in semantic reasoning.

Indeed, a machine state consists of the remainder of a command and a control stack that describes the context in which it occurs, namely the control flow that is required by the reasoning. So, it becomes easier to reason inductively only on the structure of the control stack, rather than on a mix of case analysis together with rule induction on semantics.

First, let us describe inductive reasoning on structural operational semantics. When a step is recursive, one may need the induction hypothesis H_{ind} to concern the reduction of a deeper sub-term than the direct sub-terms. For instance, proving a predicate P by induction on $(t_1\ t_2)\ t_3 \longrightarrow (t_1'\ t_2)\ t_3$ may require P to hold on $t_1 \longrightarrow t_1'$. Obtaining H_{ind} on deeper sub-terms is challenging because the automatically generated induction principle for $\longrightarrow$ is shallow: it only provides H_{ind} for direct sub-terms. Hence, it would merely state that P holds inductively for $t_1\ t_2 \longrightarrow t_1'\ t_2$. To address this issue, we can either write a custom induction principle (which is error prone) or do the induction on a well-founded measure (which may be tricky to define).

In paper-and-pencil proofs, the induction is often performed on the size of the judgment. Transposing this into proof assistants requires reflection capabilities on the terms, which in turn requires the proof to operate at a meta-level. Concretely, in Rocq, we would have to define our judgments as `Type` instead of `Prop` since we cannot write a function from `Prop` to `nat`. But working with `Type` instead of `Prop` is not recommended since the standard library lemmas on `Prop` cannot be reused, leaving the proof engineer to reprove all trivial results in the meta world. This increases the complexity of the proof engineering effort, making it an undesirable strategy to adopt. Performing induction on the size of terms is the most common solution, as it only requires defining a decreasing measure on terms. Such proofs generally start with performing induction on the size of terms and use `inversion` on the induction hypothesis featuring the semantic judgment. However, this adds an indirection between the induction and the semantic judgment, increasing the size of predicates and intermediate hypotheses. Moreover, it also requires using `inversion`, and thus, issues related to `inversion` may arise again. Finally, each application of the induction hypothesis requires checking that the size is indeed smaller, which may not always be the case. For instance, the β-Reduction rule might increase the size of the terms, requiring the measure to take into account the effects of yet-to-be-performed substitutions.

The other workaround is to write a custom induction principle, but it requires a deep understanding of the proof before writing it, and it is difficult to maintain [48] since it is the synchronized combination of the induction principle and the induction predicate that makes the proof go through. Since the induction principles are not yet defined at this stage, they must be written manually, often requiring low-level tactics or basic lemmas. As a result, any update of the semantics will require tedious manual restating and reproving of the induction principles. By contrast, abstract machine semantics handles induction principles smoothly. Since the judgment is not recursive, inductive reasoning boils down to case analysis. Interestingly, we found that properties such as determinism, progress, and even the correctness of a translation pass require no more than a case analysis of the semantic judgment.

Nevertheless, a stronger truly recursive induction principle is needed to prove some more complex theorems, as illustrated by the proof of the semantic equivalence between $\longrightarrow$ and $\rightsquigarrow$. We found that using well-founded induction on the control stack is sufficient in those cases, particularly due to the contextual reduction property it admits (formalized in Lemma 3.2). Each implication is a theorem proved by a standard simulation of one semantics by the other, involving an invariant $\equiv$ defining the matching between a semantic state and a term. For instance, the state $\mathcal{S}(\texttt{if true then } t_1 \texttt{ else } t_2, [], [])$ matches with the term `if true then` t_1 `else` t_2, and the state $C(\texttt{true}, [\texttt{if}(t_1, t_2, \cdot)])$ with the term `if true then` t_1 `else` t_2. As there are more rules in the abstract machine semantics, a step $\longrightarrow$ may need to be simulated by several steps $\rightsquigarrow$, hence the multi-step execution $\rightsquigarrow^+$ and the 1-n simulation. To utilize the well-founded induction hypothesis, the proof of Theorem 3.1 relies on Lemma 3.2; it states that for each step $\rightsquigarrow$, there is a similar step, but with an extended control stack κ_x to the current state s (written $s +\!\!+ \kappa_x$). For example, $\mathcal{S}(t, \kappa, \sigma) +\!\!+ \kappa_x$ is $\mathcal{S}(t, \kappa +\!\!+ \kappa_x, \sigma)$. This allows us to apply the induction hypothesis on the smaller control stack κ when proving properties about the extended stack $\kappa +\!\!+ \kappa_x$.

Theorem 3.1. *Let s, t, t' be states such that $t \longrightarrow t'$ and $s \equiv t$. Then, there exists s' such that $s \rightsquigarrow^+ s'$ and $s' \equiv t'$.*

Lemma 3.2. *Let s_1, s_2 be states and κ_x a control stack. If $s_1 \rightsquigarrow s_2$, then $s_1 +\!\!+ \kappa_x \rightsquigarrow s_2 +\!\!+ \kappa_x$.*

It is not possible to generate such a compact lemma for $\longrightarrow$, even for two levels of reduction *i.e.*, two frames in κ_x, as it would require a number of distinct lemmas that is quadratic in the number of contextual rules. One way to mitigate this is to add a context in an unified contextual rule. We could then derive a similar lemma but it would still suffer from previous issues related to `inversion` and `constructor`. Lemma 3.2 is very useful in verified compilation proofs as it allows for reusing local results into larger contexts, for instance showing implications between different kinds of simulation diagrams for the same semantics; hence its usefulness outside the narrow scope of internal proofs for the abstract machine semantics.

So, structural induction on the control stack is particularly convenient for handling complex steps as it allows to introduce intermediary steps. Thanks to Lemma 3.2, we replace the control stack with an empty one and apply the induction hypothesis, prove the intermediary needed property, and go back to the control stack inductive cases to generalize our result. While easy proofs can be handled with straightforward case analysis like determinism and progress, induction on the control stack scales remarkably well, for instance for correctness proofs of compilation passes, independently of the level of sub-term inspection required to complete the proof.

3.2 Comparing Semantic Styles and Proof Automation

This section shows that choosing a semantic style with good interaction towards the building blocks of the previous subsection allows for easy, powerful proof automation. Furthermore, our techniques are resilient to semantic changes, facilitating proof maintenance and repair.

Most proofs involving semantics can be structured using the following generic proof outline. The proof begins with an induction on one hypothesis. Then, an inversion on the remaining judgments extracts the necessary information, and the proof proceeds by case analysis. Each case requires applying appropriate tactics and leveraging hand-crafted lemmas. This last phase is the most time consuming and challenging. Hence, one way to mitigate the complexity of the proof is to limit the number of cases where manual intervention is needed, by automating away the handling of a maximum number of cases.

Having a consistent proof outline during proof maintenance greatly simplifies the process. A study on the activity of Rocq proof engineers [49] shows that most of the maintenance time is spent updating definitions, followed by repairing the impacted proofs. By keeping the proof outline unchanged during such changes, only the specific affected cases need to be updated. This ensures that proofs often do not need to be adjusted.

Although heavier automation techniques exist, they often require a deep understanding of the proof assistant's inner workings, which can make maintenance challenging [49]. Instead, we focus on lightweight automation that operates effectively with a well-suited semantic style. This approach works better with the abstract machine semantics style, due to its efficient interaction with `constructor` and `inversion` tactics. We now show how to leverage these benefits in two case studies, where we strive to simplify proofs and reduce their complexity.

3.2.1 Determinism and Progress. Our first case study is the proof that our semantics are deterministic: if $t \longrightarrow t_1$ and $t \longrightarrow t_2$, then $t_1 = t_2$. Its proof is simpler in abstract machine semantics thanks to its interaction with `inversion`. Indeed, the number of cases left to handle manually afterwards differs, and in structural operational semantics, it is quadratic in the number of judgments for each language construct. In contrast, with abstract machine semanitcs, the proof size is constant: basic automation solves all cases. In a more complex setting, we believe that the remaining cases will involve language features where determinism is actually difficult to prove.

In Section 3.1.1, we explained that a partial fix to the problems with the `inversion` is to add extra premises to the structural operational semantics contextual reduction rules specifying that the sub-terms are not values. This partial fix also improves complexity of the determinism proof: the number of cases to handle manually is linear, one for each contextual reduction.

Our second case study is the proof of progress; in structural operational semantics it states that if term t has type τ (written $\vdash t : \tau$), then either there exists t' such that $t \longrightarrow t'$, or t is a value. In abstract machine semantics, it states a similar property, but it relies on the typing of states $\vdash_{\text{state}} s : \tau$ and control stack $\vdash_{\text{control}} \kappa : \tau_1 \Rightarrow \tau_2$ (a trivial extension of term typing).

For structural operational semantics, we first perform induction on typing. Then, `constructor` handles cases where reduction occurs, and syntactic checks identify values. After basic automation, several cases remain, which are inverted and solved using `constructor` or manually (for a few of them). The number of cases scales linearly with the number of constructors, but the proof is brittle: changes to the order of constructors or a missing case analysis can break the proof automation, leading to expensive manual intervention.

For abstract machine semantics, we start by a case analysis on states. For $\mathcal{S}(t, \kappa, \sigma)$, we further perform case analysis on t to handle the different constructors. For $\mathcal{C}(v, \kappa)$, we examine the shapes of v and the top control unit in κ. We then invert the typing judgments, which does not introduce inconsistent goals thanks to the abstract machine semantics. Most of the proof cases left are solved using basic automation, except for a case corresponding to the handling of variables that requires a little extra effort. Overall, the proof is a lot simpler in abstract machine semantics. Moreover, if we forget to perform case analysis (*e.g.*, on an if condition), additional goals are added without introducing inconsistencies contrary to the structural operational semantics proof, as the application of the `constructor` tactic will fail instead of choosing the wrong constructor to instantiate. Hence, using abstract machine semantics is more robust with respect to proof maintenance and repair.

So, maintaining typing definitions and theorems requires effort linear to the number of language constructs. However, abstract machine semantics is more robust: while structural operational semantics can break easily with changes to the order of constructors or missing inductions, abstract machine semantics manages changes better through its handling of control units, supported by custom tactics. This makes these semantics more resilient and easier to maintain when proving progress.

3.2.2 Correctness of a Peep-hole Optimization. Our third case study is the correctness of an optimization (denoted $[\![\cdot]\!]$) that is trickier to prove, as illustrated by several attempts to conduct the proof while reusing already solved sub-proofs. It simplifies nested `if` commands:

$$[\![\texttt{if}\ (\texttt{if}\ b\ \texttt{then}\ \texttt{false}\ \texttt{else}\ \texttt{true})\ \texttt{then}\ t_{\text{then}}\ \texttt{else}\ t_{\text{else}}]\!] = \texttt{if}\ b\ \texttt{then}\ t_{\text{else}}\ \texttt{else}\ t_{\text{then}}$$

We first try to prove the semantics preservation of this optimization pass with a standard 1-1 simulation diagram, but relying on an invariant $\sim$ of our own. The proof reveals that this simulation does not always hold (multiple steps are needed to reduce the original nested `if` *vs.* one step after the optimization), so we generalize the diagram into this 1-n simulation (if $t_1 \longrightarrow t_2$ and $t_1 \sim [\![t_1]\!]$, then there exists $[\![t_2]\!]$ such that $t_2 \sim [\![t_2]\!]$ and $[\![t_1]\!] \longrightarrow^* [\![t_2]\!]$) and make progress in our proof while reusing some proofs from the previous attempt. However, this diagram still does not hold, because the optimization only applies to nested `if` patterns from the initial program (t_1), and not to nested `if` patterns introduced by reduction steps during evaluation (t_2).

Our final solution is to generalize the invariant together with the simulation, leading to trying to prove the following and final simulation diagram: if $t_1 \longrightarrow t_2$ and $t_1 \sim t_1'$, then there exists t_3 and t_3' such that $t_3 \sim t_3'$, $t_2 \longrightarrow^* t_3$ and $t_1' \longrightarrow^* t_3'$. This diagram is iteratively discovered through careful examination of all cases, which may add new cases in the invariant. Efficiency is achieved when existing proofs are reused and straightforward cases are automated, enabling complex scenarios to be given proper attention. The ease with which these proofs are constructed is significantly impacted by the semantics.

The proof of this simulation with structural operational semantics can be done with two alternative strategies. First, an induction on $\sim$ gives very good induction hypotheses, both strong and precise, but this is because we hand-crafted $\sim$ with the relevant information for the proof. Overall, this proof strategy is sound and straightforward, although each of the cases still needs to be manually solved, and the issues raised in Section 3.1 still apply. However, this strategy only works if you have cracked the correct $\sim$ beforehand, and therefore is not advised when starting a verified compilation endeavor. Furthermore, it is impractical for realistic applications because $\sim$ is not always an inductive predicate.

The second strategy is to start by an induction on $\longrightarrow$. It is standard but cumbersome, as it also involves the previous issues of the structural operational semantics. Indeed, the induction principle only gives the induction hypothesis for sub-reductions of depth one, while the rule Trans-If manipulates sub-reductions of depth two. Still, it is possible to make the proof work in this setting, but it requires clever thinking. Indeed, the proof case showing a sub-reduction of depth two involves the optimised nested `if` pattern; we need to know how the nested `if` b `then false else true` reduces to make progress in the proof. Fortunately here, we do not need the full power of the recursive hypothesis to know how this expression reduces (either to `true`, or to `false`, or b takes a step inside); it thus suffices to state and prove the right lemma about this sub-reduction to complete the proof. Hence, using this proof strategy requires deep analysis of the problem at hand and is not solvable with a systematic methodology. Alternatively, one could define a specialized induction principle of depth two for the reduction. However, defining a new induction principle for the needs of each proof does not scale. It is precisely these proof engineering inefficiencies that the use of abstract machine semantics addresses.

Abstract machine semantics enables a more systematic yet flexible proof strategy to show the simulation diagram by performing a well-founded induction on the control stack. This generalizes both previous approaches, offering significant advantages over induction on the semantics (the induction hypothesis is not limited to sub-reductions of depth one), while maintaining more flexibility than invariant-based induction (you do not have to have cracked the correct $\sim$ before starting the proof).

To go on with the proof in abstract machine semantics, we must first adapt the invariant $\sim$ from terms to semantic states, as we did for typing proofs. Then, we start the proof with a well-founded induction on the control stack, and invert both the invariant and the semantics. Applying this strategy leads to a quadratic number of cases: we must consider each possible rule combined with each possible invariant. Moreover, the semantics picks frames from the top of the control stack while the invariant picks frames from the bottom of the stack, which makes reconciliation difficult in each proof case. But this difficulty can be dealt with systematically with the following lemma about reduction refining the progress theorem:

Lemma 3.3. *Suppose that $s_1 +\!\!+ \kappa \rightsquigarrow s_2$ and s_1 is well-typed. Then either there is some v such that $s_1 = C(v, [])$, or there is some s_3 such that $s_1 \rightsquigarrow s_3$ and $s_2 = s_3 +\!\!+ \kappa$.*

With this lemma, the effects of the reduction can be propagated through the control stack stack, helping the reconciliation with the invariant on each proof case.

Then, we can automate most of the reasoning steps of this proof for all the quadratic cases, with a tactic that pattern matches on the current goal to select the right hypothesis or lemma to apply. Finally, we leverage `constructor` to automatically apply reduction steps when needed. By combining the use of abstract machine semantics as well as a pattern-matching-based proof automation by tactic, we claim that our proof methodology is robust to changes in the simulation diagram, semantics or invariants, as most of the proof cases are automated, and the manual cases left to solve do not need to come up with clever lemmas like in structural operational semantics.

