Informal Proceedings of the 11th International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2025)

Birmingham, UK, July 20, 2025

Preface

The 11th International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2025) was held on July 20, 2025, in Birmingham, UK, as part of FSCD 2025. WPTE brings together researchers working on program transformations, evaluation, and operationally based programming language semantics, using rewriting methods. This volume collects the extended abstracts of the contributions accepted for presentation. We thank all authors for their submissions and the program committee and reviewers for their feedback. We hope the contributions will foster further research and collaboration in the rewriting community.

Invited Talks

- **Transformations of Macro Tree Programs**
  by *Sebastian Maneth*
- **Generating feedback from rewrite strategies for interactive exercises**
  by *Alex Gerdes*

Program Committee

- Martin Avanzini, Inria Sophia Antipolis
- Patrick Bahr, IT University of Copenhagen
- Demis Ballis, University of Udine
- Mirai Ikebuchi, Kyoto University
- Cynthia Kop (co-chair), Radboud University Nijmegen
- Ivan Lanese, University of Bologna
- Pierre Lermusiaux, Inria Rennes
- Steven Libby, University of Portland
- Luca Roversi, University of Turin
- Janis Voigtländer (chair), University of Duisburg-Essen
- Johannes Waldmann, Leipzig University of Applied Sciences
- Sarah Winkler, Free University of Bozen-Bolzano

Contents

Refactoring Communication Protocols for Crash Safety

Leonid Nosovitskiy
School of Computer Science
University of St Andrews
St Andrews, UK
ln60@st-andrews.ac.uk

Adam D. Barwell
School of Computer Science
University of St Andrews
St Andrews, UK
adb23@st-andrews.ac.uk

Development of concurrent and distributed systems is notoriously difficult owing to the many subtle and difficult-to-debug errors that may occur. Session types address the issues arising from communicating processes, and failures thereof, by enabling the specification and verification of communication protocols, providing *correct-by-construction* behavioural guarantees, e.g. communication safety and deadlock-freedom. Implementations of session type theories commonly combine concurrency libraries with code generation tooling, enabling automatic production of protocol-conforming code from high-level protocol specifications. Despite the proliferation of tooling for a range of programming languages, most session type approaches assume that protocols are specified once. This contrasts real-world development, where bugs are found and fixed, and new features are developed over time. Presently, such code transformations, or *refactorings*, over session types are performed *manually*. In this paper, we explore (work-in-progress) refactoring tooling for multiparty session types that facilitates the introduction of crash-handling behaviour in protocols. We introduce, *BigT*, a code generation and refactoring toolchain, that semi-automatically transforms protocol specifications and the associated generated code, whilst preserving behavioural safety properties. In the full paper, we will evaluate our approach on a range of examples from both the session types and the distributed systems literature.

1 Introduction

Distributed systems are ubiquitous in modern society. They underpin key infrastructure, including finance, online banking, and telecommunication networks. However, developing such systems is notoriously difficult, since programmers must account for a broad spectrum of potential issues, including deadlocks and communication mismatches, in spite of inevitable failures. Multiparty Session Types (MPST) [15] are a formal framework for specifying communication protocols that statically guarantee desirable behavioural properties, including communication safety, deadlock-freedom, and liveness.

MPST libraries are available for a wide range of programming languages [14], and many MPST toolchains, e.g. SCRIBBLE [16], enable automatic generation of protocol-conforming code skeletons. Generated code skeletons represent the communications behaviour of the protocol, and are specialised by the programmer with appropriate business logic. The TEATRINO toolchain [3] extends SCRIBBLE with support for crash-stop failures, where unreliable processes can arbitrarily crash (i.e. stop communicating) and do not recover. TEATRINO's code generation targets SCALA, leveraging its expressive type system to define session types directly in code. TEATRINO is thus capable of generating *fault-tolerant*, protocol-conforming, concurrent SCALA code.

Although these toolchains abstract over low-level implementation details when specifying protocols, it remains that the programmer must nevertheless specify the protocol to be used. The introduction of arbitrary process crashes complicates protocol specifications since *all* interactions with unreliable processes must specify *crash-handling behaviour*. Introducing crash-handling behaviour to a core protocol,

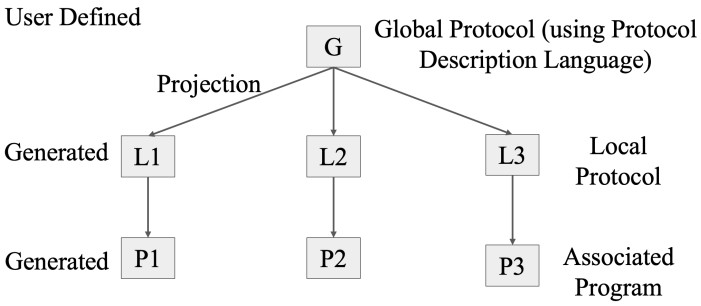


Figure 1: Top-Down Approach of MPST

where all participants are assumed to be reliable, whilst preserving the protocol's behavioural properties, presents a challenge that scales with the size of the core protocol and the number of participants to be made unreliable. Consequently, even simple benchmark protocols can grow significantly [4]. Since introducing crash-handling behaviour is currently a manual process, it is both laborious and error-prone, and so hinders adoption of MPST tooling despite their advantages. This is further compounded by the implicit assumption that the given protocol is static and code is generated only once. Consequently, should the protocol be updated, new code must be generated, and manually integrated into the program.

Refactoring refers to the act of changing the internal structure of a program without changing its functional behaviour [6]. We adopt a broader definition of refactoring — one that allows for differences from full functional equivalence, while maintain the overall intent or semantics of the protocol [2]. Whilst refactorings can be applied manually, tooling can provide a semi-automatic approach that reduces the opportunity for error and ensures that code is transformed safely. A range of refactoring tools have been developed for a number of programming languages and IDEs [1].

In this paper, we introduce BIGT, a novel extension of TEATRINO that facilitates the refactoring of protocol specifications to semi-automatically introduce crash-handling behaviour. Moreover, our approach enables the simultaneous refactoring of both protocol specifications and any associated code generated from protocols, ensuring that implementation and specification do not diverge. To that end, we present a novel refactoring for MPST, *Introduce Unreliability*, that introduces crash-handling behaviour for a selected participant of a protocol. This refactoring represents the conversion of a *reliable* participant to an *unreliable* participant, thereby introducing an assumption of unreliability. In the first instance, we focus on non-recursive protocols and on terminating the protocol gracefully in the event of a crash. To facilitate the definition of *Introduce Unreliability*, and expand the range of protocols to which it can be applied, we additionally present *Converge Protocol* (see section 4). In the full version of this paper, we will give the full definitions of our refactorings, and their implementations in BIGT. We will extend *Introduce Unreliability* to both recursive protocols and other crash-handling behaviours. Finally, we will evaluate our toolchain on examples from both the MPST and distributed systems literature.

2 Multiparty Session Types

Multiparty Session Types (MPST) facilitate the specification and verification of communication protocols [15]. In this paper, we extend the TEATRINO toolchain [3], and thus assume the top-down asynchronous MPST theory by Barwell *et al.* [5], which extends classical MPST [11] approaches by support-

$$
\begin{array}{rcll}
G & ::= & \mathsf{p}\to\mathsf{q}:\{\mathsf{m}_i(B_i).G_i\}_{i\in I} & \text{Transmission} \\
 & \mid & \mu\mathbf{t}.G & \text{Recursion} \\
 & \mid & \mathbf{t} & \text{Type variable} \\
 & \mid & \mathsf{end} & \text{Termination} \\
S,T & ::= & \mathsf{p}\&\{\mathsf{m}_i(B_i).T_i\}_{i\in I} \mid \mathsf{p}\oplus\{\mathsf{m}_i(B_i).T_i\}_{i\in I} & \text{External choice (receive), Internal choice (send)} \\
 & \mid & \mu\mathbf{t}.T \mid \mathbf{t} \mid \mathsf{end} & \text{Recursion, Type variable, Termination}
\end{array}
$$

Figure 2: Syntax of global and local types [3]. We elide runtime types and annotations.

ing *crash-stop failures* [7], i.e. where processes can crash arbitrarily and do not recover. Their theory is *top-down* in the sense that it enables code generation with correct-by-construction guarantees of communication safety, deadlock-freedom, and liveness. Fig. 1 gives an overview of the top-down approach. Further details of syntax and semantics can be found in the paper originally introducing TEATRINO [3] and full details in the journal version [5]. In this section, we introduce MPST and the top-down approach via a pair of example protocols.

Global Types We first consider a simple Hierarchical Authenticator that controls access to both public and secret resources, where *public* and *secret* are security levels inspired by information flow techniques [8]. Our protocol has three *participants*: *i)* a u ser, who requests access to a resource; *ii)* an authenticator, which can grant access to public resources; and *iii)* a guard, which can grant access to secret resources. In the top-down approach, we first define the hierarchical authenticator protocol using *global types*, which presents a bird's-eye view of the communications between participants. We give the global type for G_1, below.

$$
G_1 = \mathsf{u}\to\mathsf{a}:\mathsf{request}(\mathsf{Cred}).\mathsf{a}\to\mathsf{g}:\begin{cases}\mathsf{log}(\mathsf{Cred}).\mathsf{a}\to\mathsf{u}:\mathsf{validate}(\mathsf{Cred}).\mathsf{end} \\ \mathsf{delegate}(\mathsf{Cred}).\mathsf{a}\to\mathsf{u}:\mathsf{wait}.\mathsf{g}\to\mathsf{u}:\begin{cases}\mathsf{validate}(\mathsf{Cred}).\mathsf{end} \\ \mathsf{deny}.\mathsf{end}\end{cases}\end{cases}
$$

The protocol begins with u sending a request to a in order to be granted access to a resource. Here, request is a label used to distinguish between messages, and Cred is a payload type used to define the data being transmitted. In our example, Cred is the type of credential objects [9]. Insignificant payload types are omitted, e.g. when a sends wait. If granting access to the requested resource is within the gift of a, a first logs the validation with g, then responds to u with the validated credential object. Alternatively, a delegates the request to g. It then instructs u to await authorisation from g, who either validates the credential or denies access to the secret resource, ending the protocol.

In a distributed setting, any of the Hierarchical Authenticator participants may fail. TEATRINO facilitates the modelling of failures via *optional reliability assumptions*, where participants are assumed either *reliable* (i.e. never crash) or *unreliable* (i.e. can crash arbitrarily). Reliability assumptions are expressed by the set of reliable participants $\mathscr{R} \subseteq \mathfrak{R}$, where $\mathfrak{R}$ is the set of all participants in a given global type. Crash detection is modelled on receiving participants, thus all interactions where the sender is unreliable must include a *crash-handling branch*. Crash-handling branches are identified by the reserved label crash. In our Hierarchical Authenticator example, assuming $\mathscr{R} = \{\mathsf{a},\mathsf{g}\}$, G_1 must be extended with a

crash-handling branch when receiving from u:

$$G'_1 = \mathtt{u}\to\mathtt{a}:\begin{cases}\mathsf{request}(\mathsf{Cred}).\mathtt{a}\to\mathtt{g}:\begin{cases}\mathsf{log}(\mathsf{Cred}).\mathtt{a}\to\mathtt{u}:\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{delegate}(\mathsf{Cred}).\mathtt{a}\to\mathtt{u}:\mathsf{wait}.\mathtt{g}\to\mathtt{u}:\begin{cases}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{deny}.\mathsf{end}\end{cases}\end{cases}\\ \mathsf{crash}.\mathtt{a}\to\mathtt{g}:\mathsf{abort}.\mathsf{end}\end{cases}$$

Here, when a awaits a request from u it may instead detect that u has crashed. Since crash-handling branches are required only on interactions where the sender is unreliable, G'_1 does not require crash-handling branches on interactions between a and g and where either a or g send messages to u.

Local Types Within the top-down approach, *local types* are automatically *projected* from global types and represent the protocol from a given participant's perspective. The projection of G'_1 onto u, denoted $G'_1 \upharpoonright \mathtt{u}$, is:

$$G'_1 \upharpoonright \mathtt{u} = \mathtt{a}\oplus\mathsf{request}(\mathsf{Cred}).\mathtt{a}\&\begin{cases}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{wait}.\mathtt{g}\&\begin{cases}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{deny}.\mathsf{end}\end{cases}\end{cases}$$

Here, u first sends (denoted $\oplus$) its request to a. It then receives (denoted $\&$) either the validated credential object, or an indication to await validation from g. In the latter case, u subsequently receives either the validated credential object or an indication that the request has been denied, ending the protocol. Since crash-handling branches denote crash detection of the sender by the receiver, crash labels are never sent by u in $G'_1 \upharpoonright \mathtt{u}$. Conversely, projecting on a results in a crash-handling branch when receiving from u:

$$G'_1 \upharpoonright \mathtt{a} = \mathtt{u}\&\begin{cases}\mathsf{request}(\mathsf{Cred}).\mathtt{g}\oplus\begin{cases}\mathsf{log}(\mathsf{Cred}).\mathtt{u}\oplus\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{delegate}(\mathsf{Cred}).\mathtt{u}\oplus\mathsf{wait}.\mathsf{end}\end{cases}\\ \mathsf{crash}.\mathtt{g}\oplus\mathsf{abort}.\mathsf{end}\end{cases}$$

Type-Checked Processes The final stage of the top-down approach uses local types to type-check implementations of the protocol. In this paper, we focus on the Scala implementation over the process calculus formulation [3]. This approach exploits and extends the Effpi concurrency library [13] to directly represent local types in code, which are then used to type (and generate) concomitant implementations of processes acting as the given participant. Effpi presents an Actor-based API with *processes* that communicate via *channels*. The local type $G'_1 \upharpoonright \mathtt{u}$ is represented in Effpi in the form:

```
type U[C0 <: OutChannel[Request], C3 <: InChannel[Validate],
       C5 <: InChannel[Deny | Validate], C7 <: InChannel[Wait | Validate]] =
  Out[C0, Request]
    >>: In[C7, Wait | Validate, (x0 : Wait | Validate) => U0[x0.type, C3, C5]]
```

The code snippet defines a local session type U that specifies a process which first sends a Request on an output channel C0 (Out[C0, Request]), then waits to receive a message of either type Wait or Validate on an input channel C7 (In[C7, Wait Validate, ...]|). The sequencing operator »: links these two actions, indicating that the receive action follows the send.
The continuation after receiving the message is a function that takes the received value x0 and continues according to the type U0[x0.type, C3, C5], which varies depending on the exact message

received. This allows the system to use conditional behavior based on runtime values, while still preserving compile-time guarantees. Each channel is typed to be either `OutChannel` or `InChannel` which means that they can not be used interchangeably.

Two-Seller In addition to our Hierarchical Authenticator, we consider a simple Two-Seller protocol in which a buyer chooses to buy from one of two sellers, s1 and s2. Both sellers use the same courier.

$$G_2 = \mathtt{b}\to\mathtt{s1}:\begin{cases}\mathsf{buy}.\mathtt{b}\to\mathtt{s2}:\mathsf{reject}.\mathtt{s1}\to\mathtt{c}:\mathsf{request}.\mathtt{s2}\to\mathtt{c}:\mathsf{cancel}.\mathsf{end}\\ \mathsf{reject}.\mathtt{b}\to\mathtt{s2}:\mathsf{buy}.\mathtt{s1}\to\mathtt{c}:\mathsf{cancel}.\mathtt{s2}\to\mathtt{c}:\mathsf{request}.\mathsf{end}\end{cases}$$

Whereas the Hierarchical Authenticator protocol demonstrates a high degree of interaction between all three processes, Two-Seller demonstrates how a single interaction can determine the communication behaviour of its peers. The respective features of both protocols are useful for illustrating important aspects of *Introduce Unreliability* and *Converge Protocol*.

3 Protocol-Level Refactoring for Introducing Unreliability Assumptions

In this section, we present a novel refactoring, *Introduce Unreliability*, for global types that converts a reliable participant p into an unreliable participant. This is principally achieved by the introduction of crash-handling branches to interactions where p is the sender. We initially take a *best-effort* approach to failure-handling in that introduced crash-handling branches are designed to delay termination of the protocol for as long as possible. We take this approach first in order to maintain best effort protocol that need to keep going even when some participants crash. Additionally, this refactoring acts as a prerequisite for fail-gracefully refactoring, which we explore in the full version of the paper along with other patterns.

Given a global type G that is projectable on all its participants $\mathfrak{R}$, the set of reliable participants $\mathscr{R}$, and a reliable role $\mathtt{p}\in\mathscr{R}$, *Introduce Unreliability* refactors G, producing $G_{\not\downarrow\mathtt{p}}$, denoted $G \rightsquigarrow_{\mathscr{R}\setminus\mathtt{p}} G_{\not\downarrow\mathtt{p}}$. Projectability refers to the ability to correctly construct each participant's local type from a global protocol, ensuring that the local types preserve all safety properties—full definitions can be found in [3]. To illustrate our approach, we apply *Introduce Unreliability* to the Two-Seller protocol from Section 2, making only b unreliable.

$$G_2 \rightsquigarrow_{\mathscr{R}\setminus\mathtt{b}} G_2' = \mathtt{b}\to\mathtt{s1}:\begin{cases}\mathsf{buy}.\mathtt{b}\to\mathtt{s2}:\begin{cases}\mathsf{reject}.\mathtt{s1}\to\mathtt{c}:\mathsf{request}.\mathtt{s2}\to\mathtt{c}:\mathsf{cancel}.\mathsf{end}\\ \mathsf{crash}.\mathtt{s1}\to\mathtt{c}:\mathsf{request}.\mathtt{s2}\to\mathtt{c}:\mathsf{cancel}.\mathsf{end}\end{cases}\\ \mathsf{reject}.\mathtt{b}\to\mathtt{s2}:\begin{cases}\mathsf{buy}.\mathtt{s1}\to\mathtt{c}:\mathsf{cancel}.\mathtt{s2}\to\mathtt{c}:\mathsf{request}.\mathsf{end}\\ \mathsf{crash}.\mathtt{s1}\to\mathtt{c}:\mathsf{cancel}.\mathtt{s2}\to\mathtt{c}:\mathsf{cancel}.\mathsf{end}\end{cases}\\ \mathsf{crash}.\mathtt{b}\to\mathtt{s2}:\begin{cases}\mathsf{buy}.\mathtt{s1}\to\mathtt{c}:\mathsf{cancel}.\mathtt{s2}\to\mathtt{c}:\mathsf{request}.\mathsf{end}\\ \mathsf{crash}.\mathtt{s1}\to\mathtt{c}:\mathsf{cancel}.\mathtt{s2}\to\mathtt{c}:\mathsf{cancel}.\mathsf{end}\end{cases}\end{cases}$$

Here, *Introduce Unreliability* introduces two crash-handling branches to G_2. The first interaction between b and s1 is extended with a crash-handling branch whose continuation is taken from the reject branch. The selection of which branch to copy is made by the programmer. The interactions between b and s2 are similarly extended with a crash-handling branch. In each case, the continuation is taken from its peer (i.e. reject or buy).

We remark that G_2' may not reflect the final state of the intended fault-tolerant protocol: further participants may be made unreliable and labels may be changed in the crash-handling branches to better reflect the desired behaviour in the event of a crash. *Introduce Unreliability* is designed to ensure that any introduced crash-handling branch *preserves projectability* of the protocol, thereby ensuring that the resulting protocol retains communication safety, deadlock-freedom, and liveness properties.

In full paper, we will define *Introduce Unreliability* formally, and give an intuition of our approach here. *Introduce Unreliability* proceeds according to two cases: *merged communications* and *stand-alone communications*.

Stand-Alone Communications In cases where a crash-handling branch is introduced to an interaction that is not merged with other interactions in the global type, the naïve approach suffices. The interested reader can find the full definition of merging, with examples, in the original paper [3]. The naïve approach refers to copying a peer's continuation. For example, the interaction between b and s1 that begins the protocol is not merged with any other interaction in G_2. Accordingly, the continuation for the crash-handling branch can be taken from either buy or reject branches. This produces the partially refactored, non-projectable protocol G_2'':

$$G_2'' = \mathtt{b}\to\mathtt{s1}:\begin{cases}\mathsf{buy}.\mathtt{b}\to\mathtt{s2}:\mathsf{reject}.\mathtt{s1}\to\mathtt{c}:\mathsf{request}.\mathtt{s2}\to\mathtt{c}:\mathsf{cancel}.\mathsf{end}\\ \mathsf{reject}.\mathtt{b}\to\mathtt{s2}:\mathsf{buy}.\mathtt{s1}\to\mathtt{c}:\mathsf{cancel}.\mathtt{s2}\to\mathtt{c}:\mathsf{request}.\mathsf{end}\\ \mathsf{crash}.\mathtt{b}\to\mathtt{s2}:\mathsf{buy}.\mathtt{s1}\to\mathtt{c}:\mathsf{cancel}.\mathtt{s2}\to\mathtt{c}:\mathsf{request}.\mathsf{end}\end{cases}$$

G_2'' is not projectable because interactions between b and s2 are not yet refactored; however, if b is assumed to be reliable, the protocol would be projectable. The refactoring then proceeds in order to introduce all relevant crash-handling branches.

Merged Communications When projecting G_2 on s2, two ostensibly distinct interactions in G_2 (i.e. b→s2:reject and b→s2:buy) are *merged* to form a single reception from b:

$$G_2 \upharpoonright \mathtt{s2} = \mathtt{b}\&\begin{cases}\mathsf{reject}.\mathtt{c}\oplus\mathsf{cancel}.\mathsf{end}\\ \mathsf{buy}.\mathtt{c}\oplus\mathsf{request}.\mathsf{end}\end{cases}$$

This merging of two interactions in the global type occurs because s2 is not party to the preceding interaction between b and s1 and is thus unaware of the message sent by b.

The consequence of this merging is that, when introducing crash-handling branches to interactions in the global type, it does not suffice to consider only the interaction to which the crash-handling branch is being introduced. In our Two-Seller example, we must consider b→s2:reject when introducing a crash-handling branch to b→s2:buy and vice versa. Here, the naïve copying of a peer's continuation *does not preserve projection*, as it introduces a choice in the crash-handling branch:

$$G_2 \upharpoonright \mathtt{s2} = \mathtt{b}\&\begin{cases}\mathsf{buy}.\mathtt{c}\oplus\mathsf{request}.\mathsf{end}\\ \mathsf{reject}.\mathtt{c}\oplus\mathsf{cancel}.\mathsf{end}\\ \mathsf{crash}.\mathtt{c}\oplus\begin{cases}\mathsf{request}.\mathsf{end}\\ \mathsf{cancel}.\mathsf{end}\end{cases}\end{cases}$$

which can result in deadlock. For example, when s1 sends a request to c, c expects a cancel from s2. A deadlock will occur in the event that s2 chooses to send request to c. In order to preserve projectability,

$$G_1'' = \mathtt{u}\to\mathtt{a}{:}\mathsf{request}(\mathsf{Cred}).\mathtt{a}\to\mathtt{g}:\begin{cases}\mathsf{log}(\mathsf{Cred}).\mathtt{a}\to\mathtt{u}:\begin{cases}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{crash}.\mathtt{g}\to\mathtt{u}:\begin{cases}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{deny}.\mathsf{end}\end{cases}\end{cases}\\ \mathsf{delegate}(\mathsf{Cred}).\mathtt{a}\to\mathtt{u}:\begin{cases}\mathsf{wait}.\mathtt{g}\to\mathtt{u}:\begin{cases}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{deny}.\mathsf{end}\end{cases}\\ \mathsf{crash}.\mathtt{g}\to\mathtt{u}:\begin{cases}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{deny}.\mathsf{end}\end{cases}\end{cases}\\ \mathsf{crash}.\mathtt{a}\to\mathtt{u}:\begin{cases}\mathsf{wait}.\mathtt{g}\to\mathtt{u}:\begin{cases}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{deny}.\mathsf{end}\end{cases}\\ \mathsf{crash}.\mathtt{g}\to\mathtt{u}:\begin{cases}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{deny}.\mathsf{end}\end{cases}\end{cases}\end{cases}$$

Figure 3: Naïve and incorrect transformation of G_1, making a unreliable.

we therefore select one of the possible crash-handling branches for both interactions. Here, we select the reject continuation:

$$G_2 \upharpoonright \mathtt{s2} = \mathtt{b}\&\begin{cases}\mathsf{buy}.\mathtt{c}\oplus\mathsf{request}.\mathsf{end}\\ \mathsf{reject}.\mathtt{c}\oplus\mathsf{cancel}.\mathsf{end}\\ \mathsf{crash}.\mathtt{c}\oplus\mathsf{cancel}.\mathsf{end}\end{cases}$$

4 Protocol-Level Refactoring for Enabling Unreliability

As seen in Section 3, some interactions in a global type are associated via *merging* when projecting on a given participant. In our Two-Seller protocol, the associated interactions occur in *all* peer branches. However, projection does not preclude protocols where different branches may include different numbers of subsequent interactions. An example of this can be found in our Hierarchical Authenticator example, where g in Section 2 only communicates with u when a delegates a request. Protocols that *diverge* in this manner can render it impossible to introduce unreliability assumptions to one or more participants. For example, consider the naïvely transformed global type G_1'' in Fig. 3, a is made unreliable in G_1. This transformation does not preserve projectability since a deadlock can now occur. Specifically, a deadlock occurs when a crashes *after* it has logged the request with g, but before a sends the validated credential: u expects a message from g, but g has ended.

Whilst we define *Introduce Unreliability* to fail when it detects such diverging protocols, said diverging protocols present a separate refactoring opportunity. For example, by first transforming the communication of G_1 to include an additional interaction between g and u:

$$G_1^3 = \mathtt{u}\to\mathtt{a}{:}\mathsf{request}(\mathsf{Cred}).\mathtt{a}\to\mathtt{g}:\begin{cases}\mathsf{log}(\mathsf{Cred}).\mathtt{a}\to\mathtt{u}{:}\mathsf{validate}(\mathsf{Cred}).\mathtt{g}\to\mathtt{u}{:}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{delegate}(\mathsf{Cred}).\mathtt{a}\to\mathtt{u}{:}\mathsf{wait}.\mathtt{g}\to\mathtt{u}:\begin{cases}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{deny}.\mathsf{end}\end{cases}\end{cases}$$

If granting access to the requested resource is within the gift of a, g now sends a copy of the validated credential to u that grants access to the requested public resource. In practice, this may be viewed as an additional step in order to ensure correctness of the validation.

```
1 global protocol PingPong(reliable role P, reliable role Q) {
2     ping(T) from P to Q;
3     pong(T) from Q to P;
4 }
```

Figure 4: Ping-Pong protocol in SCRIBBLE

```
1 def p(c0:OutChannel[Ping],c1:InChannel[Pong]):P[c0.type, c1.type]={
2   send(c0, new Ping(new T())) >> {
3     receive(c1) {(x0 : Pong) =>
4       nil
5     }
6   }
7 }
```

Figure 5: Code Skeleton

Accordingly, we propose a second refactoring *Converge Protocol*, for removing such divergence in protocols. We will give the full definition of *Converge Protocol* in the full paper. Whereas we conjecture that protocols produced by applications of *Introduce Unreliability* may *simulate* [12] their original protocols, the same cannot be said of protocols refactored via *Converge Protocol*. For example, the programmer could instead refactor G_1 such that g does not communicate with u, as in OAuth2 [10].

$$\mathtt{u}\to\mathtt{a}{:}\mathsf{request}(\mathsf{Cred}).\mathtt{a}\to\mathtt{g}{:}\begin{cases}\mathsf{log}(\mathsf{Cred}).\mathtt{g}\to\mathtt{a}{:}\mathsf{commit}.\mathtt{a}\to\mathtt{u}{:}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\\ \mathsf{delegate}.\mathtt{g}\to\mathtt{a}{:}\mathsf{validate}(\mathsf{Cred}).\mathtt{a}\to\mathtt{u}{:}\mathsf{validate}(\mathsf{Cred}).\mathsf{end}\end{cases}$$

Since such a refactoring represents a potentially significant change to a protocol's communication behaviour, the programmer must determine whether such a change is acceptable.

5 Refactorings for Code

In this section, we illustrate extensions to EFFPI and the SCRIBBLE protocol description language that facilitate refactoring at the code-level. These extensions are designed to enable transformations to already generated code *in situ*, when applying *Introduce Unreliability* and *Converge Protocol* to global types.

The TEATRINO dialect of the SCRIBBLE protocol description language permits a programmatic representation of global types in order to enable code generation. For example, a simple Ping-Pong protocol that has the global type, $G_3 = \mathtt{p}\to\mathtt{q}{:}\mathsf{ping}(\mathsf{T}).\mathtt{q}\to\mathtt{p}{:}\mathsf{pong}(\mathsf{T}).\mathsf{end}$ can be represented in SCRIBBLE via the protocol definition in Fig. 4. Participants p and q are declared as being `reliable`. In generating SCALA code skeletons, TEATRINO projects the protocol onto all participants, obtaining a set of local types, and thence derives EFFPI representations and implementations of the participants, where each participant is played by one process. Fig. 5 gives the skeleton implementation that conforms to $G_3 \upharpoonright \mathtt{p}$: it constructs a SCALA case class `new T()` representing payload type T, sends ping on Line 2 via the `send` construct, receives pong from q on Line 3 via the `receive` construct, and then terminates on Line 4 via the `nil` construct.

Although generated code conforms to the given protocol, the programmer must subsequently extend the skeleton with business logic, e.g. specific values to be sent and concrete decision procedures by which

```
1 global protocol PingPong(reliable role P, reliable role Q) {
2     @SendingName firstSend
3     @ReceivingName consumePing
4     ping(T) from P to Q;
5     @GeneralName pongInteraction
6     pong(T) from Q to P;
7 }
```

Figure 6: Annotated PingPong protocol in Scribble

```
1  abstract class PingPong {
2      ...
3      def firstSend(state:pStateType):Ping
4      def pongInteractionReceiving(message:Pong,state:pStateType):Unit
5      ...
6
7      def p(ch0:OutChannel[Ping],ch1:InChannel[Pong]):P[ch0.type,ch1.type]={
8          send(ch0, firstSend(pState)) >> {
9              receive(ch1) {(x0 : Pong) =>
10                 pongInteractionReceiving(x0,pState)
11                 nil
12             }
13         }
14     }
15     ...
16 }
```

Figure 7: Abstracted Code Skeleton

internal choices are made, in order to integrate the generated code into both existing and novel code. This requires a level of expertise and familiarity with both session types and EFFPI from the programmer. A consequence of which is a lack of clarity in terms of what should be modified, and what can be modified without modifying the communication structure, particularly as protocols scale in size. For example, in Fig. 5, the programmer should introduce code between Lines 1 and 2 to prepare the payload, and between Lines 3 and 4 to process the received message. Moreover, since any modification to the protocol and code regeneration will produce new code, any extensions implemented by the programmer must be manually re-integrated in the newly generated code skeleton. As before the difficulty of this task scales with the size of the protocols.

In order to address these issues, we extend SCRIBBLE syntax with annotations for interactions, enabling the programmer to provide function names that are propagated to the generated code. We illustrate this in Fig. 6, introducing (function) name annotations. On Line 2, `p` is defined to use a function named `firstSend`, whereas on Line 3, `q` is defined to use a function `consumePing` for process the received value. For additional flexibility, name annotations permit partial function names, e.g. on Line 5 of Fig. 6, where this function name will be appended with a suffix of `Sending` or `Receiving`, according to the participant's role in the interaction.

When generating code, BIGT now creates an abstract class (e.g. Fig. 7) in which the corresponding abstract functions are declared. Their types are determined by the transmission type (input/output) and the involved message labels—for example, as shown in Lines 3–4. This indicates to the programmer

```
class ConcretePingPong extends PingPong {
    ...
    override def firstSend(state:pStateType):Ping={
        return new Ping(new T("some","args"))
    }
    override def pongInteractionReceiving(message:Pong,state:pStateType):Unit={
        println("Some custom behaviour")
    }
    ...
}
```

Figure 8: Concrete PingPong Implementation

that definitions need to be provided, with additional guidance from the compiler regarding any missing or extraneous functions. Novel implementations can be provided via concrete classes that extend the abstract declarations, e.g. as in Fig. 8. Alternatively, existing functions can be provided, with potentially minor tool-driven modifications to ensure proper adherence to the generated EFFPI interface.

Moreover, this approach facilitates code-level refactoring. A refactoring applied to the global type preserves programmer-given annotations. When introducing new interactions, as in *Introduce Unreliability*, the programmer can provide a name for the corresponding introduced function. Consequently, when skeleton code is generated as a result of the refactoring, programmer-provided code is maintained and easily adapted for the updated protocol.

In the full version of this paper, we will present full definitions and implementations of our code-level refactorings, and their integration with *Introduce Unreliability* and *Converge Protocol*.

References

[1] Adam D. Barwell (2018): *Pattern discovery for parallelism in functional languages*. Ph.D. thesis, University of St Andrews, UK.

[2] Adam D. Barwell, Christopher Brown, Mun See Chang, Constantine Theocharis & Simon Thompson (2025): *Structural Refactorings for Exploring Dependently Typed Programming*. In Jason Hemann & Stephen Chang, editors: *Trends in Functional Programming*, Springer Nature Switzerland, Cham, pp. 1–21.

[3] Adam D. Barwell, Ping Hou, Nobuko Yoshida & Fangyi Zhou (2023): *Designing Asynchronous Multiparty Protocols with Crash-Stop Failures*. In Karim Ali & Guido Salvaneschi, editors: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 263, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 1:1–1:30, doi:10.4230/LIPIcs.ECOOP.2023.1. Available at `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.1`.

[4] Adam D. Barwell, Ping Hou, Nobuko Yoshida & Fangyi Zhou (2023): *Designing Asynchronous Multiparty Protocols with Crash-Stop Failures*. *CoRR* abs/2305.06238.

[5] Adam D. Barwell, Ping Hou, Nobuko Yoshida & Fangyi Zhou (2025): *Crash-Stop Failures in Asynchronous Multiparty Session Types*. *Logical Methods in Computer Science* Volume 21, Issue 2:5, doi:10.46298/lmcs-21(2:5)2025. Available at `https://lmcs.episciences.org/12622`.

[6] Christopher Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick & Archibald Elliott (2014): *Cost-Directed Refactoring for Parallel Erlang Programs*. *Int. J. Parallel Program.* 42(4), pp. 564–582.

[7] Christian Cachin, Rachid Guerraoui & Luís E. T. Rodrigues (2011): *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer.

[8] Dorothy E. Denning (1976): *A Lattice Model of Secure Information Flow*. *Commun. ACM* 19(5), pp. 236–243.

[9] Sufian Hameed, Lamak Qaizar & Shankar Khatri (2017): *Efficacy of Object-Based Passwords for User Authentication*, doi:10.48550/arXiv.1711.11303.