4 Scaling Up Proof Automation: Application to Catala and λ^δ

This section shows how the proof engineering differences between semantic styles shown so far scale up with Catala [39], a more realistic language targeted for industrial usage inside government agencies, for computer programs in charge of estimating the amount of taxes and social benefits. These programs have to follow a legal specification, and Catala has special semantic features to account for the peculiar logical structure of the law. Here, the context and details of the Catala compilation chain are irrelevant, as we focus on the core intermediate representation λ^δ of Catala and its default calculus.

4.1 The Intermediate Representation λ^δ

The syntax of our novel mechanization λ^δ of Catala's default calculus is presented in Figure 3. It is a standard λ-calculus, enriched with a *default term*. For instance, let x, y : `bool` be two variables and consider default terms $a = \langle \mid x \text{ : - } \langle 1 \rangle \rangle$, $b = \langle \mid y \text{ : - } \langle 2 \rangle \rangle$ and $c = \langle a, b \mid \texttt{true} \text{ : - } \langle 0 \rangle \rangle$. a and b do not have any exceptions (left of "$\mid$"), hence they behave as conditionals on x and y: if x is `true`, then $a = \langle 1 \rangle$, otherwise $a = \varnothing$ (nothing applies), and likewise for b. Note that the consequence of a default term must have a default type $\langle$`integer`$\rangle$, while 1 has type `integer`. Transforming 1 into a pure default term $\langle 1 \rangle$ solves this problem. However, c is a default term with two exceptions, a and b. If one of the exceptions applies (for instance, $a \neq \varnothing$), then it is returned (and $c = a$). If none apply, then $c = \langle 0 \rangle$ because the justification of the default term (left of " : - ") is `true`. If a and b apply ($x = y =$ `true`), then there is a conflict of exceptions and the computation crashes, returning $c = \circledast$.

In practice and in the Catala computer code, a, b and c are partial definitions of the value of a legal variable, that are activated by a boolean justification and related to each other by exceptional and base case relationships.

A traditional small-step semantics was presented previously for the default calculus in [39]. Its judgment is $t_1 \longrightarrow t_2$. For λ^δ, we modified the semantics in the three following ways (see the detailed rules in the Rocq development). Firstly, we extend the semantics with classic rules for integer values, arithmetic operators, optional constructors and pattern matching, and a fold operator. Secondly, we changed the computation of defaults. Consider $d = \langle \varnothing, t_2 \mid t_{\text{just}} \text{ : - } t_{\text{cons}} \rangle$, where the first exception evaluates to $\varnothing$. In the original semantics, when $t_2 \longrightarrow t_2'$, then d reduces to $\langle \varnothing, t_2' \mid t_{\text{just}} \text{ : - } t_{\text{cons}} \rangle$. Note that $\varnothing$ exceptions are kept while evaluating the subsequent ones. These extra $\varnothing$ were an unpleasant

Types	τ	::=	$\texttt{bool} \mid \texttt{integer} \mid \texttt{unit}$	atomic types
		$\mid$	$\tau \longrightarrow \tau \mid \texttt{option}\ \tau$	function and option type
		$\mid$	$\langle \tau \rangle$	default type
Values	v	::=	$\langle v \rangle$	pure default
		$\mid$	$n \in \mathbb{N} \mid \texttt{true} \mid \texttt{false}$	integers and booleans
		$\mid$	$\texttt{None} \mid \texttt{Some}\ v$	option
Term	t	::=	$x \mid v$	variable, value
		$\mid$	$t\ \texttt{op}\ t \mid \texttt{fold}\ t\ t^*\ t$	binary operator, fold
		$\mid$	$\lambda\, x.\ t \mid t\ t$	λ-calculus
		$\mid$	$\texttt{match}\ t\ \texttt{with} \mid \texttt{None} \longrightarrow t \mid \texttt{Some}\ x \longrightarrow t$	match
		$\mid$	$\texttt{None} \mid \texttt{Some}\ t$	option constructors
		$\mid$	d	default term
Default	d	::=	$\langle t^* \mid t :\!\!- t \rangle$	default term
		$\mid$	$\langle t \rangle$	pure default term
		$\mid$	$\texttt{crash\_if\_empty}\ t$	crash if empty
		$\mid$	$\circledast \mid \varnothing$	conflict and empty error term

Figure 3: Syntax of λ^δ (t^* denotes a list of terms).

source of bookkeeping for little semantic added value, since they do not influence the final value. Hence, we decided to tweak the default evaluation which leads to d reducing to $\langle t_2 \mid t_{\text{just}} :\!\!- t_{\text{cons}} \rangle$. Third, we added a `crash_if_empty` t operator that aborts if the given term is $\varnothing$ and is the identity otherwise, transforming type $\langle t \rangle$ into t. The last modification concerns the typing of defaults and the introduction of a pure default term, that better tracks through typing the program points where $\varnothing$ might appear and propagate.

4.2 Proof Engineering with λ^δ

We proved the classic meta-theoretical results about λ^δ, in both styles: determinism, progress, preservation. We also proved a bi-directional result about the equivalence of the reduction judgments between both semantics. Additionally, we proved in both styles the correctness of a non-trivial compilation pass on λ^δ, an optimization to eliminate default terms in favor of matching option-typed terms where the None case represents the $\varnothing$ value.

This non-trivial compilation pass is illustrated in Figure 4. The translation is straightforward for basic cases: the empty term $\varnothing$ becomes None, the conflict value $\circledast$ remains $\circledast$, and wrapped terms $\langle t \rangle$ translate to Some $[\![t]\!]$. The `crash_if_empty` t operator compiles to a pattern match that returns $\circledast$ when $[\![t]\!]$ reduces to None and preserves the value v when $[\![t]\!]$ reduces to Some v. The most complex translation handles default terms $\langle t_1, t_2, \ldots, t_n \mid t_j :\!\!- t_c \rangle$, which use a fold operation and a merge_option function to combine multiple option values into one, returning $\circledast$ when two Some _ are found. If no exceptions are found, the justification t_j conditionally returns the conclusion t_c; otherwise the selected exception is returned. This compilation pass systematically transforms $\langle \tau \rangle$ types into `option` τ types, preserving the original typing structure while making the handling of potentially absent values explicit through standard option type operations. This improves the older compilation scheme of [39] that used try/catch exceptions in the target language.

As λ^δ is a realistic language, the number of proof cases becomes significant. To manage proofs, we developed automated proof scripts that handle most cases. For each of the main proofs, an automation script is co-developed in Ltac [19], Rocq's tactic language. This script is continually updated with the following methodology: if the current version of our automated script fails on some individual case, we can check this case, determine which hypothesis or lemma is missing to solve the goal automatically, and add it to our proof script. To illustrate this methodology, let us revisit the different parts of a proof with a proof engineering eye. For instance, for the proof of equivalence (Theorem 3.1 page 8), the induction of $t \longrightarrow t'$ followed by the inversion on $s \equiv t$ results in approximately 2760 cases. To handle this bulk of cases, we use proof automation techniques like hypothesis saturation [45] to apply induction principles as well as inversion lemmas. This eliminates most of the inconsistent goals, reducing the number of cases to 344. An example of such an inconsistent goal in this proof would be to try proving that the term reconstructed from state $\mathcal{S}(t_1, \kappa +\!\!+ [\texttt{if}(t_{\text{then}}, t_{\text{else}}, \sigma_2)], \sigma)$ is of the shape $\lambda\, x.\ t'$: because the frame at the bottom of the control stack is $\texttt{if}(t_{\text{then}}, t_{\text{else}}, \sigma_2)$, the reconstruction of the term from the state above necessarily results in a top-level conditional term, and not a λ-term. Our tactics detect such discrepancies and discard these inconsistent goals by contradiction.

From the remaining consistent goals, we step automatically to the remaining cases. This is where we take the most advantage of the automation-friendliness of abstract machine semantics. Indeed, a lot of the goals to solve look like $\exists s', s \rightsquigarrow^* s' \wedge P(s')$, where P is a predicate we want to prove and s a starting state. The proof then consists in figuring out how s reduces to be able to prove P on one of the products of its reduction. This proof pattern requires the ability to reduce the terms inside the proof goal. If our reduction judgment was encoded in Rocq like a function, we could apply it and try to simplify the resulting term, but remember that our judgments are encoded as inductive types. So we need a way to "apply" the inductive type in our proof goal; which in our setting is as simple as the Ltac tactic `solve[constructor; eauto]`. The tactic starts by performing one reduction step in the proof goal, transposing to a meta level what an executable interpreter would do on a term. We can then chain reduction steps in the proof goal with the transitivity lemma for reduction to get full meta-interpretation. This is why we call this kind of tactic a *meta-interpreter*. We improve this automation with the ability to perform steps or full interpretations when we only have partial but sufficient information in our hypotheses. Indeed, a lot of the proofs follow this pattern. Moreover, we can run such a meta-interpreter deep inside our terms and under quantifiers to avoid introducing extra existential variables before goal simplification.

$$
\begin{array}{rcl}
[\![\varnothing]\!] & := & \texttt{None} \qquad [\![\circledast]\!] := \circledast \qquad [\![\langle t \rangle]\!] := \texttt{Some}\ [\![t]\!] \\
[\![\texttt{crash\_if\_empty}\ t]\!] & := & \texttt{match}\ [\![t]\!]\ \texttt{with} \mid \texttt{None} \longrightarrow \circledast \mid \texttt{Some}\ v \longrightarrow v \\
[\![\langle t_1, t_2, \ldots, t_n \mid t_j :- t_c \rangle]\!] & := & \texttt{let}\ \text{merge\_option a b} = \texttt{match}\ a, b\ \texttt{with} \\
 & & \quad \mid \texttt{Some}\ \_, \texttt{Some}\ \_ \longrightarrow \circledast \\
 & & \quad \mid \texttt{None}, \texttt{Some}\ v \mid \texttt{Some}\ v, \texttt{None} \longrightarrow \texttt{Some}\ v \mid \texttt{None}, \texttt{None} \longrightarrow \texttt{None} \\
 & & \texttt{in match fold merge\_option}\ [\ [\![t_1]\!], [\![t_2]\!], \ldots, [\![t_n]\!]\ ]\ \texttt{None with} \\
 & & \mid \texttt{None} \longrightarrow \texttt{if}\ [\![t_j]\!]\ \texttt{then}\ [\![t_c]\!]\ \texttt{else None} \mid \texttt{Some}\ v \longrightarrow \texttt{Some}\ v
\end{array}
$$

Figure 4: Selected compilation rules for eliminating the default term by using option-typed terms. For the sake of clarity, we use `let ... in` . In our mechanized development, the `let ... in` are inlined as abstractions and application.

Overall, we use dozens of these meta-interpreters, each fitting in a few lines of proof script. We make sure the subgoals they generate are proved on the spot by an ad-hoc automation script using basic simplification tactics, so we do not introduce proof goals that do not have the same shape as the ones we are trying to prove. This proof pattern proves to be robust to changes, precisely because it is low-tech: instead of factorizing our proof scripts into complex tactics with a lot of arguments, we copy-paste snippets of a few lines of code as we go, changing in them the names of the lemmas and simplification steps to call depending on the goals at hand. Having the certainty that applying a constructor on a abstract machine semantics term will either correctly reduce, or not reduce at all permits us to confidently use these automation techniques without risking getting stuck in a computation. Hence, abstract machine semantics allows us to benefit almost for free from a very powerful meta-interpretation system that vastly automates semantic proofs. We believe this technique to be the killer application of abstract machine semantics, for scaling up mechanized proof developments.

Another automation technique on which we rely extensively is a smart inversion tactic. Indeed, the `inversion` tactic only works on a specific judgment, meaning that if the proof is updated, then we need to make sure the right judgments are still being inverted. This makes proof repair tedious, especially in compiler verification proofs where we want to invert all judgments depending on terms, such as reduction, invariant and typing judgments. What we would really want is a smart `inversion` tactic for inverting all inductive judgments that concern a specific term on which we have prior information (*e.g.*, t is a function application). Sadly, if we naively repeat `inversion` on all inductive hypotheses, we get a case analysis that loops because of the recursive nature of judgments in structural operational semantics. Our solution is to implement a smart `inversion` tactic using Ltac2 [43] (Rocq's improved tactic language with reflection capabilities) which greatly improves proof repair and helps us develop our proofs in an iterative way. This smart inversion only applies `inversion` when the inductive judgment features a term on which we have prior information (usually, we know the term is in a specific constructor coming from a prior case analysis). For instance it applies `inversion` on $\Gamma \vdash t_1\ t_2 : \tau$ but not on $\Gamma \vdash t : \tau$. We believe the smart inversion trick could be applied to many Rocq developments.

This automation discipline allows us to quickly deal with the trivial aspects of proofs, and dive in specifically to remaining goals. While they might be automated as well, there is a tradeoff between spending time automating them, or simply treating them manually. Empirically, we find that manually handling a dozen cases is the best way to balance this tradeoff. Nevertheless, there are some limitations to our approach. Most importantly, the derivation of induction principles resists automation (see 3.1.3): the presence of both a `list` and an `autosubst` binder in λ^{δ}'s syntax suffices to break the automatic induction principle derivation, which forces us to write some induction principles by hand.

We report proof figures on our most difficult proof, namely the equivalence between both semantics. The part that generalizes Theorem 3.1 relies on structural operational semantics and counts 606 lines of specification and 609 lines of proof. Compare this to 213 lines of specification and 219 lines of proof for the other side, that relies on abstract machine semantics. The case study of the first side helped us understand how we put our automation discipline to work and the pitfalls of traditional small-step semantics in a mechanized development.

5 Conclusion and Future Work

The shape and details of semantic judgments inside a proof assistant directly influence the proof experience for all the theorems involving the semantics. While traditional operational semantics use a recursive inductive judgment, handling it for a small-step semantics becomes tedious in a proof assistant. Specifically, automating proofs about structural operational semantics requires a significant effort in building ad-hoc proof automation frameworks, that themselves can become tricky to maintain. Instead, we make explicit in this paper the advantages of abstract machine small-step semantics, relying on the control stack to represent recursivity in the judgments. This style gives inductive judgments with no recursive premises, and a much more regular shape of inference rules. While harder to parse and compare to a pen-and-paper presentation, abstract machine semantics plays along very well with the basic tactics of proof assistants like Rocq, and allow for building cheap-yet-powerful automation that unlock scaling up the development.

If the use of abstract machine semantics in mechanized developments is not new, the rigorous and principled analysis of the consequences of the semantic style on proof automation presented in this paper opens up new research questions for future work. On a meta level, for instance using MetaCoq, checkers could be developed to test if a given reduction judgment is proof-automation-friendly, before diving into the hardships of the proofs themselves. Second, the trick of syntactically encoding semantic structures to leverage the proof assistant's unification can be ported to other judgments and generalized. Of course, using a lot of syntactic encodings will widen the gap between pen-and-paper formalizations and their mechanized encodings, which should be filled by equivalence proofs so as not to widen the trusted computing base.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming* (Uppsala, Sweden) *(PPDP '03)*. Association for Computing Machinery, New York, NY, USA, 8–19. doi:10.1145/888251.888254

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1807–1823. doi:10.1145/3133956.3134078

[3] Abhishek Anand, Zoe Paraskevopoulou, Andrew Appel, Greg Morrisett, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2021. CertiCoq: A verified compiler for Coq. In *CoqPL*.

[4] Andrew W. Appel and Sandrine Blazy. 2007. Separation Logic for Small-Step CMinor. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4732)*, Klaus Schneider and Jens Brandt (Eds.). Springer, 5–21. https://doi.org/10.1007/978-3-540-74591-4_3

[5] Bob Atkey. 2023. Simple semantics for defaults in Catala. https://bentnib.org/posts/2023-01-16-catala.html

[6] Véronique Benzaken, Évelyne Contejean, Mohammed Houssem Hachmaoui, Chantal Keller, Louis Mandel, Avraham Shinnar, and Jérôme Siméon. 2022. Translating Canonical SQL to Imperative Code in Coq. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 83 (April 2022), 27 pages. doi:10.1145/3527327

[7] Małgorzata Biernacka, Witold Charatonik, and Tomasz Drab. 2022. The Zoo of Lambda-Calculus Reduction Strategies, And Coq. In *13th International Conference on Interactive Theorem Proving (ITP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:19. doi:10.4230/LIPIcs.ITP.2022.7

[8] Sandrine Blazy. 2024. From Mechanized Semantics to Verified Compilation: the Clight Semantics of CompCert. In *Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14573)*, Dirk Beyer and Ana Cavalcanti (Eds.). Springer, 1–21. doi:10.1007/978-3-031-57259-3_1

[9] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288. doi:10.1007/s10817-009-9148-3

[10] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised JavaScript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 87–100.

[11] Martin Bodin, Philippa Gardner, Thomas P. Jensen, and Alan Schmitt. 2018. Skeletal Semantics and their Interpretations. *Proceedings of the ACM on Programming Languages* 3 (2018), 1 – 31. https://api.semanticscholar.org/CorpusID:52846325

[12] Olivier Boite and Catherine Dubois. 2001. Proving type soundness of a simply typed ML-like language with references. *Supplemental Proceedings of TPHOL* 1 (2001), 69–84.

[13] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 586–601. doi:10.1145/3062341.3062358

[14] Timothy Bourke, Basile Pesin, and Marc Pouzet. 2023. Verified Compilation of Synchronous Dataflow with State Machines. *ACM Transactions on Embedded Computing Systems* 22, 5s (Sept. 2023), 137:1–137:26. doi:10.1145/3608102 ESWEEK special issue including presentations at the 23rd Int. Conf. on Embedded Software (EMSOFT 2023).

[15] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth handling of nondeterminism. *ACM Transactions on Programming Languages and Systems* 45, 1 (2023), 1–43.

[16] Arthur Correnson and Dominic Steinhöfel. 2023. Engineering a Formally Verified Automated Bug Finder. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1165–1176. doi:10.1145/3611643.3616290

[17] Nathanaëlle Courant, Antoine Séré, and Natarajan Shankar. 2020. The correctness of a code generator for a functional language. In *Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings 21*. Springer, 68–89.

[18] Guy Cousineau, P-L Curien, and Michel Mauny. 1987. The categorical abstract machine. *Science of computer programming* 8, 2 (1987), 173–202.

[19] David Delahaye. 2000. A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000 Reunion Island, France, November 6–10, 2000 Proceedings 7*. Springer, 85–95.

[20] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. *Acm Sigplan Notices* 50, 1 (2015), 689–700.

[21] Alain Delaët, Sandrine Blazy, and Denis Merigoux. 2025. *Artifact for "Abstract Machines and Small-step Semantics: a Winning Ticket for Proof Automation?"*. doi:10.5281/zenodo.15583348

[22] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2020. Simple High-Level Code for Cryptographic Arithmetic: With Proofs, Without Compromises. *ACM SIGOPS Operating Systems Review* 54, 1 (2020), 23–30.

[23] Matthias Felleisen and Daniel P. Friedman. 1986. *Control Operators, the SECD Machine, and the λ-Calculus*. Technical Report 197. Department of Computer Science, Indiana University.

[24] Pierre Goutagny, Aymeric Fromherz, and Raphaël Monat. 2025. CUTECat: Concolic Execution for Computational Law. In *European Symposium on Programming*. Springer, 31–61.

[25] Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. 2023. Practical and Sound Equality Tests, Automatically: Deriving eqType Instances for Jasmin's Data Types with Coq-Elpi. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, Boston MA USA, 167–181. doi:10.1145/3573105.3575683

[26] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. 2022. Noise: A Library of Verified High-Performance Secure Channel Protocol Implementations. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 107–124.

[27] Jacques-Henri Jourdan. 2016. *Verasco: a Formally Verified C Static Analyzer*. Ph. D. Dissertation. Universite Paris Diderot-Paris VII.

[28] G. Kahn. 1987. Natural semantics. In *STACS 87*, Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Springer, Berlin, Heidelberg, 22–39.

[29] Adam Khayam and Alan Schmitt. 2022. A Practical Approach for Describing Language Semantics. *Submitted for publication* (2022).

[30] Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. *Higher-order and symbolic computation* 20 (2007), 199–207.

[31] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Principles of Programming Languages (POPL)*. ACM Press, 179–191. doi:10.1145/2535838.2535841

[32] P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (01 1964), 308–320. arXiv:https://academic.oup.com/comjnl/article-pdf/6/4/308/1067901/6-4-308.pdf doi:10.1093/comjnl/6.4.308

[33] Sarah B. Lawsky. 2018. A Logic for Statutes. *Florida Tax Review* (2018).

[34] Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language*. Ph. D. Dissertation. INRIA.

[35] John M. Li and Andrew W. Appel. 2021. Deriving Efficient Program Transformations From Rewrite Rules. *Proc. ACM Program. Lang.* 5, ICFP, Article 74 (Aug. 2021), 29 pages. doi:10.1145/3473579

[36] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. *ACM SIGPLAN Notices* 51, 6 (2016), 1–15.

[37] Denis Merigoux. 2021. *Proof-Oriented Domain-Specific Language Design for High-Assurance Software*. Theses. Université Paris sciences et lettres. https://tel.archives-ouvertes.fr/tel-03622012

[38] Denis Merigoux. 2023. Experience Report: Implementing a Real-World, Medium-Sized Program Derived from a Legislative Specification. In *Programming Languages and the Law 2023 (affiliated with POPL)*. Boston (MA), United States. https://inria.hal.science/hal-03933574

[39] Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. 2021. Catala: A Programming Language for the Law. *Proc. ACM Program. Lang.* 5, ICFP, Article 77 (Aug. 2021), 29 pages. doi:10.1145/3473582

[40] Denis Merigoux, Raphaël Monat, and Jonathan Protzenko. 2021. A Modern Compiler for the French Tax Code. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 71–82. doi:10.1145/3446804.3446850

[41] Tobias Nipkow and Gerwin Klein. 2016. Concrete Semantics. *with Isabelle/HOL* 29 (2016).

[42] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *European Symposium on Programming*. https://api.semanticscholar.org/CorpusID:4864472

[43] Pierre-Marie Pédrot. 2019. Ltac2: Tactical Warfare. In *The Fifth International Workshop on Coq for Programming Languages, CoqPL*, Vol. 2019.

[44] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2010. *Software Foundations*. Webpage:http://www.cis.upenn.edu/bcpierce/sf/current/index.html

[45] Clément Pit-Claudel. 2016. *Compilation Using Correct-by-Construction Program Synthesis*. Master's thesis. Massachusetts Institute of Technology. doi:1721.1/107293

[46] Gordon Plotkin. 2004. A Structural Approach to Operational Semantics. *J. Log. Algebr. Program.* 60-61 (07 2004), 17–139. doi:10.1016/j.jlap.2004.05.001
[47] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 151 (June 2023), 25 pages. doi:10.1145/3591265
[48] Talia Ringer. 2021. *Proof Repair.* Ph. D. Dissertation. University of Washington.
[49] Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. 2020. REPLica: REPL Instrumentation for Coq Analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 99–113. doi:10.1145/3372885.3373823
[50] Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 624–641.
[51] Wojciech Krzysztof Rozowski. 2021. *Formally verified derivation of an executable and terminating CEK machine from call-by-value lambda-p-calculus.* Technical Report. University of Southampton.
[52] Grigore Roșu and Traian Florin Șerbănuță. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. doi:10.1016/j.jlap.2010.03.012 Membrane computing and programming.
[53] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015 (LNAI)*, Xingyuan Zhang and Christian Urban (Eds.). Springer-Verlag.
[54] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* 64, 5 (01 Jun 2020), 947–999. doi:10.1007/s10817-019-09540-0
[55] Conrad Watt. 2018. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 53–65. doi:10.1145/3167082
[56] Conrad Watt. 2021. *Mechanising and evolving the formal semantics of WebAssembly: the Web's new low-level language.* Ph. D. Dissertation.
[57] A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (Nov. 1994), 38–94. doi:10.1006/inco.1994.1093

Received 3 June 2025

An Adequate While-Language for Stochastic Hybrid Computation

Renato Neves
nevrenato@di.uminho.pt
University of Minho & INESC-TEC
Braga, Portugal

José Proença
jose.proenca@fc.up.pt
University of Porto & CISTER
Porto, Portugal

Juliana Souza
juliana.p.souza@inesctec.pt
University of Minho & INESC-TEC
Braga, Portugal

Abstract

We introduce a language for formally reasoning about programs that combine differential constructs with probabilistic ones. The language harbours, for example, such systems as adaptive cruise controllers, continuous-time random walks, and physical processes involving multiple collisions, like in Einstein's Brownian motion.

We furnish the language with an operational semantics and use it to implement a corresponding interpreter. We also present a complementary, denotational semantics and establish an adequacy theorem between both cases.

CCS Concepts

• **Theory of computation → Semantics and reasoning**.

Keywords

Semantics, Probabilistic Computation, Hybrid Computation

ACM Reference Format:
Renato Neves, José Proença, and Juliana Souza. 2025. An Adequate While-Language for Stochastic Hybrid Computation. In *Proceedings of the 27th International Symposium on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Motivation. This paper aims at combining two lines of research in programming theory – hybrid and probabilistic programming. Both paradigms are rapidly proliferating and have numerous applications (see *e.g.* [7, 15, 18, 38]), however, despite increasing demand, their combination from a programming-oriented perspective is rarely considered.

Examples abound on why such a combination is crucial, and even extremely simple cases attest to this. Let us briefly analyse one such case. The essence of hybrid programming is the idea of mixing differential constructs with classical program operations, as a way of modelling and analysing computational devices that closely interact with physical processes, such as movement, energy, and electromagnetism. One of the simplest tasks in this context is to move a stationary particle (for illustrative purposes one can regard it as a vehicle) from position zero to position, say, three. A simple hybrid program that implements this task is the following one.

```
p := 0 ;
v := 0 ;
p' = v, v' = 1  for x  ;
p' = v, v' = -1 for y
```

Observe the use of variable assignments which set the vehicle's position (p) and velocity (v) to zero, and note as well the presence of the sequencing operator (;). Most notably, the last two instructions are the aforementioned differential constructs which in this case describe the continuous dynamics of the vehicle at certain stages of the program's execution. Specifically `p' = v, v'= 1 for x` states that the vehicle will *accelerate at the rate of* $1^m/_{s^2}$ *for* x *seconds* while the last instruction states that it will *deccelerate at the rate of* $-1^m/_{s^2}$ *for* y *seconds*. The goal then is to 'solve' the program for x and y so that the vehicle moves and subsequently stops precisely at position three.

Now, the reader has probably noticed that the program just described is an idealised version of reality: there will be noise in the vehicle's actuators, which will cause fluctuations in the acceleration rates, and the switching time between one continuous dynamics to the other is not expected to be precisely x seconds but a value close to it. In face of this issue it is natural to introduce uncertainty factors in the previous program and change the nature of the question *"will my vehicle be at position three at* x+y *seconds?"* to a more probabilistic one, where we ask about probabilities of reaching the desired position instead.

Remarkably, the alternative approach of simply considering *discretisations* of hybrid programs combined with probabilistic constructs (*i.e.* using *purely* probabilistic programming) does not work in general – choosing suitables sizes for the discrete steps can be as hard or even harder than taking the continuous variant, particularly when so-called Zeno behaviour is present [39].

Contributions. We contribute towards a programming theory of stochastic hybrid computation – *i.e.* the combination of hybrid with probabilistic programming. Following traditions of programming theory, we first introduce a simple while-language on which to study stochastic hybrid computation. Our language extends the original while-language [40, 42] merely with the kind of differential construct just seen and with random sampling [25]. Its simplicity is intended, so that one can focus on the core essence of stochastic hybrid computation, but we will see that despite such it already covers a myriad of interesting and well-known examples.

We then furnish the language with an operational semantics, so that we can formally reason about stochastic hybrid programs. Among other things, we use the semantics to extend *Lince* – an existing interpreter of hybrid programs [15, 29] – to an interpreter of stochastic hybrid ones. We will show how this interpreter can be used to automatically present statistical information about the

PPDP'25, Rende, Italy

ACM ISBN 979-8-4007-2085-7
https://doi.org/XXXXXXX.XXXXXXX

program under analysis. All examples in this paper were animated using our extended Lince, available online.[1]

Finally, following the basic motto in programming theory that different semantic approaches are necessary to fully understand the computational paradigm at hand, we introduce a compositional, denotational semantics for our language. In a nutshell, for a given initial state σ a program denotation $[\![\mathtt{p}]\!]$ will correspond to a *Markov process* [35] – intuitively its outputs are given in the form of a probability distribution and are time-dependent. The semantics is built in a systematic way, using basic principles of monad theory [14, 30], measure theory, and functional analysis [2, 11, 35], to which we can then recur (via the semantics) to derive different results about stochastic hybrid computation. We end our contributions with the proof of an adequacy theorem between the operational semantics and the denotational counterpart.

Related work. Whilst research on probabilistic programming is extensive (see *e.g.* [4, 6, 7, 18, 25]), work on marrying it with hybrid programming is much scarcer and mostly focussed on (deductive) verification. The two core examples in this line of research are [37] and [38]. The former presents an extension of the process algebra CSP that harbours both probabilistic and differential constructs. Among other things it furnishes this extension with a process-algebra like, transition-based semantics – which although quite interesting for verification purposes is less amenable to operational perspectives involving *e.g.* implementations. It presents moreover a corresponding proof system for reasoning about certain kinds of Hoare triple. The latter *op. cit.* [38] extends the well-known *differential dynamic logic* with probabilistic constructs. This logic is based on a Kleene-algebraic approach which while has resulted in remarkable progress w.r.t. verification, it is also known to have fundamental limitations in the context of hybrid programming, particularly in the presence of non-terminating behaviour which is frequent in this domain (see details for example in [15, 21, 22]).

A computational paradigm related to hybrid programming is reactive programming [3, 36]. The latter is strongly oriented to the notions of signal and signal transformer (thus in the same spirit of Simulink [43]) and puts emphasis on a declarative programming style (as opposed to a more imperative one, like ours). It has been proposed as a way of modelling hybrid systems, among other things, and recently was extended to a probabilistic setting although no differentiation was involved [5].

Document structure. Section 2 introduces our stochastic language, its operational semantics, and corresponding interpreter. Section 3 recalls a series of measure-theoretic foundations for developing our denotational semantics – which is then presented in Section 4 together with the aforementioned adequacy theorem. Section 5 discusses future work and concludes.

We assume from the reader knowledge of probability theory [11], monads [14, 20, 30], and category theory [28]. A monad will often be denoted in the form of a Kleisli triple, *i.e.* $(T, \eta^T, (-)^{\star^T})$, and whenever no ambiguities arise we will omit the superscripts in the unit and Kleisli operations. Similarly we will sometimes denote a monad just by its functorial component T. Still about notation, we will denote respectively by inl and inr the left $X \longrightarrow X + Y$ and right $Y \longrightarrow X + Y$ injections into a coproduct. We will denote measurable spaces by the letters $X, Y, Z \ldots$ and whenever we need to explicitly work with the underlying σ-algebras we will use instead (X, Σ_X), (Y, Σ_Y), (Z, Σ_Z), and so forth. An extended version of this paper is available online [33].

[1] http://arcatools.org/lince – plots were adapted to avoid relying on colours

2 The language and its operational semantics

We now introduce our stochastic hybrid language. In a nutshell, it extends the hybrid language in [15, 29] with an instruction for sampling from the uniform continuous distribution over the unit interval $[0, 1]$.