[10] Dick Hardt (2012): *The OAuth 2.0 Authorization Framework*. RFC 6749, doi:10.17487/RFC6749. Available at `https://www.rfc-editor.org/info/rfc6749`.

[11] 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 & Gianluigi Zavattaro (2016): *Foundations of Session Types and Behavioural Contracts*. *ACM Comput. Surv.* 49(1), doi:10.1145/2873052. Available at `https://doi.org/10.1145/2873052`.

[12] Davide Sangiorgi (2011): *Introduction to Bisimulation and Coinduction*. Cambridge University Press, USA.

[13] Alceste Scalas, Nobuko Yoshida & Elias Benussi (2019): *Effpi: verified message-passing programs in Dotty*. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, Scala '19, Association for Computing Machinery, New York, NY, USA, p. 27–31, doi:10.1145/3337932.3338812. Available at `https://doi.org/10.1145/3337932.3338812`.

[14] Nobuko Yoshida (2024): *Programming Language Implementations with Multiparty Session Types*. In: *Active Object Languages: Current Research Trends*, Springer Nature Switzerland, pp. 147–165, doi:10.1007/978-3-031-51060-1_6.

[15] Nobuko Yoshida & Lorenzo Gheri (2020): *A Very Gentle Introduction to Multiparty Session Types*. In: *16th International Conference on Distributed Computing and Internet Technology*, *LNCS* 11969, Springer, pp. 73–93.

[16] Nobuko Yoshida, Raymond Hu, Rumyana Neykova & Nicholas Ng (2014): *The Scribble Protocol Language*. In Martín Abadi & Alberto Lluch Lafuente, editors: *Trustworthy Global Computing*, Springer International Publishing, Cham, pp. 22–41. Available at `https://doi.org/10.1007/978-3-319-05119-2_3`.

Proving Termination of Scala Programs by Constrained Term Rewriting

Dragana Milovančević
EPFL, Station 14
CH-1015 Lausanne, Switzerland
dragana.milovancevic@epfl.ch

Carsten Fuhs
Birkbeck, University of London
United Kingdom
c.fuhs@bbk.ac.uk

Viktor Kunčak
EPFL, Station 14
CH-1015 Lausanne, Switzerland
viktor.kuncak@epfl.ch

Techniques and tools for program verification often require a termination proof as an ingredient of correctness proofs for other program properties (e.g., program equivalence). However, the range of termination proof techniques is vast, so implementing them for every single programming language anew would not be an efficient use of time for tool developers.

In this extended abstract, we propose an indirect route for proving termination of Scala programs, via a translation to constrained term rewriting systems so that termination of the rewrite system implies termination of the Scala program. This allows for separating the language-specific and the language-independent aspects of the analysis into different tools. We have implemented our approach in the Stainless program verifier. Preliminary experiments with the termination prover AProVE indicate that this approach can lead to an increased termination success rate compared to existing approaches.

1 Introduction

One of the tasks for program analysis tools is to prove termination of their input programs. This task matters both on its own and also as a precondition for proof techniques for other properties, such as program equivalence [20] and properties expressed using multiple user-defined function invocations. While implementing language-specific proof techniques is an option, it is arduous and ignores opportunities for reuse of existing infrastructure: over the last decades, techniques and tools for push-button termination analysis of different computational formalisms have matured, particularly so for various flavors of *Term Rewriting Systems (TRSs)*. This suggests a two-stage approach to termination analysis by program analysis tools with a separation of concerns: (1) extract a TRS from the input program such that termination of the TRS implies termination of the input program, and (2) call an external termination tool for TRSs to get a termination proof (if one can be found).

Similar two-stage approaches for analysis of properties via term rewriting have been used in the literature for proofs of termination of programs in many languages (Haskell [8], Java [25], Prolog [9,27], and C [6,15]), for proofs of equivalence (e.g., in C programs [6,15]), or inference of complexity bounds (for Prolog [9], Jinja [22], and OCaml [1]).

A strength of term rewriting is that it can express inductive data types in a natural way. However, classic term rewriting has a drawback for this use in program analysis: it does not offer a direct representation of primitive data types, which are predefined in most programming languages (e.g., integer numbers and their operations). As classic term rewriting is Turing-complete, it is possible to encode these data types and their operations via terms and recursive rewrite rules. However, a lot of the available domain knowledge baked into the programming language semantics (e.g., the meaning of the operators such as $+$ and $*$) would be "obfuscated in translation", so the existing infrastructure for automated reasoning, such as SMT solvers, cannot directly benefit from this domain knowledge, e.g., in the search for

```
expr ::=
  literal
  | id
  | expr ∼ expr
  | not expr
  | expr.id
  | id(expr, ..., expr)
  | val id = expr in expr
  | if expr then expr else expr
  | if expr isInstanceOf id then expr else expr
  | assume(expr) in expr

literal ::=
  integerLiteral
  | true
  | false
```

Figure 1: Input syntax, where $\sim$ is a symbol in $\{+, -, *, /, \%, \text{and}, \text{or}, >, \geq, <, \leq, ==\}$.

```
rule ::=
  term → term
  | term → term ⟨ term ⟩

term ::=
  literal
  | id
  | id ( term* )
  | term ∼ term
  | ! term

literal ::=
  integerLiteral
  | TRUE
  | FALSE
```

Figure 2: LCTRS syntax, where $\sim$ is a symbol in $\{+, -, *, /, \%, \text{and}, \text{or}, >, \geq, <, \leq, ==\}$.

a termination proof step. This is why in recent years, *constrained* term rewriting [4, 11, 13] has gained popularity for applications in program analysis. This form of term rewriting is augmented with built-in logical constraints, which can be analyzed by SMT solvers. Kop and Nishida [13] proposed *Logically Constrained Term Rewriting Systems (LCTRSs)* as an attempt at unifying different forms of constrained term rewriting. This formalism has gained traction in the community, and several research groups have built tools to analyze different properties of LCTRSs (e.g., Cora [16], CRaris [23], crest [28], Ctrl [14], RMT [2], TcT [32]).

In this work, we propose a translation from Scala programs to LCTRSs such that termination of the generated LCTRS implies termination of the original Scala program. We have implemented the translation in the Stainless program verifier [17]. Stainless already contains a pipeline that successively "desugars" Scala programs for their analysis and applies program transformations between different tree representations of the program. Our prototype implementation enters the tool chain at a suitable level, allowing Stainless to extract an LCTRS for delegation of the termination analysis to an external tool.

To analyze termination of the extracted LCTRSs, our implementation calls the program analysis tool AProVE [7]. As Scala (more precisely, the language fragment that our translation supports) uses a call-by-value evaluation strategy, a proof of *innermost* termination of the generated LCTRS suffices to prove termination of the original Scala program. In this extended abstract, we present our Scala-to-LCTRS export and a preliminary evaluation.

2 Language and Background

We use LCTRSs [13] as an intermediate representation for proving termination of Scala [24] programs. This section defines the subset of Scala that we consider and gives a brief background on LCTRSs.

We consider a purely functional subset of Scala. The input to our system is an abstract syntax tree (AST) that we extract from Stainless at the last stage of its transformation pipeline. We define the supported AST syntax in Figure 1.

An LCTRS is defined as a sequence of rules $l \to r \langle C \rangle$, where l and r are terms, and C is a logical constraint. We define the supported LCTRS syntax in Figure 2. Termination of the LCTRSs that we generate can be checked by external tools such as AProVE [7], which we used in our evaluation, Ctrl [14], or Cora [16]. Initial experimentation with the three tools indicated that AProVE was the most successful in dealing with systems originating from automated translations, which tend to have many more rewrite rules than a hand-optimized system, as we illustrate on an example in the next section. Our translation from Scala to LCTRSs is inspired by the related work on converting C programs to LCTRS in the c2lctrs tool [15]. After illustrating our translation on an example Scala program in Section 3, we define the translation rules in Section 4.

3 Illustrative Example: Formula Programming Assignment

This section gives an overview of our Stainless to LCTRS export on one illustrative example.

We consider the example student submission from the formula benchmark in related work on automated grading of programming assignments [20,29]. Specifically, we consider a minimized definition to focus on the essence of our translation:

```
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

In our evaluation in Section 5, we consider the full definition from the formula benchmark, which includes cases for boolean variables, conjunction and disjunction.

Figure 3 shows the Scala program under analysis. This example is particularly challenging for termination proving due to the recursive call f(Not(l)), whose argument does not trivially decrease over Imply(l, r). In Stainless, measure inference for functions on algebraic data types considers the structural size of data type arguments. In our example, the default size function defines the size of Imply(True, True) as 1 + 0 + 0 = 1, and the size of Not(True) as 1 + 0 = 1 (not decreasing). As a result, Stainless fails to automatically infer a termination measure for this function and cannot prove termination. We thus turn to external termination provers.

The remainder of Figure 3 shows the input of our translation (Stainless tree as defined in Figure 1) and the output of our translation (LCTRS rules as defined in Figure 2). Our translation traverses the input Stainless trees and iteratively constructs the corresponding LCTRS rules. To capture the semantics of the program control flow, during the translation we keep track of conditions as constraints along the execution path. For example, the **else** branch from **if** f(Not(s.l)) **then** true **else** f(s.r) produces the following condition:

f16(t, Imply(s_l, s_r), tmp1) → f18(t, Imply(s_l, s_r), tmp1) ⟨ !tmp1 ⟩

which is then preserved in the subsequent rules for this **else** branch (symbols f18 to f22).

The AProVE termination prover successfully proves termination of our resulting LCTRS, ensuring termination of the original Scala program.

This simplified example already illustrates interesting aspects of our translation, such as handling of algebraic data types (ADTs) and ADT selectors.

```
// Scala source code
def f(t: Formula): Boolean = t match
  case True ⇒ true
  case False ⇒ false
  case Not(t) ⇒ if f(t) then false else true
  case Imply(l, r) ⇒ f(Not(l)) || f(r)
```

```
// Intermediate representation of source code
def f(t: Formula): Boolean =
  val s: Formula = t
  if s isInstanceOf True then true
  else if s isInstanceOf False then false
  else if s isInstanceOf Not then
    val tb: Boolean = f(s.p)
    if tb then false else true
  else if s isInstanceOf Imply then f(Not(s.l)) || f(s.r)
```

```
f(t) → f1(t, t)
f1(t, True) → f2(t, True)
f1(t, Not(s_p)) → f3(t, Not(s_p))
f1(t, False) → f3(t, False)
f1(t, Imply(s_l, s_r)) → f3(t, Imply(s_l, s_r))
f2(t, True) → f22(t, True, TRUE)
f3(t, False) → f4(t, False)
f3(t, Not(s_p)) → f5(t, Not(s_p))
f3(t, True) → f5(t, True)
f3(t, Imply(s_l, s_r)) → f5(t, Imply(s_l, s_r))
f4(t, False) → f22(t, False, FALSE)
f5(t, Not(s_p)) → f6(t, Not(s_p))
f5(t, True) → f12(t, True)
f5(t, False) → f12(t, False)
f5(t, Imply(s_l, s_r)) → f12(t, Imply(s_l, s_r))
f6(t, Not(s_p)) → f7(t, Not(s_p), s_p)
f7(t, Not(s_p), tmp0) → f8(t, Not(s_p), tmp0, f(tmp0))
f8(t, Not(s_p), tmp0, r_f(w)) → f9(t, Not(s_p), w)
f9(t, Not(s_p), tb) → f10(t, Not(s_p), tb) ⟨ tb ⟩
f9(t, Not(s_p), tb) → f11(t, Not(s_p), tb) ⟨ !tb ⟩
f10(t, Not(s_p), tb) → f22(t, Not(s_p), FALSE) ⟨ tb ⟩
f11(t, Not(s_p), tb) → f22(t, Not(s_p), TRUE) ⟨ !tb ⟩
f12(t, Imply(s_l, s_r)) → f13(t, Imply(s_l, s_r))
f12(t, Not(s_p)) → f21(t, Not(s_p))
f12(t, True) → f21(t, True)
f12(t, False) → f21(t, False)
f13(t, Imply(s_l, s_r)) → f14(t, Imply(s_l, s_r), Not(s_l))
f14(t, Imply(s_l, s_r), tmp2) → f15(t, Imply(s_l, s_r), tmp2, f(tmp2))
f15(t, Imply(s_l, s_r), tmp2, r_f(w)) → f16(t, Imply(s_l, s_r), w)
f16(t, Imply(s_l, s_r), tmp1) → f17(t, Imply(s_l, s_r), tmp1) ⟨ tmp1 ⟩
f16(t, Imply(s_l, s_r), tmp1) → f18(t, Imply(s_l, s_r), tmp1) ⟨ !tmp1 ⟩
f17(t, Imply(s_l, s_r), tmp1) → f22(t, Imply(s_l, s_r), TRUE) ⟨ tmp1 ⟩
f18(t, Imply(s_l, s_r), tmp1) → f19(t, Imply(s_l, s_r), tmp1, s_r) ⟨ !tmp1 ⟩
f19(t, Imply(s_l, s_r), tmp1, tmp3) → f20(t, Imply(s_l, s_r), tmp1, tmp3, f(tmp3)) ⟨ !tmp1 ⟩
f20(t, Imply(s_l, s_r), tmp1, tmp3, r_f(w)) → f22(t, Imply(s_l, s_r), w) ⟨ !tmp1 ⟩
f21(t, s) → f22(t, s, e)
f22(tmp4, tmp5, ret) → r_f(ret)
```

Figure 3: Input to Stainless: Scala program (top-left). Input to our translation: Stainless tree at the last pipeline stage, after pattern matching elimination (top-right). Output of our translation: LCTRS rules (bottom). The ⟨ ⟩ syntax denotes the constraints gathered along the execution path.

4 Scala to LCTRS Translation

In this section, we present our translation from Scala to LCTRS.

We take a Scala program as an abstract syntax tree at the last stage of the Stainless pipeline (Inox trees). Figure 1 illustrates the syntax of expressions in these input trees. We then perform a series of transformations to obtain the output trees corresponding to a LCTRS. Figure 2 illustrates the syntax of these output trees. Our pretty-printer then exports the resulting LCTRS rules such that their termination with regard to innermost rewriting can be analyzed by AProVE.

We define the series of transformations as follows:

1. Pre-transformation phase
2. Translation phase
3. Post-transformation phase.

The pre-transformation phase consists of let-transformations (introducing let bindings for expressions) and short-circuiting (eliminating and/or operators by transforming them to if-then-else). For example, in our formula submission in Section 3, the boolean expression f(Not(s.l)) || f(s.r) is transformed to:

val tmp1 = f(Not(s.l))
if tmp1 **then** true **else** f(s.r),

which produces rules corresponding to the symbols f14 to f22 in Figure 3.

The post-transformation phase consists of introducing function wrappers, that is, rules that simplify extraction of function results. For example, in our formula submission in Figure 3, the symbol r_f enables the extraction of function f's result.

The main translation phase is defined in the convert function, which takes the following input:

- f, the name of the function under conversion;
- i, the number of symbols already declared for this function;
- $\overrightarrow{x} = [x_1, ..., x_n]$, a sequence of variables which are known at the start of the function or block;
- $\overrightarrow{y} = [y_1, ..., y_m]$, a sequence of locally declared variables (to be erased at the end of the function or block);
- R: the set of rules declared so far;
- C: the constraints so far;
- S: the statement to convert.

It produces the following output:

- R': the resulting set of rules (the original set R union rules from converting the S statement);
- j: the last index of symbols declared for the S statement.

At the end of this conversion, the resulting set R' defines the resulting LCTRS. To translate a full program, we convert each individual function and then take the union of the resulting sets R' for each function.

We next consider the definition of convert for each case in Figure 1. We use the syntax $\overrightarrow{x}$ to denote a sequence of elements $x_1, ..., x_n$ and the ::: symbol to denote sequence concatenation. We denote substitution of t by b in a by $a\,[t := b\,]$. In each invocation of convert, Stainless expressions are highlighted in gray.

4.1 Assume Statements

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, assume(p) in s) := convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, $p \wedge C$, s).

4.2 Constants and Variables

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, a) := (R', i +1), where:
- R' = $R \cup \{ f_i(\overrightarrow{x}, \overrightarrow{y}) \to f_{i+1}(\overrightarrow{x}, \overrightarrow{y}, a) \langle C \rangle \}$

4.3 Binary Operators

For each binary operator $\sim$[1], we define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, $a \sim b$) := (R', i +1), where:
- R' = $R \cup \{ f_i(\overrightarrow{x}, \overrightarrow{y}) \to f_{i+1}(\overrightarrow{x}, \overrightarrow{y}, a \sim b) \langle C \rangle \}$

4.4 Not

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, not a) := (R', i +1), where:
- R' = $R \cup \{ f_i(\overrightarrow{x}, \overrightarrow{y}) \to f_{i+1}(\overrightarrow{x}, \overrightarrow{y}, \text{not } a) \langle C \rangle \}$

4.5 Function Invocations

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, $g(e_1, ..., e_n)$) := (R', i +2), where:
- R' = $R \cup \{ f_i(\overrightarrow{x}, \overrightarrow{y}) \to f_{i+1}(\overrightarrow{x}, \overrightarrow{y}, g(e_1, ..., e_n)) \langle C \rangle$,
  $f_{i+1}(\overrightarrow{x}, \overrightarrow{y}, r_g(t)) \to f_{i+2}(\overrightarrow{x}, \overrightarrow{y}, t) \langle C \rangle \}$

Here, the term $r_g(t)$ denotes extraction of function g's result. This term is introduced in the post-transformation phase, as the right-hand side of the last rule in g's translation. For example, in the formula submission in Figure 3, the function call f(r) evaluates to the term r_f(w) below the symbol f20.

4.6 Let Bindings

To convert a statement val $b = d$ in e:
- Let (R_1, k) be the result of convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, { }, C, d)
- Let R_1' be R_1 [$f_k(e_1, ..., e_m)$:= $f_k(e_1, ..., e_{|\vec{x}|+|\vec{y}|}, e_m)$]
- Let R_2 be $R \cup R_1$'

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, val $b = d$ in e) := convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$ ::: [b], R_2, C, e).

Here, the sequence $e_1, ..., e_{|\vec{x}|+|\vec{y}|}, e_m$ denotes the removal of local variables which are out of scope at the end of a block. The same substitution takes place in the branches of the **if** expressions (Section 4.7) and in pattern matching cases (Section 4.9).

4.7 If Expressions

To convert a statement if c then s else t:
- Let (R_2, k) be the result of convert(f, i +1, $\overrightarrow{x}$ ::: $\overrightarrow{y}$, [], { }, $c \wedge C$, s)
- Let (R_3, n) be the result of convert(f, k, $\overrightarrow{x}$ ::: $\overrightarrow{y}$, [], { }, $!c \wedge C$, t)
- Let R_2' be R_2 [$f_k(e_1, ..., e_m)$:= $f_n(e_1, ..., e_{|\vec{x}|+|\vec{y}|}, e_m)$]
- Let R_3' be R_3 [$f_n(e_1, ..., e_m)$:= $f_n(e_1, ..., e_{|\vec{x}|+|\vec{y}|}, e_m)$]

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, if c then s else t) := (R', n), where:
- R' = $R \cup \{ f_i(\overrightarrow{x}, \overrightarrow{y}) \to f_{i+1}(\overrightarrow{x}, \overrightarrow{y}) \langle c \wedge C \rangle, f_i(\overrightarrow{x}, \overrightarrow{y}) \to f_k(\overrightarrow{x}, \overrightarrow{y}) \langle !c \wedge C \rangle \} \cup R_2' \cup R_3'$

For example, a function

[1] At this stage of the translation, $\sim$ can be any of the symbols in {+, −, ∗, /, %, >, ≥, <, ≤, ==}. The symbols and, or cannot appear because they get eliminated in the pre-transformation phase.