We start by postulating a finite set $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ of variables and a stock of *partial* functions $\mathtt{f} : \mathbb{R}^n \rightharpoonup \mathbb{R}$ that contains the usual arithmetic operations, trigonometric ones, and so forth. As usual partiality will be crucial for handling division by 0, logarithms, and square roots, among other things. We then define expressions and Boolean conditions via the following standard BNF grammars,

$$\mathtt{e} ::= \mathtt{x} \mid \mathtt{f(e,\ldots,e)}$$
$$\mathtt{b} ::= \mathtt{e <= e} \mid \mathtt{b \&\& b} \mid \mathtt{b \mid\mid b} \mid \mathtt{tt} \mid \mathtt{ff}$$

where x is a variable in the stock of variables and f is a function in the stock of partial functions. Programs are then built according to the two BNF grammars below.

$$\mathtt{a} ::= \mathtt{x_1' = e\ ,\ldots,\ x_n' = e\ for\ e} \mid \mathtt{x := e} \mid \mathtt{x := unif(0,1)}$$
$$\mathtt{p} ::= \mathtt{a} \mid \mathtt{p\ ;\ p} \mid \mathtt{if\ b\ then\ p\ else\ p} \mid \mathtt{while\ b\ do\ \{\ p\ \}}$$

The first grammar formally describes the three forms that an atomic program a can take: *viz.* a differential operation – expressing a system's continuous dynamics – that will 'run' for e time units, an assignment, and the aforementioned sampling operation. The second grammar formally describes the usual program constructs of imperative programming [40, 42].

We proceed by introducing some syntactic sugar relative to the differential operations and sampling. We will use this sugaring later on to provide further intuitions about the language.

First, observe that the language supports *wait calls* by virtue of the instruction $\mathtt{x_1' = 0\ ,\ldots,\ x_n' = 0\ for\ e}$. The latter states that the system is halted for e time units, as no variable can change during this time period. The operation will be denoted by the more suggestive notation `wait e`. Next, although we have introduced merely sampling from the uniform distribution over $[0, 1]$, it is well-known that other kinds of sampling can be encoded from it. For example, given two real numbers $\mathtt{a} \leq \mathtt{b}$ one can effectively sample from the uniform distribution over [a, b] via the sequence of instructions,

```
x := unif(0,1) ; x := (b - a) * x + a
```

For simplicity we abbreviate such sequence to `x := unif(a,b)`. In the same spirit, it will be useful to sample from the exponential distribution with a given rate `lambda` > 0, in which case we write,

```
x := unif(0,1) ; x := - ln(x)/lambda
```

and abbreviate this program to `x := exp(lambda)`. Next, in order to sample from normal distributions we resort for example to the Box-Muller method [10]. Specifically we write,

```
x1 := unif(0,1) ;
x2 := unif(0,1) ;
x  := sqrt(-2 * (ln x1)) * cos(2 * pi * x2)
```

and suggestively abbreviate the program to `x := normal(0,1)`. The latter amounts to sampling from the normal distribution with mean 0 and standard deviation 1. Note that sampling from a normal distribution with mean `m` and standard deviation `s` is then given by,

```
x := normal(0,1) ; x := m + s * x
```

We abbreviate this last program to `x := normal(m,s)`. We encode Bernoulli trials in our language standardly. Specifically Bernoulli trials, denoted by `bernoulli(r,p,q)`, with $r \in [0,1]$ and `p`, `q` two programs, are encoded as,

```
x := unif(0,1) ; if x <= r then p else q
```

which denotes the evaluation of `p` with probability `r` and `q` with probability `1 - r`. Finally, we will also resort to the usual syntactic sugar constructs in imperative programming, *e.g.* `x := x + 1` as `x++` and so forth.

We are ready to introduce a series of examples written in our programming language. In order to render their descriptions more lively we complement the latter with analysis information given by the aforementioned interpreter. The interpreter as well as the examples are available online at http://arcatools.org/lince.

Example 2.1 (Approximations of normal distributions via random walks). We start with a very simple case that does not involve any differential operation. Specifically we approximate the standard normal distribution via a random walk – a very common procedure in probabilistic programming [4, 24]. Note that this is different from the previous sampling operation `x := normal(0,1)`, in that it does not involve any trigonometric or logarithmic operation; furthermore the resulting distribution will always be discrete. The general idea is to write down the program below.

```
x := 0 ; c := 0 ;
while c <= n do {
      bernoulli(1/2, x++, x--) ; c++
} ;
x := x/sqrt(n)
```

Via an appeal to the central limit theorem [11] one easily sees that the program approximates the expected normal distribution. The parameter `n` refers to the degree of precision, getting a perfect result as $n \longrightarrow \infty$.

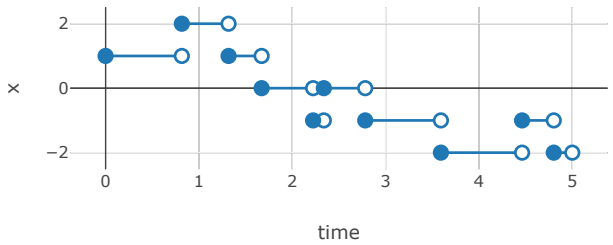


Figure 1: An execution sample of a continuous-time random walk in which the waiting time is given by sampling from the uniform distribution on $[0,1]$.

Example 2.2 (Continuous-time random walks). We now shift our focus from random walks to continuous-time ones [24] – a natural generalisation that introduces uncertainty in the number of steps a walker performs in a given time interval. These are particularly useful for studying anomalous diffusion patterns (*i.e.* the mean squared displacement), with applications ranging from biology to finance [24]. A very simple example in our language is as follows.

```
x := 0 ;
while tt {
      bernoulli(1/2, x++, x--) ;
      d := unif(0,1) ;
      wait d
}
```

The prominent feature is that the walker now waits – according to the uniform distribution on $[0,1]$ – before jumping. A helpful intuition from Nature is to think for example of a pollinating insect that jumps from one flower to another.

Whilst we do not aim at fully exploring the example here, a quick inspection tells that the average waiting time will be $1/2$ and thus the diffusion pattern of this stochastic process grows linearly in time. In accordance to this, Figure 1 presents an execution sample w.r.t. the first 5 time units (horizontal axis) of the process: the plot was given by our interpreter, and indeed shows 10 jumps performed during this period.

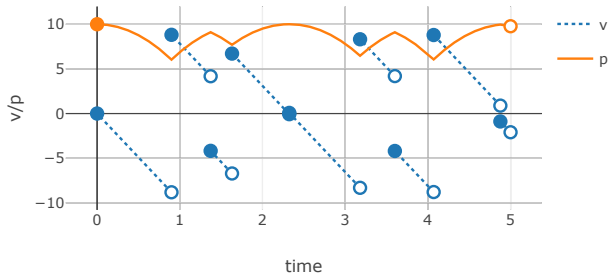


Figure 2: An execution sample of the ball's position (p) and velocity (v) during the first 5 time units.

Example 2.3 (A ball and random kicks). Let us now consider one of the classical examples in hybrid systems theory, *viz.* a bouncing ball [32, 39]. As usual we express the ball's continuous dynamics via a system of differential equations (those of motion) and jumps as assignments that 'reset' velocity. In fact we will consider a variant in which there is no ground for the ball to bounce off. Instead it will be kicked randomly, as encoded in the following program.

```
p := 10 ; v := 0 ;
while tt do {
      d := unif(0,1) ;
      p' = v, v' = -9.8 for d ;
      v := -v
}
```

In a nutshell, the ball moves according to the system of differential equations until it is kicked up (or down) as dictated by `v := -v`. The duration between jumps is random, again with a mean time

of 1/2. Figure 2 presents an execution sample of the bouncing ball during the first 5 time units.

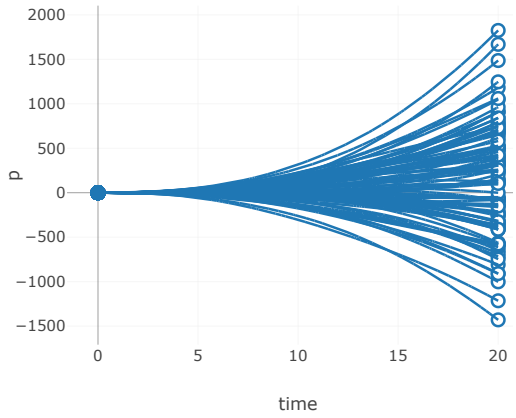


Figure 3: Multiple execution samples of the particle's position overlayed, in order to depict how the position's probability mass spreads over space w.r.t time.

Example 2.4 (Einstein's Brownian motion). Since the last example already moved us so close to it, we might as well consider Einstein's thesis for the cause of Brownian motion [12]. Essentially he posited that the (apparent) erratic motion of a particle suspended in a fluid is due to invisible collisions with atoms and molecules in the liquid. In the one-dimensional setting, one can describe a particular instance of this stochastic process as follows.

```
p := 0 ; v := 0 ; a := 0 ;
while tt do {
      d := exp(lambda) ;
      bernoulli (1/2, a--, a++) ;
      p' = v, v' = a for d
}
```

Note that random collisions are here encoded in the form a Bernoulli trial, and that their frequency is given by the Poisson process prescribed by the sampling operation. Each collision causes a bump in the acceleration (which will either be incremented or decremented). Figure 3 then presents multiple execution samples overlayed on top of each other, in order to depict how the probability mass of the particle's position spreads over space w.r.t. time.

Example 2.5 (Positioning of a particle). We now revisit the task of moving a stationary particle from position $0m$ to position $3m$, using acceleration rates $a = 1m/s^2$ and $a = -1m/s^2$. Recall that the respective program consists in accelerating (with rate $a\ m/s^2$) and then decelerating ($-a\ m/s^2$) the particle for prescribed durations x and y. Now, since a and $-a$ have the same magnitude we can safely assume that x = y. Such durations are then analytically deduced via the observation that,

$$dist = \int_0^t v_a(x)\,dx \ + \ \int_0^t v_{-a}(x)\,dx$$

where $v_a(x) = a \cdot x$ and $v_{-a}(x) = v_a(t) + -a \cdot x$ are respectively the particle's velocity functions w.r.t. the time intervals $[0, t)$ and

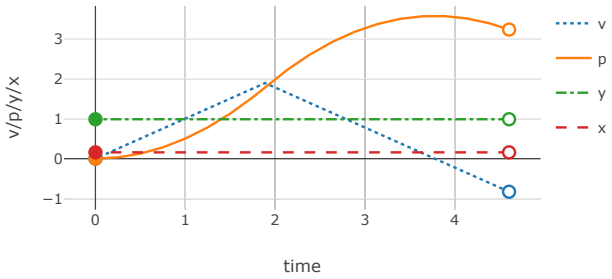


Figure 4: Execution sample of the particle's position (p) and velocity (v).

$[t, 2 \cdot t]$. In this context, t is the value (*i.e.* the duration x) that we wish to find out (see further details in [29]). Then observe that the value $dist$ corresponds to the area of a triangle with basis $2 \cdot t$ and height $v_a(t)$. This geometric shape yields the equations,

$$\begin{cases} dist & = \frac{1}{2} \cdot (2 \cdot t) \cdot v_a(t) \quad \textbf{(area)} \\ v_a(t) & = a \cdot t \quad \textbf{(height)} \end{cases} \qquad \Longrightarrow \qquad t = \sqrt{\frac{dist}{a}}$$

Finally note that for $dist = 3$ and $a = 1$ we obtain $t = \sqrt{3}$. Thus, ideally we would like to set the durations of both acceleration and deceleration to $\sqrt{3}$. This would then give rise to a total duration of $2\sqrt{3}$ and the particle would stop precisely at position 3. However in reality we should expect a small error in the durations of such instructions. To this effect, in the program below we add an uncertainty factor to the calculated durations x and y.

```
x := exp(2) + sqrt(3);
y := exp(2) + sqrt(3);
p' = v, v' = 1  for x ;
p' = v, v' = -1 for y
```

Figure 4 presents an execution sample where we see the effects of the small errors in the durations: specifically the program at some point exceeds position 3 and terminates slightly above it. Note also that the expected shape of the velocity function is lost.

Example 2.6 (Adaptive cruise controller). Lastly we consider a scenario in which a particle tries to follow another one as closely as possible and without crashing into it. For illustration purposes we consider that the following particle (henceforth, the follower) starts 50 meters behind the other one (henceforth, the leader) and that it is stationary. It will be able to either accelerate or brake with forces *e.g.* $2m/s^2$ and $-2m/s^2$, respectively, during 1 time unit each time. The leader, on the other hand, starts with its velocity at $10m/s$ and cannot accelerate or brake.

The follower's choice of whether to accelerate or brake each time is determined by checking whether, in case of choosing to accelerate during one time unit, a 'safe braking distance' from the leader is maintained – a braking distance is determined 'safe' if the follower's braking trajectory does not intersect that of the leader. Technically this amounts to finding the roots of the pointwise difference of both trajectories (*i.e.* finding the roots of a quadratic equation): the absence of roots amounts to the absence of intersections (see further details in [29]). The overall idea of the scenario just described is

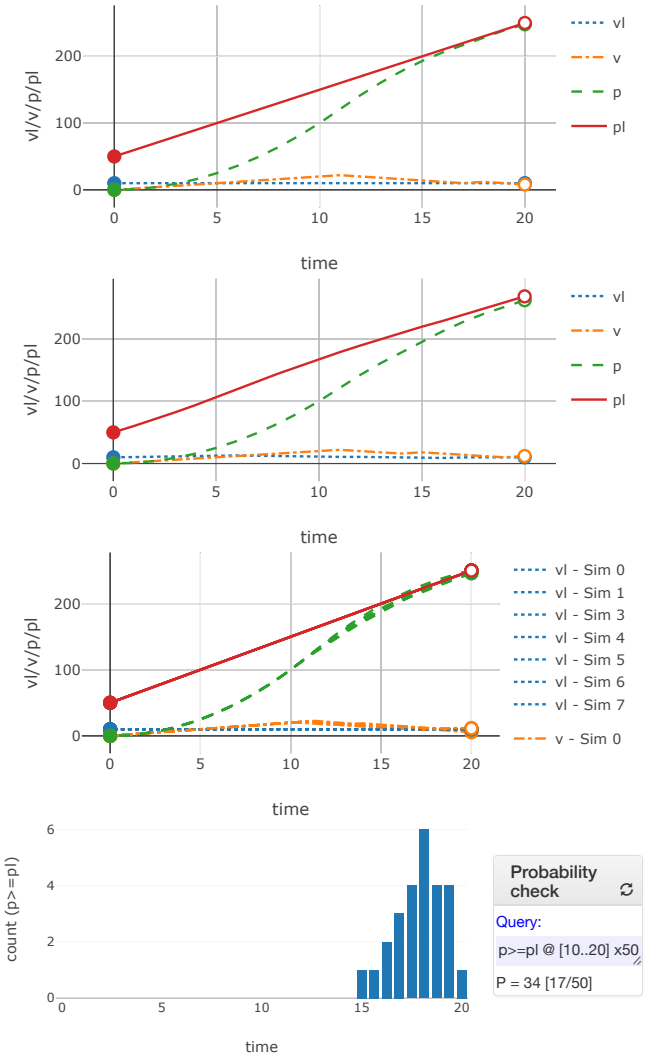


Figure 5: Multiple execution samples of different variants of an adaptive cruise controller. Labels p and v denote the follower's position and velocity while pl and vl indicate the leader's position and velocity. An histogram and a probability checker that count how many times p≥pl in 50 runs with probabilistic waiting times.

encoded by the following program, where the operation `safe(p,v,pl,vl)` informs whether roots were found or not.

```
p := 0 ; v := 0 ; pl := 50 ; vl := 10 ;
while tt {
  if safe(p,v,pl,vl)
  then p' = v, v' =  2, pl' = vl, vl' = 0 for 1
  else p' = v, v' = -2, pl' = vl, vl' = 0 for 1
}
```

Let us now add some uncertainty to the leader: it will be able to uniformly take any acceleration in the range $[-1, 1]$. This means that for *complete safety*, the function `safe(p,v,pl,vl)` needs to be tweaked to assume the *worst possible scenario*: *i.e.* while the follower's braking trajectory will be the same, the leader's trajectory is now assumed to be the one that results from choosing acceleration $-1m/s^2$ (and not $0m/s^2$, as before). The resulting program is then as follows.

```
p := 0 ; v := 0 ; pl := 50 ; vl := 10 ; a := 0;
while tt {
  a := unif(-1,1) ;
  if safe(p,v,pl,vl)
  then p' = v, v' =  2, pl' = vl, vl' = a for 1
  else p' = v, v' = -2, pl' = vl, vl' = a for 1
}
```

Yet another option for introducing uncertainty is to consider the fact that the waiting times will be given by an exponential distribution, as follows.

```
p := 0 ; v := 0 ; pl := 50 ; vl := 10 ;
while tt {
  x := exp(lambda) ; x++ ;
  if safe(p,v,pl,vl)
  then p' = v, v' =  2, pl' = vl, vl' = 0 for x
  else p' = v, v' = -2, pl' = vl, vl' = 0 for x
}
```

The function `safe(p,v,pl,vl)` would then need to be tweaked again – but remarkably now with no hope for complete safety, as in theory `x` can take any value from $[1, \infty)$ and thus no worst-case scenario exists.

Figure 5 (top) presents an execution sample in which the system is completely deterministic (*i.e.* the leader's velocity is constant with 100% certainty) and thus the follower gets as close as possible to the leader. On the other hand, Figure 5 (middle) presents an execution sample in which the follower assumes the worst-case scenario just described and thus cannot get as close to the leader. Finally Figure 5 (bottom plot) presents several execution samples overlayed in which the original `safe` function is used and the respective durations are given by the exponential distribution `1 + exp(8)`. It shows that, while collisions are improbable they do occur. This low probability of collision is depicted in the histogram at the bottom of Figure 5, which counts how many times `p >= pl` holds in 50 runs over time (bottom-left) and anywhere in the interval [10,20] (bottom-right).

Operational semantics. The section's remainder is devoted to introducing an operational semantics for the language – not only such is a basis for formal reasoning about stochastic hybrid programs it is also the engine of the interpreter that we have been showcasing thus far. In a nutshell, the semantics marries Kozen's operational semantics for a probabilistic language [25] with the semantics of hybrid programs that was presented in [15, 29]. We will need some preliminaries.

We take the Hilbert cube $[0, 1]^\omega$ as the source of randomness. Operationally speaking this means that sampling will amount to drawing values from an element of $[0, 1]^\omega$ (a stream) that is fixed *a priori*. For example, sampling once will amount to taking the head of this element and sampling n times will amount to taking the respective prefix of size n. Next we assume that the semantics of expressions e and Boolean conditions b are given by partial maps $[\![\mathtt{e}]\!] : \mathbb{R}^n \rightharpoonup \mathbb{R}$ and $[\![\mathtt{b}]\!] : \mathbb{R}^n \rightharpoonup \{\mathtt{tt}, \mathtt{ff}\}$. These can be defined

in the usual way. Now, since we are in the context of imperative programming we will recur to the notion of a store $\sigma : \{\mathtt{x}_1, \ldots, \mathtt{x}_n\} \to \mathbb{R}$ (also known as memory or environment) [40, 42]. It assigns a real number to any given variable in the language. For a store σ, we will use the notation $\sigma[\mathtt{x} \mapsto v]$ to denote the store that is exactly like σ except for the fact that x is now assigned value v. Finally we assume that any system of differential equations in our language induces a continuous map $\phi : \mathbb{R}^n \times \mathbb{R}_{\geq 0} \rightharpoonup \mathbb{R}^n$ – which in our context will be regarded as the respective solution. For simplicity we denote an operation `x1' = e1,...,xn' = en for e` by the simpler expression `diff(e1,...,en,e)`.

The rules of our semantics are then presented in Figure 6. They dictate what the next computational step will be when evaluating a program p with initial store σ w.r.t. time instant t. As alluded before each evaluation is also associated with a source of randomness $s \in [0,1]^\omega$ from which p draws sampling results. Note from the rules that such computational steps (coming out of $\mathtt{p}, \sigma, t, s$) lead to one of three possible outcomes: *viz.* an error flag *err*, an output store, or a resumption (*i.e.* an updated evaluation stack of programs, store, time instant, and source of randomness) which can then be evaluated in the next step (the empty stack is denoted by *skip*).

A subtle feature of our semantics is that the time instant t at the beginning of the evaluation will tend to decrease along the computational steps performed. Intuitively this means that the evaluation is 'moving forward in time' until reaching the target time instant t that we wish to evaluate. Most notably when it detects that such time instant was surpassed it forces the termination of the evaluation, even if the evaluation stack of programs is currently non-empty. Such is expressed by the rule **(seq-stop$^{\rightarrow}$)**, and is crucial for evaluating non-terminating programs, like the ones described in Example 2.2, Example 2.3, Example 2.4, and Example 2.6. We illustrate this feature next with a simple example, but more details can also be found in [15].

Example 2.7. Consider the following non-terminating program,

```
x := 0 ; while tt { x++ ; wait 1 }
```

Although the loop involved does not terminate, one can always evaluate the program in a finite amount of steps for any given time instant. Let us see what happens, for example, at time instant $1 + 1/2$. First, for simplicity we denote the loop simply by p and the store $\sigma : \{\mathtt{x}\} \to \mathbb{R}$ that is defined as $\sigma(\mathtt{x}) = v$ by $\mathtt{x} \mapsto v$. We then deduce the following sequence of small-step transitions which arise from the rules in Figure 6.

$$
\begin{aligned}
\mathtt{x:=\ 0\ ;\ p}, \sigma, 1 + 1/2, s &\to \mathtt{p}, (\mathtt{x} \mapsto 0), 1 + 1/2, s \\
&\to \mathtt{x{+}{+}\ ;\ wait\ 1\ ;\ p}, (\mathtt{x} \mapsto 0), 1 + 1/2, s \\
&\to \mathtt{wait\ 1\ ;\ p}, (\mathtt{x} \mapsto 1), 1 + 1/2, s \\
&\to \mathtt{p}, (\mathtt{x} \mapsto 1), 1/2, s \\
&\to \mathtt{x{+}{+}\ ;\ wait\ 1\ ;\ p}, (\mathtt{x} \mapsto 1), 1/2, s \\
&\to \mathtt{wait\ 1\ ;\ p}, (\mathtt{x} \mapsto 2), 1/2, s \\
&\to \mathtt{x} \mapsto 2
\end{aligned}
$$

The last transition arises precisely due to rules **(diff-stop$^{\rightarrow}$)** and **(seq-stop$^{\rightarrow}$)**, which are applicable due to $1/2$ being *strictly* smaller than the wait call's duration (*viz.* 1). Henceforth we will call events such as the one just described *time-based terminations*, in order to distinguish from those ordinary terminations that originate from emptying the program evaluation stack. We will see later on that this subtle aspect can be neatly handled in the denotational context via an exception monad.

Let us briefly mention how our interpreter uses this semantics to provide (overlayed) execution samples of a given stochastic hybrid program p. The basic idea is simple: we first generate an entropy source *i.e.* a sample $s \in [0,1]^\omega$ and then use it to compute the execution chain $\mathtt{p}, \sigma, t, s \to \ldots$ for multiple time instants t, corresponding to different snapshots of the program's behavioural trajectory. In order to obtain overlayed execution samples one just repeats this process multiple times, *i.e.* with different samples.

We now introduce a big-step operational semantics in Figure 7, which abstracts from intermediate computational steps in the context of the small-step variant. Although in programming theory big-step semantics have multiple applications [40, 42], here we use it just to connect the small-step variant to the denotational counterpart (in Section 4). In other words, we use the big-step variant as a midpoint between small-step and denotational semantics. Since this big-step semantics is based on the same ideas as the small-step one, we skip its explanation.

We conclude the section by showing that the small-step and big-step semantics agree, in the sense that they give rise to the same input-output relation. Technically we will need to factor in a form of transitive closure from the small-step relation, as follows. We call 'terminal' those tuples arising from steps (in the small-step semantics) that are of the form *skip*, σ, t, s, or σ, or *err*. Then we build an input-output relation ($\Rightarrow$) from the small-step semantics in the way detailed in Figure 8. Finally,

Theorem 2.8. *For every program* p*, store* σ*, time instant* t*, and source of randomness* s*, we have the following following equivalence:*

$$\mathtt{p}, \sigma, t, s \Downarrow v \quad \textit{iff} \quad \mathtt{p}, \sigma, t, s \Rightarrow v$$

Proof. The right-to-left direction follows by induction on the length of small-step reduction sequences and Lemma 2.9. The left-to-right direction follows by induction over big-step derivations. □

Lemma 2.9. *For a program* p*, a store* σ*, time instant* t*, and a source of randomness* s*, the following holds: if* $\mathtt{p}, \sigma, t, s \to \mathtt{p}', \sigma', t', s'$ *and* $\mathtt{p}', \sigma', t', s' \Downarrow v$ *then* $\mathtt{p}, \sigma, t, s \Downarrow v$.

Proof. Follows straightforwardly by induction over small-step derivations. □

3 Measure theory

This section briefly recalls a series of results about measure theory [2, 11, 35], focus being on those that form the backbone of the semantics described in Section 4.

Our main working category will be Meas, *i.e.* that of measurable spaces and measurable functions. Recall that it has both (infinite) products and coproducts [1, Section 21]. Recall as well that it is distributive, *i.e.* for all measurable spaces X, Y, Z there exists a certain isomorphism,

$$\mathrm{dist} : X \times (Y + Z) \to X \times Y + X \times Z$$

$$\textbf{(asg-rnd}^{\rightarrow}\textbf{)}\quad \mathtt{x := unif(0,1)}, \sigma, t, (h:s) \rightarrow \mathit{skip}, \sigma[\mathtt{x} \mapsto h], t, s$$

$$\textbf{(asg}^{\rightarrow}\textbf{)}\quad \mathtt{x := e}, \sigma, t, s \rightarrow \mathit{skip}, \sigma[\mathtt{x} \mapsto [\![\mathtt{e}]\!](\sigma)], t, s \qquad ([\![\mathtt{e}]\!](\sigma) \text{ defined})$$

$$\textbf{(asg-err}^{\rightarrow}\textbf{)}\quad \mathtt{x := e}, \sigma, t, s \rightarrow \mathit{err} \qquad ([\![\mathtt{e}]\!](\sigma) \text{ undefined})$$

$$\textbf{(diff-stop}^{\rightarrow}\textbf{)}\quad \mathtt{diff(e_1,\ldots,e_n,e)}, \sigma, t, s \rightarrow \phi(\sigma, t) \qquad ([\![\mathtt{e}]\!](\sigma) > t)$$

$$\textbf{(diff-skip}^{\rightarrow}\textbf{)}\quad \mathtt{diff(e_1,\ldots,e_n,e)}, \sigma, t, s \rightarrow \mathit{skip}, \phi(\sigma, [\![\mathtt{e}]\!](\sigma)), t - [\![\mathtt{e}]\!](\sigma), s \qquad (0 \leq [\![\mathtt{e}]\!](\sigma) \leq t)$$

$$\textbf{(diff-err}^{\rightarrow}\textbf{)}\quad \mathtt{diff(e_1,\ldots,e_n,e)}, \sigma, t, s \rightarrow \mathit{err} \qquad ([\![\mathtt{e}]\!](\sigma) < 0 \text{ or } [\![\mathtt{e}]\!](\sigma) \text{ undefined})$$

$$\textbf{(if-true}^{\rightarrow}\textbf{)}\quad \mathtt{if\ b\ then\ p\ else\ q}, \sigma, t, s \rightarrow \mathtt{p}, \sigma, t, s \qquad ([\![\mathtt{b}]\!](\sigma) = \mathtt{tt})$$

$$\textbf{(if-false}^{\rightarrow}\textbf{)}\quad \mathtt{if\ b\ then\ p\ else\ q}, \sigma, t, s \rightarrow \mathtt{q}, \sigma, t, s \qquad ([\![\mathtt{q}]\!](\sigma) = \mathtt{ff})$$

$$\textbf{(if-err}^{\rightarrow}\textbf{)}\quad \mathtt{if\ b\ then\ p\ else\ q}, \sigma, t, s \rightarrow \mathit{err} \qquad ([\![\mathtt{b}]\!](\sigma) \text{ undefined})$$

$$\textbf{(wh-true}^{\rightarrow}\textbf{)}\quad \mathtt{while\ b\ do\ \{\,p\,\}}, \sigma, t, s \rightarrow \mathtt{p\ ;\ while\ b\ do\ \{\,p\,\}}, \sigma, t, s \qquad ([\![\mathtt{b}]\!](\sigma) = \mathtt{tt})$$

$$\textbf{(wh-false}^{\rightarrow}\textbf{)}\quad \mathtt{while\ b\ do\ \{\,p\,\}}, \sigma, t, s \rightarrow \mathit{skip}, \sigma, t, s \qquad ([\![\mathtt{b}]\!](\sigma) = \mathtt{ff})$$

$$\textbf{(wh-err}^{\rightarrow}\textbf{)}\quad \mathtt{while\ b\ do\ \{\,p\,\}}, \sigma, t, s \rightarrow \mathit{err} \qquad ([\![\mathtt{b}]\!](\sigma) \text{ undefined})$$

$$\textbf{(seq-stop}^{\rightarrow}\textbf{)}\quad \frac{\mathtt{p}, \sigma, t, s \rightarrow \sigma'}{\mathtt{p\ ;\ q}, \sigma, t, s \rightarrow \sigma'} \qquad \textbf{(seq-skip}^{\rightarrow}\textbf{)}\quad \frac{\mathtt{p}, \sigma, t, s \rightarrow \mathit{skip}, \sigma', t', s'}{\mathtt{p\ ;\ q}, \sigma, t, s \rightarrow \mathtt{q}, \sigma', t', s'}$$

$$\textbf{(seq-err}^{\rightarrow}\textbf{)}\quad \frac{\mathtt{p}, \sigma, t, s \rightarrow \mathit{err}}{\mathtt{p\ ;\ q}, \sigma, t, s \rightarrow \mathit{err}} \qquad \textbf{(seq}^{\rightarrow}\textbf{)}\quad \frac{\mathtt{p}, \sigma, t, s \rightarrow \mathtt{p}', \sigma', t', s'}{\mathtt{p\ ;\ q}, \sigma, t, s \rightarrow \mathtt{p}'\mathtt{\ ;\ q}, \sigma', t', s'}$$

Figure 6: Small-step operational semantics.

Now, let Top be the category of topological spaces and continuous maps, and recall that it has (infinite) products and coproducts as well [1, Section 21]. There exists a functor $B : \mathbf{Top} \to \mathbf{Meas}$ that sends any given topological space to the measurable space with the same carrier and equipped with the respective Borel σ-algebra [2, Section 4.4]. In particular when treating a subset of real numbers as a measurable space we will be tacitly referring to the respective Borel σ-algebra. It is well-known that B preserves finite products of second-countable topological spaces [17, Definition 6.3.7], which is the case for example of Polish spaces (see [2, Chapter 3] and [17, Theorem 6.3.44]). This property is key for our semantics: it will allow us to treat solutions of systems of differential equations – which are continuous functions and thus live in Top – as measurable functions. Further details about this crucial aspect are available in the following section. We proceed by briefly recalling basic results about measures – a more detailed description is available for example in [2, Chapter 10], [4, Chapter 1], and [35, Chapter 2].