```
def foo(x: BigInt): BigInt =
  if x > 0 then g(x) else h(x)
```

would be translated to:

foo0(x) → foo1(x, x)
foo1(x, tmp5) → foo2(x, tmp5, 0)
foo2(x, tmp5, tmp6) → foo3(x, tmp5 > tmp6)
foo3(x, tb) → foo4(x, tb) ⟨ tb ⟩
foo3(x, tb) → foo6(x, tb) ⟨ !tb ⟩
foo4(x, tb) → foo5(x, tb, g(x)) ⟨ tb ⟩
foo5(x, tb, ret_g(fresh0)) → foo8(x, tb, fresh0) ⟨ tb ⟩
foo6(x, tb) → foo7(x, tb, h(x)) ⟨ !tb ⟩
foo7(x, tb, ret_h(fresh1)) → foo8(x, tb, fresh1) ⟨ !tb ⟩
foo8(tmp7, tmp8, ret1) → ret_foo(ret1)

This example shows how the control flow of an **if** expression is represented with the help of different function symbols foo4 and foo6 such that a rewrite sequence starting from foo(t) for an integer t will result in exactly one of the branches of the **if** expression being evaluated.

In contrast, with a translation via rules

foo(x) → myif(x > 0, foo1(x), foo2(x))
foo1(x) → r_g(g(x))
foo2(x) → r_h(h(x))
myif(TRUE, x, y) → x
myif(FALSE, x, y) → y

with a single function symbol myif for all **if** expressions, both branches would be evaluated (which is why we avoid this alternative translation).

4.8 ADT Selectors (Field Accesses)

To convert a statement $a.e_k$:

– Let $\overrightarrow{xy}$' be $(\overrightarrow{x} ::: \overrightarrow{y})[a := A\ (\overrightarrow{w})]$, where A is the constructor of $a.e_k$ and $\overrightarrow{w}$ fresh field variables.

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, $a.e_k$) := (R', i +1), where:

– $R' = R \cup \{\ f_i(\overrightarrow{xy}') \to f_{i+1}(\overrightarrow{x}, \overrightarrow{y}, w_k)\ \langle C \rangle\}$

4.9 Pattern Matching

To convert a statement if a isInstanceOf A_l then s else t:

– Let $\{ A_1, ..., A_m \}$ be all constructors of the supertype of A_l
– Let $\overrightarrow{xy_1}, ..., \overrightarrow{xy_m}$ be $(\overrightarrow{x} ::: \overrightarrow{y})[a := A_1\ (\overrightarrow{u})], ..., (\overrightarrow{x} ::: \overrightarrow{y})[a := A_m\ (\overrightarrow{v})]$, respectively
– Let (R_2, k) be the result of convert(f, i +1, $\overrightarrow{xy_l}$, [], {}, C, s)
– Let (R_3, n) be the result of convert(f, k, $\overrightarrow{x} ::: \overrightarrow{y}$, [], {}, C, t)
– Let R_2' be $R_2\ [f_k(e_1, ..., e_m) := f_n(e_1, ..., e_{|\vec{x}|+|\vec{y}|}, e_m)]$

We define convert(f, i, $\overrightarrow{x}$, $\overrightarrow{y}$, R, C, if a isInstanceOf A_l then s else t) := (R', n), where:

– $R' = R \cup \{$
$f_i(\overrightarrow{xy_1}) \to f_k(\overrightarrow{xy_1})\ \langle C \rangle$,..., $f_i(\overrightarrow{xy_{l-1}}) \to f_k(\overrightarrow{xy_{l-1}})\ \langle C \rangle$,
$f_i(\overrightarrow{xy_l}) \to f_{i+1}(\overrightarrow{xy_l})\ \langle C \rangle$,
$f_i(\overrightarrow{xy_{l+1}}) \to f_k(\overrightarrow{xy_{l+1}})\ \langle C \rangle$,..., $f_i(\overrightarrow{xy_m}) \to f_k(\overrightarrow{xy_m})\ \langle C \rangle$
$\} \cup R_2' \cup R_3$

For example, consider the following statement from the formula submission in Figure 3: **if** s isInstanceOf Imply **then** f(Not(s.l)) || f(s.r). Here, A_l corresponds to Imply, the set { A_1, ..., A_m } corresponds to the set of constructors of the type Formula, and the sequence $\overrightarrow{xy_1}$, ... , $\overrightarrow{xy_m}$ at this point in the translation corresponds to (s, True), (s, False), (s, Not(s_p)), (s, Imply(s_l, s_r)), resulting in the following rules:

f12(s, True) $\rightarrow$ f21(s, True) // $fi \rightarrow fk$
f12(s, False) $\rightarrow$ f21(s, False) // $fi \rightarrow fk$
f12(s, Not(s_p)) $\rightarrow$ f21(s, Not(s_p)) // $fi \rightarrow fk$
f12(s, Imply(s_l, s_r)) $\rightarrow$ f13(s, Imply(s_l, s_r)) // $fi \rightarrow fi{+}1$

where f13 and f21 lead to the **then** and **else** branches, respectively.

5 Evaluation

We next present our evaluation of termination checking techniques using our LCTRS export on existing benchmarks drawn from programming assignments. In our evaluation, we consider AProVE's Integer Term Rewrite Systems format (.itrs) [4] and use AProVE to prove their termination.[2]

We compare our results to two existing techniques in Stainless for proving termination: measure inference and measure transfer. Measure inference in Stainless was partially carried over from the Leon verifier [30], whose termination analysis was developed in the MSc thesis of Nicolas Voirol, along with support for generic types and quantifiers [31]. Subsequent work introduced a foundational type system that enforces termination [12]. Measure transfer was later introduced in [21], providing significant automation over measure inference for batched termination proving of equivalent programs, while requiring the original program to be annotated with a termination measure as a termination proof.

Table 1 shows the results of our evaluation. We consider the formula and sigma assignments from [20], as well as the prime and gcd assignments from the experience report in [19]. Each benchmark contains one or more reference solutions annotated with termination measures used for measure transfer, and equivalent student submissions with no measure annotations, where automated measure inference fails. Columns Inference and Transfer show the number of submissions successfully proven terminating by Stainless using measure inference and measure transfer, respectively. Column LCTRS shows number of submissions successfully proven terminating by AProVE.

In the formula benchmark, for 37 submissions where measure inference fails, the evaluation in [20] uses manual annotations to prove termination of 25 submissions (for the remaining submissions, the manual annotator did not find a termination measure). In contrast, measure transfer automatically proves correctness of 27 submissions (73%). The LCTRS export to AProVE proves termination of 24 submissions (65%), which is a significant improvement over measure inference in Stainless. Seven submissions could not be checked due to a limitation of our translation in handling Scala constructs. In the sigma benchmark, measure transfer succeeds for 678 submissions (96%), out of 704 submissions that required manual annotations in the previous evaluation in [20], due to limitations of measure inference. The 26 submissions where measure transfer fails are due to either equivalence proof failures (11) or due to introducing inner functions that exist only in the candidate submission (15). We were unable to evaluate our LCTRS export on this benchmark due to a lack of support for higher-order functions in AProVE's ITRSs. Our experiments show that, when removing the higher-order argument in the sigma submission, AProVE manages to prove termination, while measure inference in Stainless still fails. Since the

[2]Strictly speaking, ITRSs are not LCTRSs, but innermost termination of the LCTRSs when viewed as ITRSs implies innermost termination of the LCTRS, and our LCTRS export currently does not use features that are not supported by ITRSs.

Table 1: Evaluation results on programming assignments. Column LOC shows average number of lines of code per program. Columns F and D indicate average number of function definitions and average number of measure annotations per program, respectively. Columns R and S indicate the number of reference programs and number of submissions, respectively. The last three columns show the results of termination analysis. For each run, we set a 10 second timeout per Z3 solver query in Stainless and a 10 minute overall timeout for AProVE. Column Total Proven shows the total number of submissions successfully proven terminating, either using measure transfer or by AProVE.

Name	LOC	F	D	R	S	Inference	Transfer	LCTRS	Total Proven
formula	59	2	1	1	37	0	27	24	28
sigma	10	1	1	3	704	0	678	0	678
prime	21	4	2	2	22	0	5	14	14
gcd	9	1	1	2	41	0	22	15	27

higher-order argument does not affect termination in this case, in the future, we will explore program slicing techniques to remove higher-order arguments in the LCTRS export. In the prime benchmark, we encounter submissions that, when manually annotated, pass termination checks. However, measure transfer fails, because the inner function's measure in the reference solutions gives a negative measure when transferred to the inner function of the submission. LCTRS export outperforms measure transfer in this benchmark, succeeding for 64% of purely-functional submissions (compared to 23% when using measure transfer). Out of three benchmarks with multiple reference programs, only gcd has different termination measures. Termination analysis via the LCTRS export to AProVE outperforms measure inference in Stainless on this benchmark. However, this could be due to the use of unbounded integers in the LCTRS export, instead of bounded integers in the original submissions. When using unbounded integers in Stainless, measure inference succeeds for 31 submissions (out of 41 which were previously timing out due to bounded integer arithmetic).

Overall, our evaluation suggests that our approach using LCTRS export with AProVE provides a significant improvement over the measure inference in Stainless, but is still less effective than measure transfer. However, unlike our approach, measure transfer requires initial manual measure annotations in the reference program and is only useful when there is more than one program under analysis. Our approach using LCTRS export is more general.

6 Discussion and Future Work

Our work has several limitations in its current state.

Primitive Data Types So far, the only primitive data types that our exported LCTRSs contain are unbounded integers and booleans. The reason is that our current translation is backwards compatible to ITRSs as an early form of constrained rewriting [4]. This is why we currently export both unbounded and bounded Scala integers to unbounded integers on LCTRS level. Consequently, our termination proofs are correct only if there are no overflows in the original Scala program. To ensure correctness, Stainless checks for overflows prior to running the LCTRS export.

To illustrate the issues with the discrepancy between bounded integers in Scala and their translation as unbounded integers, we consider the following example program:

```
def overflow_fun(i: Int, n: Int): Int =
  if i ≤ n then overflow_fun(i + 1, n) else i
```

This program does not terminate due to a possible overflow of i for $n = 2^{31} - 1$. However, our translation would result in an LCTRS which uses unbounded integers and thus terminates. Currently, our use of the Scala-to-LCTRS translation is to prove termination of programs after having Stainless check for safety errors such as integer overflows and division by zero. In our overflow_fun example, Stainless will report addition overflow in this program, and the analysis pipeline never runs the termination check. Similarly, safety errors can occur in binary operations (Section 4.3). Stainless can detect such errors in an earlier phase prior to termination checks, omitting the need for LCTRS translation and termination checks.

LCTRSs provide more generality, both in theory and in practice: for example, not only do LCTRSs support bounded integers out of the box, but there are also dedicated termination proving techniques for such bounded integers [18]. A natural next step would be to export bounded Scala integers to bounded LCTRS integers.

Lazy Evaluation Our translation only supports a subset of Scala defined in Figure 1. Other advanced Scala features, such as **lazy val**s, are out of the scope of our work. In future work, we could adapt the approach used for proving termination of Haskell programs via term rewriting [8]. This approach uses a form of abstract interpretation [3] with the Haskell evaluation strategy to obtain rewrite rules whose *innermost* termination implies termination of the original Haskell program with the Haskell evaluation strategy. Alternatively/in addition, we could consider integrating lazy rewriting [26] into the target language of our translation, although this integration would require setting up more complex termination analysis infrastructure.

Higher-Order Functions Another limitation of the current state of our work is that our export does not support higher-order functions, a common feature in functional programs. This matters even for our benchmark set from a restricted application domain: recall that our translation was unable to handle the sigma benchmark. However, in the meantime LCTRSs have been extended with higher-order functions [10, 11] to a formalism called *Logically Constrained Simply-typed Term Rewriting Systems (LCSTRSs)*, motivated by translation-based termination analysis for functional programs (such as the present work). What is missing in the above work is support for techniques to simplify LCSTRSs with many rules. Very recent work [5] can help us overcome this limitation: in contrast to the earlier papers on LCSTRSs, this paper targets *innermost* and *call-by-value* termination rather than termination with regard to *arbitrary* rewrite strategies. This makes it possible to "chain" consecutive function calls, which makes systems much more amenable to automated termination analysis. As this addresses the main advantage of ITRSs, we plan to target LCSTRSs with Cora [16] as termination prover to benefit from their additional generality. Thus, an explicit removal of higher-order functions as part of the export may not be needed.

Modular Analysis A limitation of our export is that the LCTRS does not contain information which function symbols are allowed to occur in a start term of a possible infinite rewrite sequence. As a result, the termination prover analyzes termination for arbitrary start terms.

For Scala programs where a helper function would be non-terminating for some inputs on which it can never be called by the entry-point function, the resulting LCTRS would still be non-terminating from corresponding terms. This behavior is in line with current termination analysis in Stainless, which