Definition 3.1. For a measurable space (X, Σ_X) a signed measure is a function $\mu : \Sigma_X \to \mathbb{R}$ such that $\mu(\emptyset) = 0$ and moreover it is σ-additive, *i.e.*

$$\mu\left(\bigcup_{i=1}^{\infty} U_i\right) = \sum_{i=1}^{\infty} \mu(U_i)$$

where $(U_i)_{i\in\omega}$ is any family of pairwise disjoint measurable sets. A signed measure will be simply called a measure if $\mu(U) \geq 0$ for all measurable subsets $U \subseteq X$. As usual a measure is called a subdistribution if $\mu(X) \leq 1$ and a distribution if $\mu(X) = 1$.

For a measurable space X the set of signed measures $\mathrm{M}(X)$ forms a vector space via pointwise extension. It also forms a normed space when equipped with the total variation norm,

$$\|\mu\| = \sup\left\{\sum_{i=1}^{n} |\mu(U_i)| \mid \{U_1, \ldots, U_n\} \text{ measurable partition of } X\right\}$$

In particular for a measure μ we have $\|\mu\| = \mu(X)$. Note that $\mathrm{M}(X)$ is also a Banach space by virtue of the reals numbers forming a Banach space, specifically the limit of a Cauchy sequence is built via pointwise extension.

Take subdistributions $\mu \in \mathrm{M}(X)$ and $\nu \in \mathrm{M}(Y)$. There exists the so-called *tensor* or *product measure* $\mu \otimes \nu \in \mathrm{M}(X \times Y)$, which is defined by the equation $\mu \otimes \nu(U \times V) = \mu(U)\nu(V)$ on all measurable rectangles $U \times V \in \Sigma_{X\times Y}$. Specifically the latter extends standardly to all measurable sets by an appeal to Carathéodory's extension and moreover the extension is unique (see *e.g.* [2, Lemma 10.33]). Another useful fact is that for any subdistributions $\mu, \nu \in \mathrm{M}(X)$ and $\rho \in \mathrm{M}(Y)$ the equation below holds.

$$(\mu + \nu) \otimes \rho = \mu \otimes \rho + \nu \otimes \rho$$

The product measure construction just described also applies to countable families of distributions $(\mu_i)_{i\in\omega}$ in $\mathrm{M}(X)$.

$$\textbf{(asg-rnd)}\ \frac{}{\texttt{x := unif(0,1)}, \sigma, t, (h : s) \Downarrow \mathit{skip}, \sigma[\texttt{x} \mapsto h], t, s}$$

$$\textbf{(asg-skip)}\ \frac{[\![\texttt{e}]\!](\sigma)\ \text{defined}}{\texttt{x := e}, \sigma, t, s \Downarrow \mathit{skip}, \sigma[\texttt{x} \mapsto [\![\texttt{e}]\!](\sigma)], t, s} \qquad \textbf{(asg-err)}\ \frac{[\![\texttt{e}]\!](\sigma)\ \text{undefined}}{\texttt{x := e}, \sigma, t, s \Downarrow \mathit{err}}$$

$$\textbf{(diff-skip)}\ \frac{0 \le [\![\texttt{e}]\!](\sigma) \le t}{\texttt{diff}(\texttt{e}_1, \ldots, \texttt{e}_n, , \texttt{e}), \sigma, t, s \Downarrow \mathit{skip}, \phi(\sigma, [\![\texttt{e}]\!](\sigma)), t - [\![\texttt{e}]\!](\sigma), s}$$

$$\textbf{(diff-stop)}\ \frac{[\![\texttt{e}]\!](\sigma) > t}{\texttt{diff}(\texttt{e}_1, \ldots, \texttt{e}_n, , \texttt{e}), \sigma, t, s \Downarrow \phi(\sigma, t)} \qquad \textbf{(diff-err)}\ \frac{[\![\texttt{e}]\!](\sigma) < 0 \text{ or } [\![\texttt{e}]\!](\sigma)\ \text{undefined}}{\texttt{diff}(\texttt{e}_1, \ldots, \texttt{e}_n, , \texttt{e}) \Downarrow \mathit{err}}$$

$$\textbf{(seq-skip)}\ \frac{\texttt{p}, \sigma, t, s \Downarrow \mathit{skip}, \sigma', t', s' \qquad \texttt{q}, \sigma', t', s' \Downarrow v}{\texttt{p ; q}, \sigma, t, s \Downarrow v}$$

$$\textbf{(seq-stop)}\ \frac{\texttt{p}, \sigma, t, s \Downarrow \sigma'}{\texttt{p ; q}, \sigma, t, s \Downarrow \sigma'} \qquad \textbf{(seq-err)}\ \frac{\texttt{p}, \sigma, t, s \Downarrow \mathit{err}}{\texttt{p ; q}, \sigma, t, s \Downarrow \mathit{err}}$$

$$\textbf{(if-true)}\ \frac{[\![\texttt{b}]\!](\sigma) = \texttt{tt} \qquad \texttt{p}, \sigma, t, s \Downarrow v}{\texttt{if b then p else q}, \sigma, t, s \Downarrow v}$$

$$\textbf{(if-false)}\ \frac{[\![\texttt{b}]\!](\sigma) = \texttt{ff} \qquad \texttt{q}, \sigma, t, s \Downarrow v}{\texttt{if b then p else q}, \sigma, t, s \Downarrow v} \qquad \textbf{(if-err)}\ \frac{[\![\texttt{b}]\!](\sigma)\ \text{undefined}}{\texttt{if b then p else q}, \sigma, t, s \Downarrow \mathit{err}}$$

$$\textbf{(wh-true)}\ \frac{[\![\texttt{b}]\!](\sigma) = \texttt{tt} \qquad \texttt{p ; while b do \{ p \}}, \sigma, t, s \Downarrow v}{\texttt{while b do \{ p \}}, \sigma, t, s \Downarrow v}$$

$$\textbf{(wh-false)}\ \frac{[\![\texttt{b}]\!](\sigma) = \texttt{ff}}{\texttt{while b do \{ p \}}, \sigma, t, s \Downarrow \mathit{skip}, \sigma, t, s} \qquad \textbf{(wh-err)}\ \frac{[\![\texttt{b}]\!](\sigma)\ \text{undefined}}{\texttt{while b do \{ p \}}, \sigma, t, s \Downarrow \mathit{err}}$$

Figure 7: Big-step operational semantics.

$$\frac{\texttt{p}, \sigma, t, s \to v}{\texttt{p}, \sigma, t, s \Rightarrow v}\ (v\ \text{terminal}) \qquad \frac{\texttt{p}, \sigma, t, s \to \texttt{p}', \sigma', t', s' \qquad \texttt{p}', \sigma', t', s' \Rightarrow v}{\texttt{p}, \sigma, t, s \Rightarrow v}$$

Figure 8: Big-step semantics via the small-step relation.

Next, for a measurable space X we will denote by $\mathrm{G}(X)$ the set of subdistributions. The construct $\mathrm{G}(-)$ thus defined forms the Giry monad in Meas when every $\mathrm{G}(X)$ is equipped with the σ-algebra generated by the evaluation maps,

$$\mathrm{eval}_U : \mathrm{G}(X) \to \mathbb{R} \qquad \mu \mapsto \mu(U) \qquad (U \subseteq X \text{ measurable})$$

[34]. The respective Kleisli morphisms $f : X \to \mathrm{G}(Y)$ are typically called Markov kernels and their Kleisli extension $f^\star : \mathrm{G}(X) \to \mathrm{G}(Y)$ is given by Lebesgue integration [2, Chapter 11],

$$f^\star(\mu) = U \mapsto \int_{x \in X} f(x)(U)\, d\mu(x)$$

For every measurable space X the unit $\delta : X \to \mathrm{G}(X)$ of this monad is given by the Dirac delta $\delta_x \in \mathrm{G}(X)$ with $x \in X$. In particular the functorial action of $\mathrm{G} : \mathrm{Meas} \to \mathrm{Meas}$ is the pushforward measure operation. We will often abuse notation by denoting a linear combination $\sum_i p_i \cdot \delta_{x_i}$ simply by $\sum_i p_i \cdot x_i$ with $x_i \in X$.

Let us recall useful properties about the Giry monad. First it is commutative when equipped with the double-strength operation $\mathrm{G}(X) \times \mathrm{G}(Y) \to \mathrm{G}(X \times Y)$ defined via the product measure [41]. The fact that such operation is measurable follows from [2, Lemma 4.11]. Second for any bounded measurable map $f : X \to \mathbb{R}$ and measures $\mu, \nu \in \mathrm{G}(X)$ Lebesgue integration satisfies the conditions,

$$\int f d(\mu + \nu) = \int f d\mu + \int g d\nu \qquad \int f d(s \cdot \mu) = s \cdot \left(\int f d\mu \right)$$

for any scalar $s \in \mathbb{R}$. Thus we immediately conclude that the Kleisli extension of a Markov kernel will always be linear. Third if the codomain of f restricts to $[0, 1]$ we obtain,

$$\int f d\mu \le \mu(X) = \|\mu\|$$

This entails that the Kleisli extension of a Markov kernel will be bounded and thus continuous (even contractive) w.r.t. the metric induced by the total variation norm. This provides a number of tools from functional analysis. For example one can analyse how $f^\star$ acts on a measure μ by a series of approximations μ_n to μ. Not only this, the set of maps $\mathrm{Meas}(\mathrm{G}(X), \mathrm{G}(Y))$ can be equipped with the metric induced by the operator norm,

$$\|T\| = \sup \{ \|T(\mu)\| \mid \mu \in \mathrm{G}(X), \|\mu\| \le 1 \}$$

which moves us beyond classical program equivalence by allowing to compare programs in terms of distances and not just equality (see for example [9]). More details about this last aspect will be given later on.

Another useful fact is that for a given measure $\mu \in \mathrm{G}(X)$ any measurable subset $U \subseteq X$ gives rise to a new measure $\mu(U \cap -)$. Moreover for any bounded measurable map $f : X \to \mathbb{R}$ and measure $\mu \in \mathrm{G}(X)$ we obtain,

$$\int_X f d\mu = \int_U f d\mu(U \cap -) + \int_{\overline{U}} f d\mu(\overline{U} \cap -)$$

where $\overline{U}$ represents the complement of U. Yet another useful property of the Giry monad is that for every measurable space X the space $\mathrm{G}(X)$ inherits the usual order on the real numbers, via pointwise extension. What is more, the induced order has a bottom element (the zero-mass measure) and it is ω-complete, by virtue of the completeness property of the real numbers. Remarkably, an ω-increasing sequence of measures $(\mu_n)_{n\in\omega}$ in $\mathrm{G}(X)$ is Cauchy and $\sup_{n\in\omega} \mu_n = \lim_{n\to\infty} \mu_n$, thanks to the monotone convergence theorem (see *e.g.* [2, Theorem 11.18] or [35, Theorem 3.6]). This is helpful to jump between domain theory and functional analysis whenever necessary.

Next, observe that the aforementioned order extends to Markov kernels via pointwise extension. It has a bottom element (the map constant on the zero-mass measure) and it is ω-complete. The last property follows directly from the definition of the σ-algebra of $\mathrm{G}(X)$ for every X and from the fact that the pointwise supremum of real-valued measurable functions is measurable. Also, it follows from the monotone convergence theorem that the equation,

$$(\sup_{i\in\omega} f_i)^\star = \sup_{i\in\omega} f_i^\star$$

holds for any increasing sequence $(f_i)_{i\in\omega}$ of Markov kernels. This will be crucial for the interpretation of while-loops in the following section. Finally it follows from the fact that multiplication preserves suprema that,

$$(\sup_{i\in\omega} \mu_i) \otimes \nu = \sup_{i\in\omega} \mu_i \otimes \nu$$

for any increasing sequence of subdistributions $(\mu_i)_{i\in\omega}$ in $\mathrm{G}(X)$ and $\nu \in \mathrm{G}(Y)$.

4 Denotational semantics

We now introduce a denotational, measure-theoretic semantics for our stochastic language. In a nutshell, it extends Kozen's well-known probabilistic semantics [25] with a mechanism for handling the time-based terminations that were described in Section 2. The extension boils down to the following categorical construction.

Any object E in a category $\mathbb{C}$ with binary coproducts induces a monad $E + (-)$ which intuitively gives semantics to exception handling [31]. It follows from the universal property of coproducts that any monad T in $\mathbb{C}$ combines with $E + (-)$. In other words we have a new monad $T \otimes E$ which handles at the same time effects arising from T and exceptions. Concretely the functorial action of this new monad is given by $T(E + (-))$, the unit $\eta^{T\otimes E}$ by the composition $\eta^T \cdot \mathrm{inr} : X \to T(E+X)$, and the Kleisli lifting $(-)^{\star^{T\otimes E}}$ by the equation,

$$f^{\star^{T\otimes E}} = [\eta^T \cdot \mathrm{inl}, f]^{\star^T} \tag{1}$$

Time-based terminations will be handled precisely via one such monad $\mathrm{G} \otimes E$ in Meas – in other words these terminations are technically seen as exceptions, in the sense that they also inhibit the execution of subsequent computations and instead are merely propagated forward along the evaluation. Specifically the denotation $[\![\mathsf{p}]\!]$ of a program p will be a Markov kernel $X \to \mathrm{G}(E + X)$ in which elements of E denote time-based terminations. The space X will be in particular the product $\mathbb{R}^n \times \mathbb{R}_{\geq 0}$ whilst E will be $\mathbb{R}^n$ (thus analogously to Section 2, n is the cardinality of our stock of variables and possible outputs are either elements of $\mathbb{R}^n \times \mathbb{R}_{\geq 0}$ or $\mathbb{R}^n$). Consequently, for every $(\sigma, t) \in \mathbb{R}^n \times \mathbb{R}_{\geq 0}$ we have $[\![\mathsf{p}]\!](\sigma, t)$ as a subdistribution which assigns probabilities to the outputs of p w.r.t. time instant t and initial state σ. For any given initial state $\sigma \in \mathbb{R}^n$, one can also see a denotation $[\![\mathsf{p}]\!]$ as inducing a Markov process $[\![\mathsf{p}]\!](\sigma, -)$ which intuitively means that the subdistribution of outputs evolves over time. Note as well that the possibility of the total mass of $[\![\mathsf{p}]\!](\sigma, t)$ being strictly lower than 1 for a given input $(\sigma, t) \in \mathbb{R}^n \times \mathbb{R}_{\geq 0}$ reflects the possibility of divergence and/or errors in the evaluation of expressions and Boolean conditions.

Lastly in order to interpret while-loops, we observe that the combined monad $\mathrm{G} \otimes E$ inherits the order of G, and moreover,

$$(\sup_{i\in\omega} f_i)^{\star^{\mathrm{G}\otimes E}} = \sup_{i\in\omega} f_i^{\star^{\mathrm{G}\otimes E}} \tag{2}$$

for any increasing sequence $(f_i)_{i\in\omega}$ of Markov kernels. This last equation follows from the Scott-continuity of co-pairing on its second argument.