```
def main(x: BigInt, y: BigInt): BigInt =
  require(x > y)
  helper(x, y)

def helper(x: BigInt, y: BigInt): BigInt =
  if x ≤ 0 then y − x
  else if 2 ∗ x > y then x − y
  else 1 + helper(x, y)
```

Figure 4: The main function invokes the non-terminating helper function under the condition $x > y$.

requires termination of all functions for all inputs.[3]

For example, the main function in Figure 4 invokes the helper function only when $x > y$ holds. However, this information is not visible in the translation of the helper function. Because the helper function is not terminating when considered in isolation (due to the recursive call helper(x, y)), the resulting LCTRS is non-terminating.

Cora supports labeling symbols that should not be considered as possible start terms as "private" [10]. Adding such labels in our export would provide the needed information for a more powerful analysis.

Proof of Correctness Future work may include a mechanized proof of correctness of our translation, including formal semantics of the supported Scala subset and of LCTRS and a proof of preservation of termination and non-termination properties between the input and the generated output.

Propagation of information from the termination analysis to Scala Currently the only information from the termination analysis tool that reaches the user is whether termination was proved or disproved, or no output was found. This can be accompanied by a human-readable proof on rewrite level, but this has the downside that the user of Stainless cannot be expected to be familiar with the intricacies of termination analysis of term rewriting.

Future work will involve extracting information from the proof that can be presented to a Scala programmer. From a termination proof, a termination measure for a Scala function could be extracted and added as an annotation to the Scala function. From a non-termination proof, a non-terminating term could be extracted and translated back to Scala level. Here it would be necessary to check whether this non-terminating term would be reachable from a valid start location of the Scala function.

7 Conclusions

We have presented an approach for termination analysis of Scala programs via a translation to LCTRSs. Our approach allows for an automated extraction of an LCTRS such that innermost termination of the LCTRS implies termination of the Scala program. We have integrated our approach into the transformation pipeline in the Stainless program verifier. Our experiments on benchmarks from student programming assignments indicate complementarity to other approaches, such as the existing measure inference in Stainless or the measure transfer of manual measure annotations from related functions.

[3] In a system like Stainless, if the intention is to have a function that only expects certain inputs, then this should be reflected in the parameter data types and function preconditions.

References

[1] Martin Avanzini, Ugo Dal Lago & Georg Moser (2015): *Analysing the complexity of functional programs: higher-order meets first-order*. In: *Proc. ICFP*, pp. 152–164, doi:10.1145/2784731.2784753.

[2] Ştefan Ciobâcă, Dorel Lucanu & Andrei-Sebastian Buruiana (2023): *Operationally-based program equivalence proofs using LCTRSs*. *J. Log. Algebraic Methods Program.* 135, p. 100894, doi:10.1016/J.JLAMP.2023.100894.

[3] Patrick Cousot & Radhia Cousot (1977): *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In: *Proc. POPL*, pp. 238–252, doi:10.1145/512950.512973.

[4] Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp & Stephan Falke (2009): *Proving Termination of Integer Term Rewriting*. In: *Proc. RTA*, pp. 32–47, doi:10.1007/978-3-642-02348-4_3.

[5] Carsten Fuhs, Liye Guo & Cynthia Kop (2025): *An Innermost DP Framework for Constrained Higher-Order Rewriting*. In: *Proc. FSCD*, p. To appear.

[6] Carsten Fuhs, Cynthia Kop & Naoki Nishida (2017): *Verifying Procedural Programs via Constrained Rewriting Induction*. *ACM Trans. Comput. Logic* 18(2), doi:10.1145/3060143.

[7] 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 & René Thiemann (2017): *Analyzing Program Termination and Complexity Automatically with AProVE*. *J. Autom. Reason.* 58(1), pp. 3–31, doi:10.1007/S10817-016-9388-Y.

[8] Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski & René Thiemann (2011): *Automated Termination Proofs for Haskell by Term Rewriting*. *ACM Trans. Program. Lang. Syst.*, doi:10.1145/1890028.1890030.

[9] Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes & Carsten Fuhs (2012): *Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs*. In: *Proc. PPDP*, pp. 1–12, doi:10.1145/2370776.2370778.

[10] Liye Guo, Kasper Hagens, Cynthia Kop & Deivid Vale (2024): *Higher-Order Constrained Dependency Pairs for (Universal) Computability*. In: *Proc. MFCS*, pp. 57:1–57:15, doi:10.4230/LIPICS.MFCS.2024.57.

[11] Liye Guo & Cynthia Kop (2024): *Higher-Order LCTRSs and Their Termination*. In: *Proc. ESOP (2)*, pp. 331–357, doi:10.1007/978-3-031-57267-8_13.

[12] Jad Hamza, Nicolas Voirol & Viktor Kunčak (2019): *System FR: Formalized Foundations for the Stainless Verifier*. *Proc. ACM Program. Lang.* 3(OOPSLA), pp. 166:1–166:30, doi:10.1145/3360592.

[13] Cynthia Kop & Naoki Nishida (2013): *Term Rewriting with Logical Constraints*. In: *Proc. FroCoS*, pp. 343–358, doi:10.1007/978-3-642-40885-4_24.

[14] Cynthia Kop & Naoki Nishida (2015): *Constrained Term Rewriting tooL*. In: *Proc. LPAR*, pp. 549–557, doi:10.1007/978-3-662-48899-7_38.

[15] Cynthia Kop & Naoki Nishida (2015): *Converting C to LCTRSs*. https://www.trs.cm.is.nagoya-u.ac.jp/c2lctrs/formal.pdf.

[16] Cynthia Kop et al.: *The Cora Analyzer*. Available at `https://github.com/hezzel/cora`.

[17] EPFL LARA (2025): *Stainless*. `https://github.com/epfl-lara/stainless`.

[18] Ayuka Matsumi, Naoki Nishida, Misaki Kojima & Donghoon Shin (2023): *On Singleton Self-Loop Removal for Termination of LCTRSs with Bit-Vector Arithmetic*. *CoRR* abs/2307.14094, doi:10.48550/ARXIV.2307.14094. arXiv:2307.14094. In Proc. WST 2023.

[19] Dragana Milovančević, Mario Bucev, Marcin Wojnarowski, Samuel Chassot & Viktor Kunčak (2025): *Formal Autograding in a Classroom*. In: *Proc. ESOP*, pp. 154–174, doi:10.1007/978-3-031-91121-7_7.

[20] Dragana Milovančević & Viktor Kunčak (2023): *Proving and Disproving Equivalence of Functional Programming Assignments*. *Proc. ACM Program. Lang.* 7(PLDI), pp. 928–951, doi:10.1145/3591258.

[21] Dragana Milovančević, Carsten Fuhs, Mario Bucev & Viktor Kunčak (2024): *Proving Termination via Measure Transfer in Equivalence Checking*. In: *Proc. iFM*, pp. 75–84, doi:10.1007/978-3-031-76554-4_5.

[22] Georg Moser & Michael Schaper (2018): *From Jinja bytecode to term rewriting: A complexity reflecting transformation*. *Inf. Comput.* 261, pp. 116–143, doi:10.1016/J.IC.2018.05.007.

[23] Naoki Nishida & Misaki Kojima (2024): *CRaris: CR checker for LCTRSs in ARI Style*. In: *Proc. IWC*, pp. 83–84. In `http://cl-informatik.uibk.ac.at/iwc/iwc2024.pdf`.

[24] Martin Odersky, Lex Spoon & Bill Venners (2019): *Programming in Scala, Fourth Edition (A comprehensive step-by-step guide)*. Artima, Sunnyvale, CA, USA.

[25] Carsten Otto, Marc Brockschmidt, Christian Essen & Jürgen Giesl (2010): *Automated Termination Analysis of Java Bytecode by Term Rewriting*. In: *Proc. RTA*, pp. 259–276, doi:10.4230/LIPIcs.RTA.2010.259.

[26] Felix Schernhammer & Bernhard Gramlich (2007): *Termination of Lazy Rewriting Revisited*. In: *Proc. WRS*, pp. 35–51, doi:10.1016/J.ENTCS.2008.03.052.

[27] Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik & René Thiemann (2010): *Automated Termination Analysis for Logic Programs With Cut*. *Theory Pract. Log. Program.* 10(4-6), pp. 365–381, doi:10.1017/S1471068410000165.

[28] Jonas Schöpf & Aart Middeldorp (2025): *Automated Analysis of Logically Constrained Rewrite Systems using crest*. In: *Proc. TACAS*, pp. 124–144, doi:10.1007/978-3-031-90643-5_7.

[29] Dowon Song, Myungho Lee & Hakjoo Oh (2019): *Automatic and Scalable Detection of Logical Errors in Functional Programming Assignments*. *Proc. ACM Program. Lang.* 3(OOPSLA), pp. 188:1–188:30, doi:10.1145/3360614.

[30] Philippe Suter, Ali Sinan Köksal & Viktor Kuncak (2011): *Satisfiability Modulo Recursive Programs*. In: *Proc. SAS*, pp. 298–315, doi:10.1007/978-3-642-23702-7_23.

[31] Nicolas Voirol (2023): *Termination Analysis in a Higher-Order Functional Context*. Master's thesis, EPFL. Available at `http://infoscience.epfl.ch/record/311772`.

[32] Sarah Winkler & Georg Moser (2020): *Runtime Complexity Analysis of Logically Constrained Rewriting*. In: *Proc. LOPSTR*, pp. 37–55, doi:10.1007/978-3-030-68446-4_2.

A Cyclic Proof System for Partial Correctness of Separation Logic with Recursive Procedure Calls

Takumi Sato Koji Nakazawa

Graduate School of Informatics
Nagoya University
Nagoya, Japan
sato.takumi@sqlab.jp knak@i.nagoya-u.ac.jp

Separation logic is an extension of Hoare logic, enabling reasoning about programs manipulating heap memories. Al Ameen and Tatsuta proposed a proof system for partial correctness in separation logic with recursive calls, establishing its soundness and relative completeness. Cyclic proof systems are proof systems for inductive statements, where induction is represented by a cyclic structure in proofs. In this work, we propose a cyclic proof system for partial correctness of separation logic for programs with recursive calls. In this system, we show that one can prove the memory safety of pointer-based programs with recursive calls using a cyclic-proof approach, and we also prove the soundness and relative completeness of our cyclic proof system. Therefore, the provability of our system is equivalent to that of Al Ameen and Tatsuta's system.

1 Introduction

1.1 Background

Separation Logic [10] is an extension of Hoare logic [8] that enables reasoning about programs manipulating heap memories. A program P is said to satisfy a *partial correctness* if, whenever the precondition A holds and execution of P terminates, the resulting state satisfies the postcondition B. We denote this partial correctness by a Hoare triple $\{A\}\ P\ \{B\}$. We say that program P satisfies *total correctness* when it satisfies termination in addition to partial correctness. Separation logic allows reasoning about the heap memory state by introducing the concept of a singleton heap $x \mapsto a$, representing a heap portion consisting of one memory cell whose address is x and that stores the value a, and the separating conjunction $A * B$, describing that A and B hold in disjoint portions of the heap. These features enable local reasoning about memory states. Al Ameen and Tatsuta proposed a proof system for partial correctness in separation logic with recursive calls, establishing its soundness and relative completeness [1]. In this system, Hoare triples are extended to contextual Hoare triples, allowing to assume specifications of procedures that appear recursively in procedure bodies. This extension to contextual triples is based on the idea of Oheimb [12].

Cyclic Proof Systems [7] are proof systems for inductive statements and fixed-point operators, and they represent induction by allowing some leaves in the proof's derivation tree to cycle back to internal nodes. For the proof to be sound, it must satisfy a soundness condition called the *global trace condition*, which is decidable [6]. Rowe and Brotherston [11] proposed a cyclic proof system for total correctness of separation logic aimed at verifying the termination of programs with recursive calls. In this system, inductive predicates are labeled, and the labels track the number of times to unfold the predicates. Every cycle in a proof of this system must include an infinite descent concerning label valuations. Since predicate unfoldings cannot occur infinitely often, the global trace condition guarantees program termination.

Brotherston and Gorogiannis [5] proposed a framework for abducting safety and termination preconditions for heap-manipulating while programs using a cyclic proof-based approach. Here, safety is the property that, starting execution from a given state, the program never transitions into undesirable states (e.g., memory errors), and termination is the property that, in addition to safety, the program eventually halts. These correspond to partial correctness and total correctness when a postcondition is specified. Brotherston et al. demonstrated the relative completeness of the cyclic proof system for total correctness of separation logic by encoding the weakest preconditions [3].

1.2 Our contribution

In this work, we propose a cyclic proof system for partial correctness of separation logic for programs with recursive calls and demonstrate its soundness and relative completeness. Following the approach of Brotherston et al. [3], we divide the inference rules into Symbolic Execution Proof Rules and Logical Proof Rules. Following Brotherston and Gorogiannis [5], we define the global trace condition for partial correctness as follows: in any infinite path in a cyclic pre-proof, Symbolic Execution Proof Rules are applied infinitely many times. The Symbolic Execution Proof Rules correspond to the symbolic execution of the first command of a program. For example, in the (READ) rule, where $[E]$ denotes the content of the memory cell whose address is the value of E,

$$\frac{\vdash \{\phi[x'/x] \wedge x = E'[x'/x] * (E \mapsto E')[x'/x]\}\ C\ \{\psi\}}{\vdash \{\phi * E \mapsto E'\}\ x := [E]; C\ \{\psi\}}\ (x' \text{ is fresh}),$$

in which the precondition in the conclusion is transferred to the precondition of the assumption through symbolic execution. For partial correctness, we consider finite execution sequences of programs; hence, our global trace condition guarantees soundness.

Comparison with prior work is as follows:

- Brotherston and Gorogiannis [5] proposed a cyclic proof system for partial correctness (through a global trace condition) but does not support postconditions or recursive calls. Our system includes postconditions and recursive calls. Furthermore, we established the relative completeness of our system.
- Brotherston et al. [3] proposed a cyclic proof system for total correctness without recursive calls and proved its relative completeness. Our system focuses on partial correctness, supports recursive calls, and establishes relative completeness via a different technique.
- Rowe and Brotherston [11] proposed a cyclic proof system for total correctness that includes recursive calls, and its relative completeness has not been established. Our system differs in that it targets partial correctness rather than total correctness.

In the proof of relative completeness, we transform proofs from Al Ameen and Tatsuta's system into our own. Therefore, since both systems handle the same class of assertion languages, the provability of our system is equivalent to that of Al Ameen and Tatsuta's system.

In our system, one can prove the memory safety of pointer-based programs with recursive calls using a cyclic-proof approach.

The remainder of this paper is as follows. section 2 describes the syntax and semantics of our programming and assertion languages. In section 3, we provide our system's soundness condition and present a proof example. In section 4, we prove the soundness of our system, and in section 5, we prove its relative completeness, and we conclude in section 6.

2 Programs and Separation Logic Assertions

In this section, we describe the language of our system. The language of our system conforms to the language of Al Ameen and Tatsuta's system [1]. The programs include recursive calls and heap manipulating commands. We assume every variable is global, and neither local variable nor parameter do not appear. We also assume that every function called within the program is accompanied by its definition.

2.1 Syntax and Semantics of Programs

2.1.1 Syntax of Programs

We use $x, y, \cdots$ for program variables. Var is the set of program variables. Programs consist of sequences of procedures and one distinguished main procedure. A procedure is declared as $\texttt{Procedure}\ p\ \{C\}$ where p is a procedure name and C is a command.

The arithmetic expressions E and the boolean expressions B are defined as follows.

Definition 2.1 (Expressions)**.**

$$E ::= x \mid 0 \mid 1 \mid \mathsf{nil} \mid E+E \mid E \times E \qquad\qquad B ::= E = E \mid E < E \mid \neg B \mid B \wedge B$$

The symbol nil means null pointer. The boolean expressions are used as branching conditions.

Definition 2.2 (Commands)**.** We define commands C as $C ::= \ \varepsilon \mid c\ ; C$, where single commands c are defined as follows.

$$\begin{aligned} c ::= x{:=}E \mid x{:=}[E] \mid [E]{:=}E' \mid x{:=}\,&\texttt{cons}(E,E') \mid \texttt{dispose}(E) \\ \mid p \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi} \mid \texttt{while}\ B\ \texttt{do}\ C_1\ \texttt{od}\ &C \end{aligned}$$

For commands C_1 and C_2, we define the composition $C_1 \fatsemi C_2$ by $\varepsilon \fatsemi C_2 = C_2$ and $(c\ ; C_1) \fatsemi C_2 = c\ ; (C_1 \fatsemi C_2)$.

c denotes an atomic command, a branching command, or a looping command. The atomic commands consist of an assignment ($x := E$), a dereference ($x := [E]$), a mutation ($[E] := E'$), a memory allocation ($x := \texttt{cons}(E, E')$), a memory deallocation ($\texttt{dispose}(E)$), and a procedure call (p). We define $\mathsf{fv}(C)$ as the set of variables that occur in C.

Definition 2.3 (Programs)**.** A program P is defined as $P ::= \langle (\texttt{Procedure}\ p_i\ \{C_i\})_{i=1,\ldots,n}, C\rangle$, where C is a command of the main procedure. For the programs P, we define $body(p_i) = C_i$ and always assume for any p called in P, $\exists i.p = p_i$.

We often omit the procedure declarations $(\texttt{Procedure}\ p_i\ \{C_i\})_{i=1,\ldots,n}$ when they are clear and only write the main procedure C to denote a program. We define $eproc(C)$ as the set of procedures that may be hereditarily called in the execution of C as follows.

$$\begin{aligned} eproc_0(C) &= \emptyset \\ eproc_{i+1}(C) &= eproc_i(C) \cup \{p \mid \text{procedures called in the execution of procedures in } eproc_i(C)\} \\ eproc(C) &= \bigcup_i eproc_i(C) \end{aligned}$$

We also write $mod(C)$ for the set of all variables whose values are modified by C, that is, all of the variables x such that C contains a command of the form $x := \cdots$. We define $emod(C)$ as the set of variables that may be modified in the execution of C as follows.

$$emod(C) := mod(C) \cup \bigcup_{p \in eproc(C)} mod(body(p)).$$

2.1.2 Semantics of Programs

Let Val denote the set of values and Loc denote the set of locations available for use as the heap. We assume that $\mathsf{Val} = \mathbb{N}$ and $\mathsf{Loc} = \mathbb{N} \setminus \{0\}$.

Definition 2.4 (Heap model)**.** A heap model is represented by a pair (s,h), where s denotes a store and h denotes a heap. A store s is a function $s : \mathsf{Var} \to \mathsf{Val}$. A heap h is a finite partial function $h : \mathsf{Loc} \to \mathsf{Val}$. $\mathrm{dom}(h)$ denotes the domain of h. Stores denotes the set of all stores. Heaps denotes the set of all heaps.

For two heaps h_1 and h_2, if $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$, they are said to be disjoint. The heap composition operation $h_1 \circ h_2$ between disjoint heaps h_1 and h_2 is defined as follows:

$$(h_1 \circ h_2)(l) = \begin{cases} h_1(l) & \text{if } l \in \mathrm{dom}(h_1) \\ h_2(l) & \text{if } l \in \mathrm{dom}(h_2) \\ \text{undefined} & \text{if } l \notin \mathrm{dom}(h_1) \cup \mathrm{dom}(h_2). \end{cases}$$

Both the interpretation of arithmetic expressions E in s and the interpretation of boolean expressions are given by the standard definitions.

Definition 2.5 (Operational semantics)**.** We model a state as a configuration κ, which is either a triple (C,s,h) or a distinguished configuration *abort*, where C is the remaining part of the program, s is a store, and h is a heap. *abort* represents a memory error.

The operational semantics of the program are given by the small-step relation $\rightsquigarrow$ on configurations defined by the rules in Figure 1. Here, an execution of n steps is denoted by $\overset{n}{\rightsquigarrow}$, and an execution of zero or more steps is denoted by $\overset{*}{\rightsquigarrow}$. Furthermore, when a configuration κ satisfies $\kappa \overset{n}{\rightsquigarrow} (\varepsilon,s,h)$ for some n, we say (s,h) is a *final state*, and we write $\kappa \overset{n}{\rightsquigarrow} (s,h)$. Assuming procedure declarations are provided separately, they are omitted from the configuration.

2.2 Syntax and Semantics of Assertions

Our assertion language is defined as follows.

Definition 2.6 (Formulas)**.**

$$\phi ::= \mathtt{emp} \mid E = E \mid E < E \mid E \mapsto E \mid \neg\phi \mid \phi \vee \phi \mid \exists x \phi \mid \phi * \phi \mid \phi \mathrel{-\!\!*} \phi$$

$\mathsf{fv}(\phi)$ denotes the set of free variables in ϕ. $E \mapsto (E', E'')$ is an abbreviation for $E \mapsto E' * E+1 \mapsto E''$.

Formulas denote concrete memory states' properties via a satisfaction relation $\models$. We can use the logical connectives $\wedge$, $\rightarrow$, and $\forall$ by combining $\neg$, $\vee$, and $\exists$ in a usual way.

Definition 2.7 (Satisfaction relation)**.** The satisfaction relation $\models$ between a heap model and a formula is defined as shown in Figure 2.

Note that the boolean expressions are special forms of the formulas. It is easy to see that $(s,h) \models B$ does not depend on h, and we also write $s \models B$.

Definition 2.8 (Entailment)**.** We write $\phi \models \psi$ for formulas ϕ, ψ to mean $(s,h) \models \phi$ implies $(s,h) \models \psi$ for any (s,h).

In our proof system, we verify the partial correctness of programs by manipulating Hoare triples of the form $\{\phi\}\, C\, \{\psi\}$.

Definition 2.9 (Validity of Hoare triples)**.** A Hoare triple $\{\phi\}\, C\, \{\psi\}$ is valid iff, for any (s,h) that satisfies $(s,h) \models \phi$, $(C,s,h) \overset{*}{\rightsquigarrow}$ *abort* does not hold and every final state (s',h') of (C,s,h) satisfies $(s',h') \models \psi$.

$$\frac{}{(x{:=}E;C,s,h) \rightsquigarrow (C,s[x \mapsto [\![E]\!]s],h)} \qquad \frac{[\![E]\!]s \in \mathrm{dom}(h)}{(x{:=}[E];C,s,h) \rightsquigarrow (C,s[x \mapsto h([\![E]\!]s)],h)}$$

$$\frac{[\![E]\!]s \in \mathrm{dom}(h)}{([E]{:=}E';C,s,h) \rightsquigarrow (C,s,h[[\![E]\!]s \mapsto [\![E']\!]s])}$$

$$\frac{l, l+1 \in \mathsf{Loc} \setminus \mathrm{dom}(h)}{(x{:=}\,\mathtt{cons}(E,E');C,s,h) \rightsquigarrow (C,s[x \mapsto l],h[l \mapsto [\![E]\!]s][(l+1) \mapsto [\![E']\!]s])}$$

$$\frac{[\![E]\!]s \in \mathrm{dom}(h)}{(\mathtt{dispose}(E);C,s,h) \rightsquigarrow (C,s,h[[\![E]\!]s \mapsto \bot])} \qquad \frac{s \models B}{(\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\ \mathtt{fi};C,s,h) \rightsquigarrow (C_1 \fatsemi C,s,h)}$$

$$\frac{s \not\models B}{(\mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2\ \mathtt{fi};C,s,h) \rightsquigarrow (C_2 \fatsemi C,s,h)}$$

$$\frac{s \models B}{(\mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od};C',s,h) \rightsquigarrow (C \fatsemi \mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od};C',s,h)} \qquad \frac{s \not\models B}{(\mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od};C',s,h) \rightsquigarrow (C',s,h)}$$

$$\frac{body(p) = C'}{(p;C,s,h) \rightsquigarrow (C' \fatsemi C,s,h)} \qquad \frac{[\![E]\!]s \notin \mathrm{dom}(h)}{(x{:=}[E];C,s,h) \rightsquigarrow abort} \qquad \frac{[\![E]\!]s \notin \mathrm{dom}(h)}{([E]{:=}E';C,s,h) \rightsquigarrow abort}$$

$$\frac{[\![E]\!]s \notin \mathrm{dom}(h)}{(\mathtt{dispose}(E);C,s,h) \rightsquigarrow abort}$$

Figure 1: Operational semantics
(For a sequence $\vec{s}$, we denote the update of a function f by the partial function $(\vec{s} \mapsto \vec{t})$ as $f[\vec{s} \mapsto \vec{t}]$. The updated value is undefined when we write $f[\vec{s} \mapsto \bot]$.)

$$\begin{array}{rcl}
(s,h) \models \mathtt{emp} & \Leftrightarrow & \mathrm{dom}(h) = \emptyset \\
(s,h) \models E_1 = E_2 & \Leftrightarrow & [\![E_1]\!]s = [\![E_2]\!]s \\
(s,h) \models E_1 < E_2 & \Leftrightarrow & [\![E_1]\!]s < [\![E_2]\!]s \\
(s,h) \models E_1 \mapsto E_2 & \Leftrightarrow & \mathrm{dom}(h) = \{[\![E_1]\!]s\} \text{ and } h([\![E_1]\!]s) = [\![E_2]\!]s \\
(s,h) \models \neg\phi & \Leftrightarrow & (s,h) \not\models \phi \\
(s,h) \models \phi_1 \vee \phi_2 & \Leftrightarrow & (s,h) \models \phi_1 \text{ or } (s,h) \models \phi_2 \\
(s,h) \models \exists x\phi & \Leftrightarrow & \exists m \in \mathsf{Val}.(s[x \mapsto m],h) \models \phi \\
(s,h) \models \phi_1 * \phi_2 & \Leftrightarrow & \exists h_1,h_2.h = h_1 \circ h_2 \text{ and } (s,h_1) \models \phi_1 \text{ and } (s,h_2) \models \phi_2 \\
(s,h) \models \phi_1 \mathrel{-\!\!*} \phi_2 & \Leftrightarrow & \forall h_1,h_2.h_2 = h_1 \circ h \text{ and } (s,h_1) \models \phi_1 \text{ implies } (s,h_2) \models \phi_2
\end{array}$$

Figure 2: Satisfaction relation

$$\text{(STOP)}\ \frac{}{\vdash \{\phi\}\ \varepsilon\ \{\psi\}}(\phi \models \psi) \qquad \text{(WRITE)}\ \frac{\vdash \{\phi * E \mapsto E'\}\ C\ \{\psi\}}{\vdash \{\phi * E \mapsto E''\}\ [E] := E'; C\ \{\psi\}} \qquad \text{(ASSIGN)}\ \frac{\vdash \{\phi[x'/x] \wedge x = E[x'/x]\}\ C\ \{\psi\}}{\vdash \{\phi\}\ x := E; C\ \{\psi\}}(x' \text{ is fresh})$$

$$\text{(READ)}\ \frac{\vdash \{(\phi * E \mapsto E')[x'/x] \wedge x = E'[x'/x]\}\ C\ \{\psi\}}{\vdash \{\phi * E \mapsto E'\}\ x := [E]; C\ \{\psi\}}(x' \text{ is fresh}) \qquad \text{(DISPOSE)}\ \frac{\vdash \{\phi\}\ C\ \{\psi\}}{\vdash \{\phi * E \mapsto E'\}\ \texttt{dispose(}E\texttt{);}C\ \{\psi\}}$$

$$\text{(CONS)}\ \frac{\vdash \{\phi[x'/x] * x \mapsto (E, E')[x'/x]\}\ C\ \{\psi\}}{\vdash \{\phi\}\ x := \texttt{cons(}E, E'\texttt{)}\ ; C\ \{\psi\}}(x' \notin \mathsf{fv}(\phi, E, E')) \qquad \text{(IF)}\ \frac{\vdash \{\phi \wedge B\}\ C_1 \fatsemi C\ \{\psi\} \qquad \vdash \{\phi \wedge \neg B\}\ C_2 \fatsemi C\ \{\psi\}}{\vdash \{\phi\}\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi;}C\ \{\psi\}}$$

$$\text{(WHILE)}\ \frac{\vdash \{\phi \wedge B\}\ C' \fatsemi (\texttt{while}\ B\ \texttt{do}\ C'\ \texttt{od;}C)\ \{\psi\} \qquad \vdash \{\phi \wedge \neg B\}\ C\ \{\psi\}}{\vdash \{\phi\}\ \texttt{while}\ B\ \texttt{do}\ C'\ \texttt{od;}C\ \{\psi\}} \qquad \text{(PROC)}\ \frac{\vdash \{\phi\}\ body(p) \fatsemi C\ \{\psi\}}{\vdash \{\phi\}\ p; C\{\psi\}}$$

Figure 3: Symbolic Execution Proof Rules

$$\text{(BOT)}\ \frac{}{\vdash \{\bot\}\ C\ \{\psi\}} \qquad \text{(CONSEQ)}\ \frac{\vdash \{\chi\}\ C\ \{\xi\}}{\vdash \{\phi\}\ C\ \{\psi\}}(\phi \models \chi, \xi \models \psi) \qquad \text{(FRAME)}\ \frac{\vdash \{\phi\}\ C\ \{\psi\}}{\vdash \{\phi * \xi\}\ C\ \{\psi * \xi\}}(\mathsf{fv}(\xi) \cap emod(C) = \emptyset)$$

$$\text{(SUBST)}\ \frac{\vdash \{\phi\}\ C\ \{\psi\}}{\vdash \{\phi[E/x]\}\ C\ \{\psi[E/x]\}}(x \notin \mathsf{fv}(C), x \in \mathsf{fv}(\psi) \Rightarrow \mathsf{fv}(E) \cap \mathsf{fv}(C) = \emptyset) \qquad \text{(SEQ)}\ \frac{\vdash \{\phi\}\ C_1\ \{\chi\} \qquad \vdash \{\chi\}\ C_2\ \{\psi\}}{\vdash \{\phi\}\ C_1 \fatsemi C_2\ \{\psi\}}$$

$$\text{(}\exists\text{VAR)}\ \frac{\vdash \{\phi[y/x]\}\ C\ \{\psi\}}{\vdash \{\exists x.\phi\}\ C\ \{\psi\}}(y \text{ is fresh})$$

Figure 4: Logical Proof Rules

3 Cyclic Proof System

In this section, we define a cyclic proof system for partial correctness.

3.1 Definition of our cyclic proof system

The rules in our proof system are given in Figure 3 and Figure 4. The *Symbolic Execution Rules* in Figure 3, viewed from conclusion to premise, are rules that execute a single command at the beginning of the program and modify the precondition. The *Logical rules* in Figure 4 are the standard rules of separation logic.

Since our system adopts a cyclic proof system, it is possible that in a finite derivation tree, non-axiom leaves are closed by back-links to internal nodes. Therefore, cyclic proof systems require additional conditions to ensure soundness called the global trace condition [7].

Definition 3.1 (Cyclic pre-proofs)**.** A *cyclic pre-proof* is defined as a pair $(\mathscr{P}, F)$, consisting of a finite

derivation tree $\mathscr{P}$ that has non-axiom leaves $S_1,\ldots,S_n$ and a mapping F that assigns internal nodes labels with the same Hoare triples as S_i for each $1 \leq i \leq n$. We write $\mathsf{Bud}(\mathscr{P})$ to denote the set of non-axiom leaves.

By identifying the back-linked nodes, a cyclic pre-proof can be regarded as representing an infinite derivation tree as a cyclic graph. $\mathscr{G}_{\mathscr{P},F}$ denotes the graph of a cyclic pre-proof $(\mathscr{P},F)$. The nodes of $\mathscr{G}_{\mathscr{P},F}$ consist of pairs (S,r), where S is a Hoare triple appearing in the cyclic pre-proof $(\mathscr{P},F)$ and r is the name of the inference rule that concludes S. In the case $S \in \mathsf{Bud}(\mathscr{P})$, we set $r = \emptyset$. $\mathscr{G}_{\mathscr{P},F}$ has the following two types of directed edges: a edge from (S,r) to $(F(S),r')$ for $S \in \mathsf{Bud}(\mathscr{P})$ and a edge from (S,r) to (S',r') for $S \notin \mathsf{Bud}(\mathscr{P})$ and each premise S' of the rule instance r.

Cyclic pre-proofs are considered valid only if they satisfy the following soundness condition.

Definition 3.2 (Cyclic proof)**.** A cyclic pre-proof $(\mathscr{P},F)$ is a cyclic proof if every infinite path $\boldsymbol{\nu}$ in $\mathscr{G}_{\mathscr{P},F}$ contains infinitely many Symbolic Execution Proof Rules.

3.2 An example

We present an example of proof in our system. Consider the program, $y := \mathsf{nil}; \mathit{Reverse}$, which reverses a linear list. Here, the procedure *Reverse* is defined as follows.

$$\texttt{Procedure}\ \mathit{Reverse}\ \{\texttt{if}\ (x \neq \mathsf{nil})\ \{z := [x]; [x] := y; y := x; x := z; \mathit{Reverse}\}\ \texttt{else}\ \varepsilon\ \texttt{fi}\ \}$$

Now, let us consider proving the Hoare triple $\{\mathsf{List}\ \alpha_0(x)\}\ y := \mathsf{nil}; \mathit{Reverse}\ \{\mathsf{List}\ \alpha_0^{\dagger}(y)\}$, where the List predicate is defined inductively as follows.

$$\mathsf{List}\ \varepsilon\ (x) := x = \mathtt{nil} \wedge \mathtt{emp} \qquad \mathsf{List}\ (a \cdot \alpha)\ (x) := x = a \wedge \exists z.x \mapsto z * \mathsf{List}\ \alpha\ (z)$$

Here, $\alpha^{\dagger}$ denotes the sequence obtained by reversing the sequence of addresses α, ε denotes an empty sequence, and $a \cdot \alpha$ denotes the concatenation of the singleton sequence with the sequence α.

We present the proof in Figure 5. The rules highlighted in red in Figure 5 are the Symbolic Execution Proof Rules. Therefore, the global trace condition is satisfied. Symbolic Execution Proof Rules are applied automatically to the program's leading commands. In this example, the (CONSEQ) rule simply transforms the assertion in the precondition according to the inductive definition of the predicate. Thus, we can construct the cyclic proof almost automatically.

4 Soundness

In this section, we demonstrate the soundness of our cyclic proof system. As a preparation, we define the notion of n-invalid for a non-negative integer n as follows.

Definition 4.1 (n-invalid)**.** $\{\phi\}\ C\ \{\psi\}$ is said to be *n-invalid* if, there exists a model (s,h) such that $(s,h) \models \phi$ and either there exists a final state (s',h') such that $(C,s,h) \overset{n}{\rightsquigarrow} (s',h')$ and $(s',h') \not\models \psi$ or $(C,s,h) \overset{n}{\rightsquigarrow} \mathit{abort}$ holds.

To prove the soundness of our system, we first establish the following proposition.

Proposition 4.1 (Local soundness)**.** For a rule instance r, let S be the conclusion of r, and assume that S is n-invalid. Then, there exists a premise S' of r that is m-invalid for some $m \leq n$. In particular, if r is a Symbolic Execution Rule, there exists a premise S' of r that is m-invalid for some $m < n$.

From the above, we obtain the following.

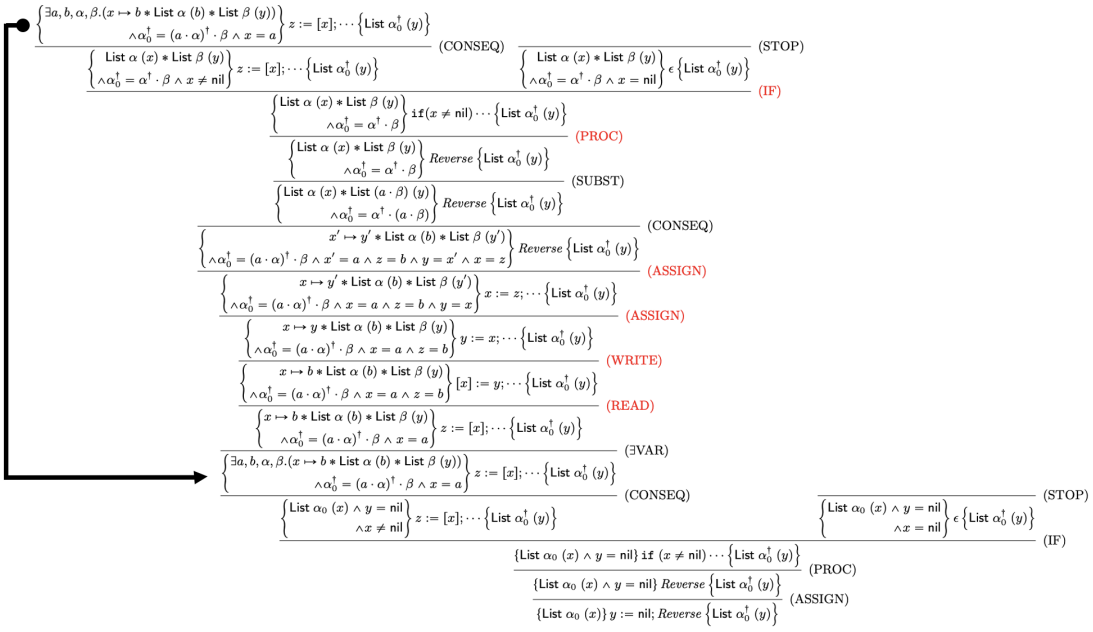


Figure 5: A Cyclic proof of reversing list program

Theorem 4.2 (Soundness)**.** *If* $\vdash \{\phi\}\, C\, \{\psi\}$ *has a cyclic proof, then* $\{\phi\}\, C\, \{\psi\}$ *is valid.*

Proof. We assume that $\vdash \{\phi\}\, C\, \{\psi\}$ has a cyclic proof $(\mathscr{P}, F)$ and $\{\phi\}\, C\, \{\psi\}$ is invalid. Obviously, $\{\phi\}\, C\, \{\psi\}$ is n-invalid for some n. By Proposition 4.1, there exists an infinite path $\boldsymbol{\nu}$ in the cyclic proof $(\mathscr{P}, F)$ of invalid triples starting from $\{\phi\}\, C\, \{\psi\}$. By the global trace condition, the Symbolic Execution Proof Rules must be applied infinitely many times in $\boldsymbol{\nu}$. Here, by Proposition 4.1, an infinite descent of n occurs in the infinite path $\boldsymbol{\nu}$, leading to a contradiction.

□

5 Relative Completeness

In this section, we demonstrate the relative completeness of our system by showing that proofs in Al Ameen and Tatsuta's separation logic system [1], whose relative completeness has already been established, can be transformed into proofs in our system. In this system, Hoare triples are extended to contextual Hoare triples. From now on, we will call this system Al Ameen's system.

Due to space limitations, we omit the inference rules of Al Ameen's system (see the original paper for details [1]). When ϕ is provable in the context Γ by Al Ameen's system, we write $\Gamma \vdash_A \phi$.

The commands of our language and those of Al Ameen's system are slightly different. We define C^* for a command in Al Ameen's system as $(C_1; C_2)^* = C_1^* \fatsemi C_2^*$. For Hoare triples, define $(\{\phi\}\, C\, \{\psi\})^*$ as $\{\phi\}\, C^*\, \{\psi\}$.

In Al Ameen's system, commands are interpreted by a denotational semantics $[\![C]\!]$, which maps from heap models to sets of heap models or *abort*. It is easy to see that their denotational semantics and our operational semantics are equivalent in the following sense: $\kappa \in [\![C]\!](s,h)$ iff $(C^*, s, h) \overset{*}{\rightsquigarrow} \kappa$. Hence, the validity of Hoare triples is equivalent in the two systems.

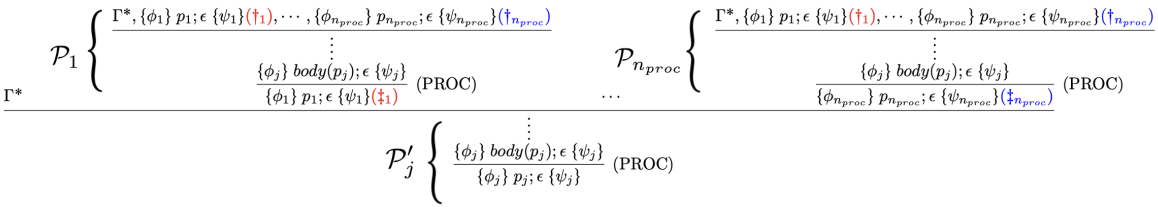


Figure 6: Grafted proof tree $\mathscr{P}$

To define the translation from Al Ameen's proofs to our proofs, we slightly extend our proof system with open assumptions.

Definition 5.1 (Cyclic proof with open assumptions)**.** A *cyclic pre-proof with open assumptions* is defined as a tuple $(\mathscr{P}, F, \Gamma)$ such that $\mathscr{P}$ is a derivation tree, F is a mapping, and Γ is a set of open assumptions satisfying (1) each leaf γ of $\mathscr{P}$ is either an axiom, in Bud, or in Γ, and (2) the mapping F assigns an internal node to each $\gamma \in \mathsf{Bud}$ such that γ and $F(\gamma)$ are the same Hoare triples. A cyclic pre-proof with open assumptions $(\mathscr{P}, F, \Gamma)$ is a cyclic proof with open assumptions if every infinite path $\boldsymbol{\nu}$ in $\mathscr{G}_{\mathscr{P},F}$ contains infinitely many Symbolic Execution Rules. When there exists a cyclic proof of ϕ with open assumptions Γ, we write $\Gamma \vdash_S \phi$.

Theorem 5.1. *For any proof of* $\Gamma \vdash_A \gamma$*, there exists a proof of* $\Gamma^* \vdash_S \gamma^*$ *in our system. We show that each rule in [1] can be proven by our system.*

Proof. We proceed by induction on the rules. In the following, we present only the rule (RECURSION) of Al Ameen's system.

$$\dfrac{\begin{array}{c}\Gamma \cup \{\{\phi_i\}\ p_i\ \{\psi_i\} \mid 1 \le i \le n_{proc}\} \vdash_A \{\phi_1\}\ body(p_1)\ \{\psi_1\} \\ \vdots \\ \Gamma \cup \{\{\phi_i\}\ p_i\ \{\psi_i\} \mid 1 \le i \le n_{proc}\} \vdash_A \{\phi_{n_{proc}}\}\ body(p_{n_{proc}})\ \{\psi_{n_{proc}}\}\end{array}}{\Gamma \vdash_A \{\phi_j\}\ p_j\ \{\psi_j\}}$$

for $1 \le j \le n_{proc}$. Henceforth, we fix j in our considerations, and we show that $\Gamma^* \vdash_S \{\phi_j\}\ p_j; \varepsilon\ \{\psi_j\}$ holds. By the induction hypothesis and the rule (PROC), we have

$$\Gamma^* \cup \{\{\phi_i\}\ p_i; \varepsilon\ \{\psi_i\} \mid 1 \le i \le n_{proc}\} \vdash_S \{\phi_k\}\ p_k; \varepsilon\ \{\psi_k\}.$$

for $1 \le k \le n_{proc}$. Let $(\mathscr{P}_k, F_k, \Gamma^* \cup \{\{\phi_i\}\ p_i; \varepsilon\ \{\psi_i\} \mid 1 \le i \le n_{proc}\})$ be its cyclic proof with open assumptions.

To construct $(\mathscr{P}, F, \Gamma^* \cup \{\{\phi_i\} p_i; \varepsilon \{\psi_i\} \mid i = 1, \ldots, n_{proc}\})$, we consider the following proof tree, as shown in Figure 6, where $\mathscr{G}_{\mathscr{P}_i,F_i}$ is grafted onto each open assumption $\{\phi_i\}\ p_i; \varepsilon\ \{\psi_i\}$ of $\mathscr{G}_{\mathscr{P}_j,F_j}$. As in Figure 6, $\mathscr{P}'_j$ denotes the portion of the original $\mathscr{P}_j$ to distinguish from the grafted tree. Let this tree be $\mathscr{P}$. Let F be a mapping extending $\bigcup_i F_i$ such that, for each open assumption $\{\phi_i\}\ p_i; \varepsilon\ \{\psi_i\}$ assigns the root node of $\mathscr{P}_i$, that is, $F(\dagger_i) = \ddagger_i$ in Figure 6 for each i. In the case of mutually recursive procedures, a single grafting step may not suffice: if not all assumptions in the context are used, one graft operation will not bring in the corresponding subproofs. However, we can repeat this grafting as many times as needed until the necessary assumption appears, and the entire process terminates after at most finitely many iterations.

We will show that the pre-proof satisfies the global trace condition. For an infinite path in the pre-proof, if a tail of the path is within $\mathscr{G}_{\mathscr{P}_i,\mathscr{F}_i}$ for some i, it contains infinitely many Symbolic Execution Proof Rules by the global trace condition for $(\mathscr{P}_i, F_i)$. Otherwise, the infinite path goes through the new back-links between $\mathscr{P}_i$'s infinitely many times. By the construction, the transitions after the new back-links must be the rule (PROC), and hence, the path contains infinitely many Symbolic Execution Proof Rules. □

Theorem 5.2 (Relative Completeness). *If* $\{\phi\}\ C\ \{\psi\}$ *is valid,* $\vdash_S \{\phi\}\ C\ \{\psi\}$ *is provable.*

Proof. If $\{\phi\}\ C\ \{\psi\}$ is valid, $\vdash_A \{\phi\}\ C\ \{\psi\}$ is provable by the relative completeness of Al Ameen's system [1]. From Theorem 5.1, there exists a transformation of the proof of $\vdash_A \{\phi\}\ C\ \{\psi\}$ into the proof of $\vdash_S \{\phi\}\ C^*\ \{\psi\}$, where $C^* = C$ holds for the command C in our language. □

6 Conclusions and Future Work

In this paper, we have proposed a cyclic proof system for partial correctness of separation logic and demonstrated its soundness and relative completeness. For the proof of the relative completeness, we have shown a transformation from proofs in Al Ameen's system [1], which has already been established to be relatively complete, to proofs in our system.

Concurrent separation logic [2, 9] extends separation logic for concurrent programs. In concurrent separation logic, it is possible to prove specifications when multiple threads operate concurrently. In Brotherston's system SL_{LP} [4], the access rights that a thread holds over a specific region of shared memory are expressed in greater detail using fractional permissions. The first challenge is extending our proposed cyclic proof system for separation logic by incorporating permission values to handle concurrent separation logic. All variables are global in our system, and local variables and function parameters are absent. The second challenge is to extend our system by introducing local variables and function parameters to accommodate a broader class of programs.

References

[1] Mahmudul Faisal Al Ameen & Makoto Tatsuta (2016): *Completeness for recursive procedures in separation logic*. Theoretical Computer Science 631, pp. 73–96.

[2] Stephen Brookes (2007): *A semantics for concurrent separation logic*. Theoretical Computer Science 375(1-3), pp. 227–270.

[3] James Brotherston, Richard Bornat & Cristiano Calcagno (2008): *Cyclic proofs of program termination in separation logic*. ACM SIGPLAN Notices 43(1), pp. 101–112.

[4] James Brotherston, Diana Costa, Aquinas Hobor & John Wickerson (2020): *Reasoning over Permissions Regions in Concurrent Separation Logic*. In: International Conference on Computer Aided Verification, Springer, pp. 203–224.

[5] James Brotherston & Nikos Gorogiannis (2014): *Cyclic abduction of inductively defined safety and termination preconditions*. In: International Static Analysis Symposium, Springer, pp. 68–84.

[6] James Brotherston, Nikos Gorogiannis & Rasmus L Petersen (2012): *A generic cyclic theorem prover*. In: Asian Symposium on Programming Languages and Systems, Springer, pp. 350–367.

[7] James Brotherston & Alex Simpson (2011): *Sequent calculi for induction and infinite descent*. Journal of Logic and Computation 21(6), pp. 1177–1216.

[8] Charles Antony Richard Hoare (1969): *An axiomatic basis for computer programming*. Communications of the ACM 12(10), pp. 576–580.

[9] Peter W O'Hearn (2007): *Resources, concurrency, and local reasoning*. Theoretical computer science 375(1), pp. 271–307.

[10] John C Reynolds (2002): *Separation logic: A logic for shared mutable data structures*. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, IEEE, pp. 55–74.

[11] Reuben NS Rowe & James Brotherston (2017): *Automatic cyclic termination proofs for recursive procedures in separation logic*. In: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, pp. 53–65.

[12] David Von Oheimb (1999): *Hoare logic for mutual recursion and local variables*. In: Foundations of Software Technology and Theoretical Computer Science: 19th Conference Chennai, India, December 13-15, 1999 Proceedings 19, Springer, pp. 168–180.