We are finally ready to introduce our denotational semantics. It is defined in Figure 9, via induction on the syntactic structure of programs. It assumes, as usual, that the semantics of expressions e and Boolean conditions b are given by *measurable* partial maps $[\![\mathsf{e}]\!] : \mathbb{R}^n \rightharpoonup \mathbb{R}$ and $[\![\mathsf{b}]\!] : \mathbb{R}^n \rightharpoonup \{\mathtt{tt}, \mathtt{ff}\}$. The measurability of each interpretation clause in Figure 9 is then straightforward to verify. Indeed, the only somewhat complicated case is the first clause, which crucially relies on two related properties. First, the fact that every continuous map $\phi : \mathbb{R}^n \times \mathbb{R}_{\geq 0} \to \mathbb{R}^n$ is measurable as a map $B(\phi) : B(\mathbb{R})^n \times B(\mathbb{R}_{\geq 0}) \to B(\mathbb{R})^n$ (which we commented on in the last section). Second, by an analogous reasoning, the fact that the subtraction $\mathbb{R} \times_{\mathsf{Meas}} \mathbb{R} \to \mathbb{R}$ map is measurable. This entails in particular that the strictly greater relation $(>)$ is measurable as a function $\mathbb{R} \times_{\mathsf{Meas}} \mathbb{R} \to 1+1$, by virtue of $(-\infty, 0)$ being a measurable subset of $\mathbb{R}$. Next, observe our slight abuse of notation in $\sigma[\mathsf{x}_i \mapsto \lambda]$ which abbreviates the product measure,

$$1 \cdot \sigma(\mathsf{x}_1) \otimes \ldots \otimes 1 \cdot \sigma(\mathsf{x}_{i-1}) \otimes \lambda \otimes 1 \cdot \sigma(\mathsf{x}_{i+1}) \otimes \ldots \otimes 1 \cdot \sigma(\mathsf{x}_n)$$

where λ is the uniform distribution on $[0, 1]$. Finally the last clause interprets while-loops via Kleene's least fixpoint construction. The fact that the map from which we take the least fixpoint is Scott-continuous follows straightforwardly from our previous observations and in particular Equation (2).

The section's remainder is devoted to proving adequacy of our denotational semantics w.r.t. the operational counterpart that was described in Section 2. In order to achieve this – and following the same steps as [25] – we will recur to an auxiliary semantics, which reframes our operational semantics as a *measurable* map. We will see that such is necessary in order to sensibly extend the input-output relation induced by the operational semantics to a probabilistic setting – and thus subsequently connect the latter to

$$[\![\texttt{diff}(\texttt{e}_1,\ldots,\texttt{e}_n,\texttt{e})]\!] = (\sigma,t) \mapsto \begin{cases} 1\cdot\phi(\sigma,t) & \text{if } [\![\texttt{e}]\!](\sigma) > t \\ 1\cdot(\phi(\sigma,[\![\texttt{e}]\!](\sigma)), t - [\![\texttt{e}]\!](\sigma)) & \text{if } 0 \le [\![\texttt{e}]\!](\sigma) \le t \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\texttt{x} := \texttt{e}]\!] = (\sigma,t) \mapsto \begin{cases} 1\cdot(\sigma[\texttt{x} \mapsto [\![\texttt{e}]\!](\sigma)], t) & \text{if } [\![\texttt{e}]\!](\sigma) \text{ is well-defined} \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\texttt{x}_i := \texttt{unif}(\texttt{0},\texttt{1})]\!] = (\sigma,t) \mapsto \sigma[\texttt{x}_i \mapsto \lambda] \otimes (1\cdot t)$$

$$[\![\texttt{if b then p else q}]\!] = (\sigma,t) \mapsto \begin{cases} [\![\texttt{p}]\!](\sigma,t) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{tt} \\ [\![\texttt{q}]\!](\sigma,t) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{ff} \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\texttt{p ; q}]\!] = [\![\texttt{q}]\!]^\star \cdot [\![\texttt{p}]\!]$$

$$[\![\texttt{while b do p}]\!] = \mathrm{lfp}\left(k \mapsto (\sigma,t) \mapsto \begin{cases} k^\star\cdot[\![\texttt{p}]\!](\sigma,t) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{tt} \\ 1\cdot(\sigma,t) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{ff} \\ 0 & \text{otherwise} \end{cases}\right)$$

Figure 9: Denotational semantics.

the denotational semantics, as intended. Let us thus proceed by presenting this auxiliary semantics.

We will need some preliminaries. Recall that any object E in a category C with binary coproducts induces a monad $E + (-)$. We take the particular case in which $\mathsf{C} = \mathsf{Meas}$ and $E = 1$. We then equip the Kleisli morphisms of this monad with the partial order that is induced from the notion of a flat domain [13]. It is easy to see that this order is ω-complete by an appeal to the following well-known theorem.

Theorem 4.1. *Consider an increasing sequence of measurable maps $(f_i)_{i\in\omega} : X \to 1 + Y$. Their supremum w.r.t. the order of flat domains is also measurable.*

Proof. Recall that the supremum $f : X \to 1 + Y$ of $(f_i)_{i\in\omega}$ is defined by,

$$f(x) = \begin{cases} \mathrm{inr}(y) & \text{if } \exists i \in \omega.\, f_i(x) = \mathrm{inr}(y) \text{ for some } y \in Y \\ \mathrm{inl}(*) & \text{otherwise} \end{cases}$$

(see details for example in [13]). Now, we need to show that f is measurable, or in other words that both pre-images $f^{-1}(\{\mathrm{inl}(*)\})$ and $f^{-1}(\mathrm{inr}[U])$ are measurable (where U is any measurable subset of Y). For the first case we reason,

$$\begin{aligned} f^{-1}(\{\mathrm{inl}(*)\}) &= \{x \in X \mid f(x) = \mathrm{inl}(*)\} \\ &= \{x \in X \mid \forall i \in \omega.\, f_i(x) = \mathrm{inl}(*)\} \\ &= \bigcap_{i=0}^{\infty} f_i^{-1}(\{\mathrm{inl}(*)\}) \end{aligned}$$

and this intersection must be measurable because σ-algebras are closed under countable intersections. The second case follows from an analogous reasoning and the fact that σ-algebras are closed under countable unions. □

Observe then that for any increasing sequence of measurable maps $(f_i)_{i\in\omega} : X \to 1 + Y$ we have,

$$(\sup_{i\in\omega} f_i)^\star = \sup_{i\in\omega} f_i^\star$$

thanks to Scott-continuity of co-pairing on its second argument. Denoting this monad by $(-)_\perp$, observe that its Kleisli category $\mathsf{Meas}_{(-)_\perp}$ is isomorphic to PMeas, *i.e.* that of measurable spaces and partial measurable maps. PMeas has binary coproducts by general categorical results [30]. We then take as the interpretation domain of our auxiliary semantics the monad $E + (-)$ in PMeas where $E = \mathbb{R}^n$. In order to interpret while-loops via this monad, note that it inherits the ω-complete order of $(-)_\perp$ and furthermore,

$$(\sup_{i\in\omega} f_i)^\star = \sup_{i\in\omega} f_i^\star \tag{3}$$

The operational semantics in functional form is now presented in Figure 10. Observe that, contrary to the denotational semantics, it involves an entropy source. The measurability of each interpretation clause is once again straightforward to verify, and similarly for the fact that the map from which we take the least fixpoint is Scott-continuous, thanks to Equation (3). The symbol $*$ represents undefinedness.

Finally the following theorem establishes the aforementioned connection between the operational semantics in Figure 7 and the functional semantics that we have just presented.

Theorem 4.2. *Consider a program* p*, an environment σ, a time instant t, and an entropy source s. Then the following implications hold:*

$$\begin{aligned} \texttt{p},\sigma,t,s \Downarrow \mathit{skip},\sigma',t',s' &\quad\Rightarrow\quad (\!|\texttt{p}|\!)(\sigma,t,s) = (\sigma',t',s') \\ \texttt{p},\sigma,t,s \Downarrow \sigma' &\quad\Rightarrow\quad (\!|\texttt{p}|\!)(\sigma,t,s) = \sigma' \\ \texttt{p},\sigma,t,s \Downarrow \mathit{err} &\quad\Rightarrow\quad (\!|\texttt{p}|\!)(\sigma,t,s) = * \end{aligned}$$

Moreover the following implications also hold:

$$\begin{aligned} (\!|\texttt{p}|\!)\,(\sigma,t,s) = (\sigma',t',s') &\quad\Rightarrow\quad \texttt{p},\sigma,t,s \Downarrow \mathit{skip},\sigma',t',s' \\ (\!|\texttt{p}|\!)\,(\sigma,t,s) = \sigma' &\quad\Rightarrow\quad \texttt{p},\sigma,t,s \Downarrow \sigma' \end{aligned}$$

Proof. The proof is laborious but straightforward. The first three implications follow by induction over the big-step derivations trees, a close inspection of Kleisli composition, and the fixpoint equation concerning while-loops. The last two implications follow

$$(\!|\texttt{diff}(\texttt{e}_1,\ldots,\texttt{e}_n,\texttt{e})|\!) = (\sigma,t,s) \mapsto \begin{cases} \phi(\sigma,t) & \text{if } [\![\texttt{e}]\!](\sigma) > t \\ (\phi(\sigma,[\![\texttt{e}]\!](\sigma)), t - [\![\texttt{e}]\!](\sigma), s) & \text{if } 0 \le [\![\texttt{e}]\!](\sigma) \le t \\ * & \text{otherwise} \end{cases}$$

$$(\!|\texttt{x} := \texttt{e}|\!) = (\sigma,t,s) \mapsto \begin{cases} (\sigma[\texttt{x} \mapsto [\![\texttt{e}]\!](\sigma)], t, s) & \text{if } [\![\texttt{e}]\!](\sigma) \text{ is well-defined} \\ * & \text{otherwise} \end{cases}$$

$$(\!|\texttt{x}_i := \texttt{unif}(\texttt{0},\texttt{1})|\!) = (\sigma,t,(h:s)) \mapsto (\sigma[\texttt{x}_i \mapsto h], t, s)$$

$$(\!|\texttt{if b then p else q}|\!) = (\sigma,t,s) \mapsto \begin{cases} (\!|\texttt{p}|\!)\,(\sigma,t,s) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{tt} \\ (\!|\texttt{q}|\!)\,(\sigma,t,s) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{ff} \\ * & \text{otherwise} \end{cases}$$

$$(\!|\texttt{p ; q}|\!) = (\!|\texttt{q}|\!)^\star \cdot (\!|\texttt{p}|\!)$$

$$(\!|\texttt{while b do p}|\!) = \mathrm{lfp}\left(k \mapsto (\sigma,t,s) \mapsto \begin{cases} k^\star \cdot (\!|\texttt{p}|\!)\,(\sigma,t,s) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{tt} \\ (\sigma,t,s) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{ff} \\ * & \text{otherwise} \end{cases}\right)$$

Figure 10: Functional version of the big-step semantics in Figure 7.

by structural induction on programs, again a close inspection of Kleisli composition, and the proof that for all $i \in \omega$ the following implications hold,

$$f_i(\sigma,t,s) = (\sigma',t',s') \quad \Rightarrow \quad \texttt{while b do p},\sigma,t,s \Downarrow \mathit{skip},\sigma',t',s'$$
$$f_i(\sigma,t,s) = \sigma' \quad \Rightarrow \quad \texttt{while b do p},\sigma,t,s \Downarrow \sigma'$$

where the maps $(f_i)_{i\in\omega} : X \rightharpoonup E + X$ are the components of the supremum involved in Kleene's least fixpoint construction. The proof for these two last implications is obtained via induction over the natural numbers. □

Note that the previous theorem excludes the implication,

$$(\!|\texttt{p}|\!)\,(\sigma,t,s) = * \qquad \Rightarrow \qquad \texttt{p},\sigma,t,s \Downarrow \mathit{err}$$

This is simply because the equation $(\!|\texttt{p}|\!)\,(\sigma,t,s) = *$ may arise from divergence (and not necessarily from an evaluation error) which the operational semantics cannot track. On the other hand, the theorem entails that if $(\!|\texttt{p}|\!)\,(\sigma,t,s) = *$ and the operational semantics evaluates the tuple $\texttt{p},\sigma,t,s$ to a value then this value is necessarily *err*.

We are now ready to extend the auxiliary semantics $(\!|-|\!)$ to a probabilistic setting. Such hinges on the fact that the pushforward measure construction $(-)_\star$ is functorial on partial measurable maps. This yields the composite functor,

$$\mathsf{PMeas}_E \xrightarrow{(-)^\star} \mathsf{PMeas} \xrightarrow{(-)_\star} \mathsf{Meas}$$

where PMeas_E is the Kleisli category of the monad $E+(-)$ in PMeas and $(-)^\star$ is the respective Kleisli extension. More concretely we obtain the inference rule,

$$\frac{f : X \rightharpoonup E + Y}{(f^\star)_\star : \mathrm{G}(E+X) \to \mathrm{G}(E+Y)}$$

which, intuitively, entails that for any $(\!|\texttt{p}|\!)$ the function $((\!|\texttt{p}|\!)^\star)_\star$ moves probability masses of the measure given as input according to the operational semantics. This is analogous to what happens with the denotational semantics; and our adequacy theorem will render such analogy precise.

We are ready to formulate our adequacy theorem. In order to keep notation simple we will abbreviate the space $\mathbb{R}^n \times \mathbb{R}_{\ge 0}$ to X and, as before, the space $\mathbb{R}^n$ to E. Note that any measure $\mu \in \mathrm{G}(E+X)$ can be decomposed into $\mu_{|E} \in \mathrm{G}(E)$ and $\mu_{|X} \in \mathrm{G}(X)$, where $\mu_{|E}(U) = \mu(U + \emptyset)$ for all measurable subsets $U \subseteq E$ and analogously for $\mu_{|X}$. These restriction operations are linear, commute with suprema, and furthermore $\mu = \mathrm{inl}_\star(\mu_{|E}) + \mathrm{inr}_\star(\mu_{|X})$. We will often abuse notation and elide the left and right injections in the previous measure decomposition.

Theorem 4.3 (Adequacy). *Consider a program* p *and let* λ *be the uniform distribution on* $[0,1]$. *For any measure* $\mu \in \mathrm{G}(E+X)$, *the following equation holds.*

$$((\!|\texttt{p}|\!)^\star)_\star\left(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega}\right) = [\![\texttt{p}]\!]^\star(\mu)_{|E} + [\![\texttt{p}]\!]^\star(\mu)_{|X} \otimes \lambda^{\otimes\omega}$$

In particular the equation below holds for all measurable subsets $U \subseteq E$ *and* $V \subseteq X$.

$$((\!|\texttt{p}|\!)^\star)_\star\left(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega}\right)(U + V \times [0,1]^\omega) = [\![\texttt{p}]\!]^\star(\mu)(U+V)$$

Proof. The proof is obtained via structural induction. The base cases follow straightforwardly although laborious. The case of sequential composition also follows straightforwardly, thanks to the functorial laws and the Kleisli extension laws. We then focus on the case of conditionals. We will need to decompose the measure μ in the cases that it satisfies and does not satisfy b: *i.e.* we will denote $\mu(\emptyset + [\![\texttt{b}]\!] \cap -)$ by ν and $\mu(\emptyset + \overline{[\![\texttt{b}]\!]} \cap -)$ by ρ, where slightly overloading notation we set $[\![\texttt{b}]\!] = \{(\sigma,t) \in X \mid [\![\texttt{b}]\!](\sigma) = \texttt{tt}\}$. Then,

$$((\!|\texttt{if b then p else q}|\!)^\star)_\star\left(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega}\right)$$
$$= \{\mu_{|X} = (\nu+\rho)_{|X}\}$$
$$((\!|\texttt{if b then p else q}|\!)^\star)_\star\left(\mu_{|E} + (\nu+\rho)_{|X} \otimes \lambda^{\otimes\omega}\right)$$

$= \{$ Addition of measures distributes over tensor $\}$

$(\!(\texttt{if b then p else q})\!)^\star)_\star\Big(\mu_{|E} + \nu_{|X} \otimes \lambda^{\otimes\omega} + \rho_{|X} \otimes \lambda^{\otimes\omega}\Big)$

$= \{$ Linearity + Semantics definition $\}$

$\mu_{|E} + ((\!|\texttt{p}|\!)^\star)_\star\Big(\nu_{|X} \otimes \lambda^{\otimes\omega}\Big) + ((\!|\texttt{q}|\!)^\star)_\star\Big(\rho_{|X} \otimes \lambda^{\otimes\omega}\Big)$

$= \{$ Induction hypothesis $\}$

$\mu_{|E} + [\![\texttt{p}]\!]^\star(\nu)_{|E} + [\![\texttt{p}]\!]^\star(\nu)_{|X} \otimes \lambda^{\otimes\omega} + [\![\texttt{q}]\!]^\star(\rho)_{|E} + [\![\texttt{q}]\!]^\star(\rho)_{|X} \otimes \lambda^{\otimes\omega}$

$= \{$ Addition of measures distributes over tensor $\}$

$\mu_{|E} + ([\![\texttt{p}]\!]^\star(\nu) + [\![\texttt{q}]\!]^\star(\rho))_{|E} + ([\![\texttt{p}]\!]^\star(\nu) + [\![\texttt{q}]\!]^\star(\rho))_{|X} \otimes \lambda^{\otimes\omega}$

$= \{$ Semantics definition $\}$

$[\![\texttt{if b then p else q}]\!]^\star(\mu)_{|E} + ([\![\texttt{p}]\!]^\star(\nu) + [\![\texttt{q}]\!]^\star(\rho))_{|X} \otimes \lambda^{\otimes\omega}$

$= \{$ Semantics definition $\}$

$[\![\texttt{if b then p else q}]\!]^\star(\mu)_{|E} + [\![\texttt{if b then p else q}]\!]^\star(\mu)_{|X} \otimes \lambda^{\otimes\omega}$

Lastly we focus on the case of while-loops. Thus let $(f_i)_{i\in\omega}$ be the family of maps involved in Kleene's fixpoint construction w.r.t. the semantics $(\!|-|\!)$ and analogously for $(g_i)_{i\in\omega}$ and the semantics $[\![-]\!]$. We need to show that for all $i \in \omega$,

$$(f_i^\star)_\star\Big(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega}\Big) = g_i^\star(\mu)_{|E} + g_i^\star(\mu)_{|X} \otimes \lambda^{\otimes\omega} \qquad (4)$$

This is obtained via induction over the natural numbers. Note that the reasoning is similar to the case concerning conditionals so we omit this step. Then,

$(\!(\texttt{while b do p})\!)^\star)_\star(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega})$

$= \{$ Semantics definition $\}$

$((\sup_{i\in\omega} f_i)^\star)_\star(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega})$

$= \{$ Equation (3) $\}$

$((\sup_{i\in\omega} f_i^\star))_\star(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega})$

$= \{$ Monotone convergence theorem $\}$

$(\sup_{i\in\omega}(f_i^\star)_\star)(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega})$

$= \{$ Equation (4) $\}$

$\sup_{i\in\omega} g_i^\star(\mu)_{|E} + g_i^\star(\mu)_{|X} \otimes \lambda^{\otimes\omega}$

$= \{$Addition commutes with sup. + Tensor distribute over sup. $\}$

$\sup_{i\in\omega} g_i^\star(\mu)_{|E} + \Big(\sup_{i\in\omega} g_i^\star(\mu)_{|X}\Big) \otimes \lambda^{\otimes\omega}$

$= \{$ Equation (2) + Semantics definition $\}$

$[\![\texttt{while b do p}]\!]^\star(\mu)_{|E} + [\![\texttt{while b do p}]\!]^\star(\mu)_{|X} \otimes \lambda^{\otimes\omega}$

□

5 Conclusions and future work

This paper provides a basis towards a programming framework of stochastic hybrid systems. It is rooted not only on an operational semantics, which we used in the implementation of an interpreter, but also on a compositional, measure-theoretic counterpart, with which one can formally reason about program equivalence and approximating behaviour, among other things. These contributions open up several interesting research lines. We briefly detail next the ones we are currently exploring.

First, the fact that we committed ourselves to a denotational semantics based on monads will now allow us to capitalise on the more general, extensive theory of monad-based program semantics. This includes for example the extension of our semantics with additional computational effects [30], with higher-order features and different evaluation mechanisms [23, 27], and corresponding logics as well as predicate transformer perspectives [16, 19]. We are already working on the last two topics [16, 19], for not only they potentially offer a complementary tool in our framework of stochastic hybrid programs they would also facilitate a more natural connection between our work and previous results on deductive verification of stochastic hybrid systems [37, 38].

Second, we would like to add new probabilistic constructs to our language, most notably conditioning [7, 26] and stochastic differential equations which have multiple uses in hybrid systems theory [37, 38].

Third, recall from Section 3 and Section 4 that in our semantics program denotations are *contractive* operators $\mathrm{G}(E{+}X) \to \mathrm{G}(E{+}X)$. Thus following the observations in [8, 9], such forms the basis of a corresponding theory of *metric* program equivalence. Concretely, instead of comparing two program denotations in terms of *equality* we are able to systematically compare them in terms of distances. In the setting of stochastic hybrid programming this is a much more desirable approach, as in practice it is unrealistic to expect that two programs match their outputs with exact precision. Not only this, reasoning about program distances is extremely important for computationally simulating such programs, since we are limited to certain finite precision aspects and thus can frequently only approximate idealised behaviours.

The works [8, 9] can inclusively be used as basis for a deductive metric equational system w.r.t. our programming language. Very briefly, the corresponding metric $d(-,=)$ would be induced by the operator norm in conjunction with the total variation norm (both are detailed in Section 3). Then [8, 9] would lead us on the analysis of how the different program constructs interact with this metric. For example it is immediate from the *op. cit.* that for any programs, p, p' and q, q', the following rule holds:

$$\frac{d([\![\texttt{p}]\!], [\![\texttt{p'}]\!]) \le \epsilon_1 \qquad d([\![\texttt{q}]\!], [\![\texttt{q'}]\!]) \le \epsilon_2}{d([\![\texttt{p ; q}]\!], [\![\texttt{p' ; q'}]\!]) \le \epsilon_1 + \epsilon_2}$$

We leave a full acount of such a metric equational system to future work.

Acknowledgments

This work was financed by National Funds through FCT/MCTES – *Fundação para a Ciência e a Tecnologia, I.P.* (Portuguese Foundation for Science and Technology) within project IBEX, with reference 10.54499/PTDC/CCI-COM/4280/2021. It was further supported by national funds through FCT/MCTES within the CISTER Research Unit (UIDP/UIDB/04234/2020) and under the project Intelligent Systems Associate Laboratory – LASI (LA/P/0104/2020). Finally we are also thankful for the reviewer's helpful feedback.

References

[1] Jiří Adámek, Horst Herrlich, and George E. Strecker. 2009. *Abstract and Concrete Categories - The Joy of Cats.* Dover Publications.

[2] Charalambos D. Aliprantis and Kim C. Border. 2006. *Infinite Dimensional Analysis: a Hitchhiker's Guide.* Springer Berlin, Heidelberg. doi:10.1007/3-540-29587-9

[3] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2013. A survey on reactive programming. *ACM Comput. Surv.* 45, 4 (2013), 52:1–52:34. doi:10.1145/2501654.2501666

[4] Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). 2020. *Foundations of Probabilistic Programming.* Cambridge University Press. doi:10.1017/9781108770750

[5] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2020. Reactive probabilistic programming. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 898–912. doi:10.1145/3385412.3386009

[6] Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 368–392. doi:10.1007/978-3-662-54434-1_14

[7] Fredrik Dahlqvist and Dexter Kozen. 2020. Semantics of higher-order probabilistic programs with conditioning. *Proc. ACM Program. Lang.* 4, POPL (2020), 57:1–57:29. doi:10.1145/3371125

[8] Fredrik Dahlqvist and Renato Neves. 2023. A Complete V-Equational System for Graded lambda-Calculus. In *Proceedings of the 39th Conference on the Mathematical Foundations of Programming Semantics, MFPS XXXIX, Indiana University, Bloomington, IN, USA, June 21-23, 2023 (EPTICS, Vol. 3)*, Marie Kerjean and Paul Blain Levy (Eds.). EpiSciences. doi:10.46298/ENTICS.12299

[9] Fredrik Dahlqvist and Renato Neves. 2023. The syntactic side of autonomous categories enriched over generalised metric spaces. *Log. Methods Comput. Sci.* 19, 4 (2023). doi:10.46298/LMCS-19(4:31)2023

[10] Luc Devroye. 1986. *Non-Uniform Random Variate Generation.* Springer New York, NY. doi:10.1007/978-1-4613-8643-8

[11] R. M. Dudley. 2002. *Real Analysis and Probability* (2 ed.). Cambridge University Press. doi:10.1017/CBO9780511755347

[12] Albert Einstein. 1905. Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen. *Annalen der physik* 4 (1905).

[13] Gerhard Gierz, Karl Heinrich Hofmann, Klaus Keimel, Jimmie D. Lawson, Michael W. Mislove, and Dana S. Scott. 1980. *A compendium of continuous lattices.* Springer Berlin, Heidelberg. xx + 371 pages. doi:10.1007/978-3-642-67678-9

[14] Michèle Giry. 1982. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, B. Banaschewski (Ed.). Lecture Notes in Mathematics, Vol. 915. Springer Berlin Heidelberg, 68–85. doi:10.1007/BFb0092872

[15] Sergey Goncharov, Renato Neves, and José Proença. 2020. Implementing Hybrid Semantics: From Functional to Imperative. In *Theoretical Aspects of Computing - ICTAC 2020 - 17th International Colloquium, Macau, China, November 30 - December 4, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12545)*, Violet Ka I Pun, Volker Stolz, and Adenilso Simão (Eds.). Springer, 262–282. doi:10.1007/978-3-030-64276-1_14

[16] Sergey Goncharov and Lutz Schröder. 2013. A Relatively Complete Generic Hoare Logic for Order-Enriched Effects. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013.* IEEE Computer Society, 273–282. doi:10.1109/LICS.2013.33

[17] Jean Goubault-Larrecq. 2013. *Non-Hausdorff Topology and Domain Theory - Selected Topics in Point-Set Topology.* New Mathematical Monographs, Vol. 22. Cambridge University Press.

[18] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017.* IEEE Computer Society, 1–12. doi:10.1109/LICS.2017.8005137

[19] Wataru Hino, Hiroki Kobayashi, Ichiro Hasuo, and Bart Jacobs. 2016. Healthiness from Duality. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 682–691. doi:10.1145/2933575.2935319

[20] Dirk Hofmann, Gavin J Seal, and Walter Tholen. 2014. *Monoidal Topology: A Categorical Approach to Order, Metric, and Topology.* Vol. 153. Cambridge University Press.

[21] Peter Höfner. 2009. *Algebraic calculi for hybrid systems.* Ph. D. Dissertation. University of Augsburg.

[22] Peter Höfner and Bernhard Möller. 2011. Fixing Zeno gaps. *Theor. Comput. Sci.* 412, 28 (2011), 3303–3322. doi:10.1016/J.TCS.2011.03.018

[23] G. A. Kavvos. 2025. Adequacy for Algebraic Effects Revisited. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025), 927–955. doi:10.1145/3720457

[24] J. Klafter and I.M. Sokolov. 2011. Continuous-time random walks. In *First Steps in Random Walks: From Tools to Applications.* Oxford University Press. doi:10.1093/acprof:oso/9780199234868.003.0003

[25] Dexter Kozen. 1981. Semantics of Probabilistic Programs. *J. Comput. Syst. Sci.* 22, 3 (1981), 328–350. doi:10.1016/0022-0000(81)90036-2

[26] Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. 2020. Towards verified stochastic variational inference for probabilistic programs. *Proc. ACM Program. Lang.* 4, POPL (2020), 16:1–16:33. doi:10.1145/3371084

[27] Paul Blain Levy. 2022. Call-by-push-value. *ACM SIGLOG News* 9, 2 (2022), 7–29. doi:10.1145/3537668.3537670

[28] Saunders Mac Lane. 1998. *Categories for the working mathematician.* Springer New York, NY. doi:10.1007/978-1-4757-4721-8

[29] Pedro Mendes, Ricardo Correia, Renato Neves, and José Proença. 2024. Formal Simulation and Visualisation of Hybrid Programs. In *Proceedings Sixth International Workshop on Formal Methods for Autonomous Systems, FMAS@iFM 2024, Manchester, UK, 11th and 12th of November 2024 (EPTCS, Vol. 411)*, Matt Luckcuck and Mengwei Xu (Eds.). 20–37. doi:10.4204/EPTCS.411.2

[30] Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989.* IEEE Computer Society, 14–23. doi:10.1109/LICS.1989.39155

[31] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. doi:10.1016/0890-5401(91)90052-4

[32] Renato Neves. 2018. *Hybrid programs.* Ph. D. Dissertation. University of Minho. https://repositorium.sdum.uminho.pt/handle/1822/56808

[33] Renato Neves, José Proença, and Juliana Souza. 2025. An Adequate While-Language for Stochastic Hybrid Computation. arXiv:2507.15913 [cs.LO] https://arxiv.org/abs/2507.15913

[34] Prakash Panangaden. 1998. The Category of Markov Kernels. In *First International Workshop on Probabilistic Methods in Verification, PROBMIV 1998, Indianapolis, Indiana, USA, June 19-20, 1998 (Electronic Notes in Theoretical Computer Science, Vol. 22)*, Christel Baier, Michael Huth, Marta Z. Kwiatkowska, and Mark Ryan (Eds.). Elsevier, 171–187. doi:10.1016/S1571-0661(05)80602-4

[35] Prakash Panangaden. 2009. *Labelled Markov Processes.* Imperial College Press. doi:10.1142/p595

[36] I. Pembeci and G. Hager. 2003. Functional reactive programming as a hybrid system framework. In *2003 IEEE International Conference on Robotics and Automation*, Vol. 1. 727–734 vol.1. doi:10.1109/ROBOT.2003.1241680

[37] Yu Peng, Shuling Wang, Naijun Zhan, and Lijun Zhang. 2015. Extending Hybrid CSP with Probability and Stochasticity. In *Dependable Software Engineering: Theories, Tools, and Applications - First International Symposium, SETTA 2015, Nanjing, China, November 4-6, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9409)*, Xuandong Li, Zhiming Liu, and Wang Yi (Eds.). Springer, 87–102. doi:10.1007/978-3-319-25942-0_6

[38] André Platzer. 2011. Stochastic Differential Dynamic Logic for Stochastic Hybrid Programs. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6803)*, Nikolaj S. Bjørner and Viorica Sofronie-Stokkermans (Eds.). Springer, 446–460. doi:10.1007/978-3-642-22438-6_34

[39] André Platzer. 2018. *Logical foundations of cyber-physical systems.* Springer Cham. doi:10.1007/978-3-319-63588-0

[40] John C Reynolds. 1998. *Theories of programming languages.* Cambridge University Press. doi:10.1017/CBO9780511626364

[41] Tetsuya Sato. 2018. The Giry monad is not strong for the canonical symmetric monoidal closed structure on Meas. *Journal of Pure and Applied Algebra* 222, 10 (2018), 2888–2896. doi:10.1016/j.jpaa.2017.11.004

[42] Glynn Winskel. 1993. *The formal semantics of programming languages - an introduction.* MIT Press. doi:10.7551/mitpress/3054.001.0001

[43] Naijun Zhan, Shuling Wang, and Hengjun Zhao. 2016. *Formal Verification of Simulink/Stateflow Diagrams, A Deductive Approach.* Springer Cham. doi:10.1007/978-3-319-47016-0