On Transforming Prioritized Multithreaded Programs into Logically Constrained Term Rewrite Systems*

Misaki Kojima
Nagoya University
Nagoya, Japan
kojima@i.nagoya-u.ac.jp

Naoki Nishida
Nagoya University
Nagoya Japan
nishida@i.nagoya-u.ac.jp

A transformation of multithreaded programs without priorities into logically constrained term rewrite systems (LCTRS, for short) has been proposed. In this paper, we extend it to prioritized threads by introducing a descending list of priorities into terms representing configurations. As a case study, we show a verification example in which a runtime-error verification method based on all-path reachability detects an assertion failure caused by the interrupt of a high-priority thread.

1 Introduction

Recently, approaches to program verification by means of *logically constrained term rewrite systems* (LCTRSs, for short) [15] are well investigated [5, 21, 3, 19, 6, 7, 2, 17, 14]. LCTRSs are useful models of not only functional but also imperative programs, which are evaluated sequentially. For instance, equivalence checking by means of LCTRSs is useful to ensure the correctness of terminating functions (cf. [5]). Since the reduction of rewrite systems is in general non-deterministic, rewrite systems are reasonable models of concurrent programs. A transformation of sequential programs into LCTRSs [5, 6] has been extended to multithreaded programs[1] without priorities [13, 14]. In addition, a method for runtime-error verification by means of *all-path reachability problem* (APR problem, for short) of LCTRSs has been developed [10, 9]. The method is expected to be applied to the verification of various runtime errors.

Experience has shown that runtime errors caused by *interrupts* are very crucial for concurrent programs, and verification for such errors—detecting an execution with a runtime error, or proving non-occurrence of runtime errors in any execution—is very important. To the best of our knowledge, there is no transformation of prioritized multithreaded programs into constrained rewrite systems, and thus, for the present, we cannot apply the verification method based on APR problems of LCTRSs to runtime errors caused by interrupts.

In this paper, we extend the aforementioned transformation of multithreaded programs into LCTRSs to prioritized threads by introducing a descending list of priorities into terms representing configurations (Section 4). We deal with a restricted class of C programs with the Pthreads Library for prioritized threads. Then, as a case study, we show a verification example in which the APR-based method detects an *assertion failure* caused by the interrupt of a high-priority thread (Section 5). As a first step of extending to priorities, we consider *round-robin scheduling* `SCHED_RR` of the Pthreads Library only, where we consider arbitrary time periods so as to verify all possibles settings. Assertions which specify

*This work was partially supported by Grant-in-Aid for JSPS Fellows Grant Number JP24KJ1240 and JSPS KAKENHI Grant Number.

[1] Although source programs in [13, 14] are called multiprocessing, they are actually multithreaded because processes can share some global variables.

assumptions to be satisfied in passing the assertions are often written in programs to be verified: If an assertion is not satisfied, then the assertion failure happens as a runtime error. As runtime errors, we mainly focus on assertion failures.

When considering interrupts, threads have priorities. A thread can be executed if its priority is the highest among those of the currently runnable threads; otherwise, the thread waits for the termination of other runnable threads with higher priorities. To control execution order based on priorities, we need to keep track of the priorities of the currently runnable threads and know the highest one among the priorities. To this end, we use a descending list of priorities of the currently runnable threads. The head priority of the list indicates which priority is the highest at the moment. Thus, the descending list is operated as a stack. The descending list performing as a stack is operated as follows:

- When a thread starts to be executed, for interruption, its priority is pushed to the list;
- a thread can be executed if its priority is greater than or equal to[2] the head element of the stack;
- when a thread terminates successfully, for other threads with lower priorities, the head element of the stack is popped.

Since there may be two or more runnable threads with the same priority, the descending list may have duplicated priorities like multisets; if we do not allow any duplicated priorities and the head element is popped for termination of a thread, then no other thread with the same priority can no longer be executed. Since round-robin scheduling is considered, we do not have to keep process identifiers together with priorities; for FIFO scheduling,[3] we need a queue of process identifiers, i.e., a descending list of pairs of priorities and identifiers. A thread can be executed if its priority is the highest, i.e., greater than or equal to the head element of the descending list. When we start the execution of a thread, its constrained rewrite rule pushes the priority to the descending list. When the thread terminates after the application of a constrained rewrite rule, the priority which is located on the top of the list is popped by the rewrite rule.

In [14], the use of lists for waiting queues of semaphores is not recommended so as to reduce the number of generated constrained rewrite rules. This is because such queues need an operation recursively represented by rewrite rules, which appends a process identifier to a queue as the last element by several reduction steps. Such steps expand the search space in verifying non-existence of runtime errors. On the other hand, we use lists for stacks of priorities in our transformed LCTRSs. Such use is not a downside because the push and pop operations can be included in rewrite rules for the transition of configurations such as $\mathsf{cnfg}(\ldots, ls) \to \mathsf{cnfg}(\ldots, p{::}ls)\ [\varphi]$ and $\mathsf{cnfg}(\ldots, p{::}ls) \to \mathsf{cnfg}(\ldots, ls)\ [\varphi]$, where ls is a variable for a stack and p is an element pushed to or popped from the stack.

2 Preliminaries

In this section, we briefly recall LCTRSs [15, 5]. Familiarity with basic notions and notations on term rewriting [1, 20] is assumed.

To define an LCTRS [15, 5] over an $\mathcal{S}$-sorted signature Σ, we consider the following sorts, signatures, mappings, and constants: *Theory sorts* in $\mathcal{S}_{theory}$ and *term sorts* in $\mathcal{S}_{term}$ such that $\mathcal{S} = \mathcal{S}_{theory} \uplus \mathcal{S}_{term}$; a *theory signature* Σ_{theory} and a *term signature* Σ_{terms} such that $\Sigma = \Sigma_{theory} \cup \Sigma_{terms}$ and $\iota_1, \ldots, \iota_n, \iota \in \mathcal{S}_{theory}$

[2] If the generated LCTRS performs correctly, then the priority coincides with the head element.

[3] A thread can be executed until its completion if there is no interruption. For successive execution of a thread without interruption, we have to keep the thread identifier, together with priorities.

for any symbol $f : \iota_1 \times \cdots \times \iota_n \to \iota \in \Sigma_{theory}$; a mapping $\mathcal{I}$ that assigns to each theory sort ι a (non-empty) set $\mathcal{A}_\iota$, so-called the universe of ι (i.e., $\mathcal{I}(\iota) = \mathcal{A}_\iota$); a mapping $\mathcal{J}$, so-called an interpretation for Σ_{theory}, that assigns to each function symbol $f : \iota_1 \times \cdots \times \iota_n \Rightarrow \iota \in \Sigma_{theory}$ a function $f^{\mathcal{J}}$ in $\mathcal{I}(\iota_1) \times \cdots \times \mathcal{I}(\iota_n) \to \mathcal{I}(\iota)$ (i.e., $\mathcal{J}(f) = f^{\mathcal{J}}$); a set $\mathcal{V}al_\iota \subseteq \Sigma_{theory}$ of *value-constants* $a : \iota$ for each theory sort ι such that $\mathcal{J}$ gives a bijection from $\mathcal{V}al_\iota$ to $\mathcal{I}(\iota)$ $(= \mathcal{A}_\iota)$. We denote $\bigcup_{\iota \in \mathcal{S}_{theory}} \mathcal{V}al_\iota$ by $\mathcal{V}al$. Note that $\mathcal{V}al \subseteq \Sigma_{theory}$. For readability, we may not distinguish $\mathcal{V}al_\iota$ and $\mathcal{I}(\iota)$ $(= \mathcal{A}_\iota)$, i.e., for each $v \in \mathcal{V}al_\iota$, v and $\mathcal{J}(v)$ may be identified. We require that $\Sigma_{terms} \cap \Sigma_{theory} \subseteq \mathcal{V}al$. Symbols in $\Sigma_{theory} \setminus \mathcal{V}al$ are *calculation symbols*, for which we may use infix notation. A term in $T(\Sigma_{theory}, \mathcal{V})$ is called a *theory term*, where $\mathcal{V}$ is a (countably infinite) set of variables. We define the *interpretation* $[\![\cdot]\!]_{\mathcal{J}}$ of ground theory terms as $[\![f(s_1, \ldots, s_n)]\!]_{\mathcal{J}} = \mathcal{J}(f)([\![s_1]\!]_{\mathcal{J}}, \ldots, [\![s_n]\!]_{\mathcal{J}})$. Note that for every ground theory term s, there is a unique value-constant c such that $[\![s]\!]_{\mathcal{J}} = [\![c]\!]_{\mathcal{J}}$.

We typically choose a theory signature $\mathcal{S}_{theory}$ such that $\mathcal{S}_{theory} \supseteq \mathcal{S}_{core} = \{bool\}$, $\mathcal{V}al_{bool} = \{\mathsf{true}, \mathsf{false} : bool\}$, $\Sigma_{theory} \supseteq \Sigma_{core} = \mathcal{V}al_{bool} \cup \{\wedge, \vee, \Rightarrow, \Leftrightarrow : bool \times bool \Rightarrow bool,\ \neg : bool \Rightarrow bool\} \cup \{=_\iota, \neq_\iota : \iota \times \iota \Rightarrow bool \mid \iota \in \mathcal{S}_{theory}\}$, $\mathcal{I}(bool) = \{\top, \bot\}$, and $\mathcal{J}$ interprets these symbols as expected: $\mathcal{J}(\mathsf{true}) = \top$ and $\mathcal{J}(\mathsf{false}) = \bot$. We omit the sort subscripts ι from $=_\iota$ and $\neq_\iota$ when they are clear from the context. A theory term with sort *bool* is called a *constraint*. A substitution γ which is a sort-preserving mapping from $T(\Sigma, \mathcal{V})$ to $T(\Sigma, \mathcal{V})$ is said to *respect* a constraint ϕ if $x\gamma \in \mathcal{V}al$ for all $x \in \mathcal{V}ar(\phi)$ and $[\![\phi\gamma]\!]_{\mathcal{J}} = \top$, where $\mathcal{V}ar(\phi)$ denotes the set of variables appearing in ϕ.

A *constrained rewrite rule* is a triple $\ell \to r\ [\varphi]$ such that ℓ and r are terms of the same sort, φ is a constraint, and ℓ is neither a theory term nor a variable. If $\varphi = \mathsf{true}$, then we may write $\ell \to r$. We define $\mathcal{LV}ar(\ell \to r\ [\varphi])$ as $\mathcal{V}ar(\varphi) \cup (\mathcal{V}ar(r) \setminus \mathcal{V}ar(\ell))$, the set of *logical variables in* $\ell \to r\ [\varphi]$ which are variables instantiated with values in rewriting terms. We say that a substitution γ *respects* $\ell \to r\ [\varphi]$ if $\gamma(x) \in \mathcal{V}al$ for all $x \in \mathcal{LV}ar(\ell \to r\ [\varphi])$ and $[\![\varphi\gamma]\!]_{\mathcal{J}} = \top$. Regarding the signature of $\mathcal{R}$, we denote the set $\{f(x_1, \ldots, x_n) \to y\ [y = f(x_1, \ldots, x_n)] \mid f \in \Sigma_{theory} \setminus \mathcal{V}al,\ x_1, \ldots, x_n, y \in \mathcal{V} \text{ are pairwise distinct}\}$ by $\mathcal{R}_{calc}$. The elements of $\mathcal{R}_{calc}$ are *calculation rules* and we often deal with them as constrained rewrite rules even though their left-hand sides are theory terms. The *rewrite relation* $\to_{\mathcal{R}}$ is a binary relation over terms, defined as follows: For a term s, $s[\ell\gamma]_p \to_{\mathcal{R}} s[r\gamma]_p$ if and only if $\ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc}$ and γ respects $\ell \to r\ [\varphi]$. A reduction step with $\mathcal{R}_{calc}$ is called a *calculation*. A *logically constrained term rewrite system* (LCTRS, for short) is defined as an abstract reduction system $(T(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$, simply denoted by $\mathcal{R}$, where $\mathcal{R}$ is a set of constrained rewrite rules. An LCTRS is usually given by supplying Σ, $\mathcal{R}$, and an informal description of $\mathcal{I}$ and $\mathcal{J}$ if these are not clear from the context.

The *standard integer signature* Σ_{int} is $\Sigma_{core} \cup \{+, -, \times, \mathsf{exp}, \mathsf{div}, \mathsf{mod} : int \times int \Rightarrow int\} \cup \{\geq, > : int \times int \Rightarrow bool\} \cup \mathcal{V}al_{int}$ where $\mathcal{S}_{theory} \supseteq \{int, bool\}$, $\mathcal{V}al_{int} = \{\mathsf{n} \mid n \in \mathbb{Z}\}$, $\mathcal{I}(int) = \mathbb{Z}$, and $\mathcal{J}(\mathsf{n}) = n$ for any $n \in \mathbb{Z}$—we use n (in sans-serif font) as the value-constant for $n \in \mathbb{Z}$ (in *math* font). We define $\mathcal{J}$ in a natural way. An LCTRS over a signature $\Sigma \supseteq \Sigma_{int}$ with $\Sigma_{theory} = \Sigma_{int}$ is called an *integer LCTRS*.

Example 2.1 The following integer LCTRS calculates the *factorial* function over $\mathbb{Z}$ iteratively:

$$\mathcal{R}_1 = \left\{ \begin{array}{ll} \mathsf{fact}(x) \to \mathsf{subfact}(x, 1) & \\ \mathsf{subfact}(x, y) \to y & [x \leq 0] \\ \mathsf{subfact}(x, y) \to \mathsf{subfact}(x - 1, x \times y) & [x > 0] \end{array} \right\}$$

The term $\mathsf{fact}(3)$ is reduced by $\mathcal{R}_1$ to 6: $\mathsf{fact}(3) \to_{\mathcal{R}_1} \mathsf{subfact}(3,1) \to_{\mathcal{R}_1} \mathsf{subfact}(3-1, 3\times 1) \to_{\mathcal{R}_1} \mathsf{subfact}(2, 3\times 1) \to_{\mathcal{R}_1} \mathsf{subfact}(2-1, 2\times(3\times 1)) \to_{\mathcal{R}_1} \mathsf{subfact}(1, 2\times(3\times 1)) \to_{\mathcal{R}_1} \mathsf{subfact}(1-1, 1\times(2\times(3\times 1))) \to_{\mathcal{R}_1} \mathsf{subfact}(0, 1\times(2\times(3\times 1))) \to_{\mathcal{R}_1} 1\times(2\times(3\times 1)) \to^*_{\mathcal{R}_1} 6$.

3 Source Programs

For brevity, as an abstract source language, we use a concurrent version of SIMP$^+$ [6][4] with priority of threads, programs of which are written in the C language with the Pthreads Library. To be more precise, the following are allowed in our source programs.

Declarations A program consists of declarations of functions and global variables and has a `main` function that only creates a fixed number of threads having individual priorities. To compile it as a C program, it includes `pthread.h`, `unistd.h`, and `assert.h`, while for the space issue, we omit the `#include` sentences from programs in this paper. We include `unistd.h` to use `usleep` for experiments, and include `assert.h` to use assertions for verification.

Types Types `int` for integers, `pthread_t` for threads, `pthread_attr_t` for attributes of threads, and `struct sched_param` for specifying scheduling parameters are allowed. As a first step, for brevity, we use `pthread_attr_t` only to specify a scheduling parameter. For the sake of simplicity and readability, unlike the C language, the range of variables with type `int` is not the 32-bit integers but the integers. This restriction is not essential for the result in this paper.

Variables Both global and local variables are used with initialized declarations, while global variables are allowed for `int` only. Local variables with types `pthread_t`, `pthread_attr_t`, and `struct sched_param` are only used in the `main` function to create threads.

Functions The `main` function is defined as the form `int main() { ... return 0; }`. Other functions are defined as either of the following forms:

- `void` f`(void *` x`) { ... }`
- `int` f`(int` x_1`, ... , int` x_n`) { ... }`

Since we follow the specification of the Pthreads Library, any function f called in creating threads takes an argument with type `void *`. The `main` function in this paper just creates some threads, waits for the termination of the created threads, and then finishes its execution.

Statements for threads with priority attributes We only use the following five statements for threads with priority attributes in some restricted way:

- `pthread_create(&`t`,` e`,` f`, NULL);` for creating threads, where t is a variable with type `pthread_t` and e is either `NULL` or `&`a with a a variable with type `pthread_attr_t`.
- `pthread_attr_init(&`a`);` for initializing attributes, where a is a variable with type `pthread_attr_t`,
- p`.sched_priority =` n`;` for setting a priority to a scheduling parameter, where p is a variable with type `struct sched_param` and n is a positive integer,
- `pthread_attr_setschedpolicy(&`a`, SCHED_RR);` for setting the round-robin scheduling policy `SCHED_RR` to a thread attribute, where a is a variable with type `pthread_attr_t`, and
- `pthread_attr_setschedparam(&`a`, &`p`);` for setting a scheduling parameter to a thread attribute, where a is a variable with type `pthread_attr_t` and p is a variable with type `struct sched_param`.

The above statements are used in the `main` function only.

Statements In addition to assignments and `if`, `while`, `return` statements over `int` expressions, function calls for user-defined ones and `usleep` are allowed.

[4] SIMP$^+$ is an extension of a small imperative language, so-called SIMP [4], to global variables and function calls.

Expressions Integer expressions with no side effects are used.

Assertions Assertions of the form `assert(`e`);` are used for verification.

Example 3.1 Program 1 is a program with medium and high-priority interrupts. The `main` function creates three threads: `task` function waits until the value of `init` becomes 1 and loads the value of the shared variable `x`; `mid` function interrupts `task`, and assigns 99 to `x` and 1 to `init`; `high` function interrupts `task` or `mid` and increments `x` by 1. The function `task` is expected to load the value written by the medium-priority interrupt. However, depending on the timing of the interrupt, the value may be incremented by the high-priority interrupt.

4 Transforming Source Programs into LCTRSs

Introducing the representation of priorities into the transformation in [14], we transform our source programs into LCTRSs, while we do not consider semaphores and their waiting queues and some minor changes are introduced. The sorts of the generated LCTRSs are *bool*, *int*, *config*, *thread*, and *state*. The fixed non-theory function symbols are the following:

- $\mathsf{cnfg} : \mathit{thread} \times \cdots \times \mathit{thread} \times \mathit{int} \times \cdots \times \mathit{int} \Rightarrow \mathit{config}$ for configurations,
- $\mathsf{t} : \mathit{state} \times \mathit{int} \Rightarrow \mathit{thread}$ for threads,
- $\mathsf{none} : \mathit{thread}$ for the non-existence of threads,
- $\mathsf{return} : \mathit{thread}$ for the terminating state of threads, and
- $\mathsf{error} : \mathit{config}$ for runtime errors.

The `main` function considered in this paper creates n threads and ends with the termination of the created threads. Since there is a root thread executing the `main` function, the number of the arguments of cnfg is $n+m+2$: $n+1$ threads, m global variables, and a stack of priorities. Rewrite rules for creating threads are similar to those for creating processes in [14].

A thread T is represented as a term $\mathsf{t}(st, p)$ where the subterm st is a state of T and p is its priority. The priorities of the currently runnable threads are maintained in a descending list that is operated as a stack. Configurations of the executing process with n threads and m global variables are represented by terms of the following form:

$$\mathsf{cnfg}(\mathsf{t}(st_1, p_1), \ldots, \mathsf{t}(st_n, p_n), v_1, \ldots, v_m, stck)$$

where st_i is either a term representing the entry point or a state of the i-th thread, v_j is a value stored for the j-th global variable, and $stck$ is a descending list for priorities of executing threads. A thread can be executed if its priority is greater than or equal to the head value of the descending list. For lists in LCTRSs, we use the list constructors :: and nil.

Following the approach in [14], we transform a statement of the function executed in the i-th thread into constrained rewrite rules. Each constrained rewrite rule represents a single reduction step of exactly one thread, having the following form:

$$\mathsf{cnfg}(t_1, \ldots, t_{i-1}, \mathsf{t}(st, p), t_{i+1}, \ldots, t_n, x_1, \ldots, x_m, s) \to \mathsf{cnfg}(t_1, \ldots, t_{i-1}, \mathsf{t}(st', p), t_{i+1}, \ldots, t_n, x'_1, \ldots, x'_m, s') \ [\varphi]$$

where $t_1, \ldots, t_{i-1}, p, t_{i+1}, \ldots, t_n, x_1, \ldots, x_m$ are pairwise different variables, st and st' are terms representing states just before and after executing the statement or a function, respectively, s and s' are terms representing stacks for priorities, φ is the conjunction of a guard constraint and a formula to update variables by

Program 1: Program with medium and high priority interrupts

```
1  int x = 0;
2  int init = 0;
3
4  void* task(void* arg) {
5      while (init == 0) usleep(100);
6      int v = x;
7      assert(v == 99);
8      return NULL;
9  }
10
11 void* mid(void* arg) {
12     x = 99;
13     init = 1;
14     return NULL;
15 }
16
17 void* high(void* arg) {
18     x ++;
19     return NULL;
20 }
21
22 int main() {
23     pthread_t thread_task, thread_mid, thread_high;
24     pthread_attr_t attr_task, attr_mid, attr_high;
25     struct sched_param param_task, param_mid, param_high;
26
27     pthread_attr_init(&attr_task);
28     pthread_attr_init(&attr_mid);
29     pthread_attr_init(&attr_high);
30
31     param_task.sched_priority = 10;
32     param_mid.sched_priority = 20;
33     param_high.sched_priority = 30;
34
35     pthread_attr_setschedpolicy(&attr_task, SCHED_RR);
36     pthread_attr_setschedpolicy(&attr_mid, SCHED_RR);
37     pthread_attr_setschedpolicy(&attr_high, SCHED_RR);
38
39     pthread_attr_setschedparam(&attr_task, &param_task);
40     pthread_attr_setschedparam(&attr_mid, &param_mid);
41     pthread_attr_setschedparam(&attr_high, &param_high);
42
43     pthread_create(&thread_task, &attr_task, task, NULL);
44     pthread_create(&thread_mid, &attr_mid, mid, NULL);
45     pthread_create(&thread_high, &attr_high, high, NULL);
46
47     return 0;
48 }
```

means of assignments in executing the statement, and $x'_1,\dots,x'_m$ are variables in $\{x_1,\dots,x_m\}\cup\mathcal{V}ar(\phi)$. State terms st and st' are rooted by auxiliary function symbols that represent which statements are executed at the next step. Such symbols may have arguments that store values of local variables. Concrete examples can be seen in Example 4.1.

In addition to the overview above, regarding descending lists of priorities, the entry point, intermediate statements, and `return` statements of the i-th thread are transformed into constrained rewrites rule of the following forms, respectively:

$$\begin{aligned}
&\mathsf{cnfg}(t_1,\dots,t_{i-1},\mathsf{t}(f,p),t_{i+1},\dots,t_n,x_1,\dots,x_m,hd::ls)\\
&\qquad\to \mathsf{cnfg}(t_1,\dots,t_{i-1},\mathsf{t}(st',p),t_{i+1},\dots,t_n,x_1,\dots,x_m,\qquad p::hd::ls)\ [p\geq hd]\\
&\mathsf{cnfg}(t_1,\dots,t_{i-1},\mathsf{t}(st,p),t_{i+1},\dots,t_n,x_1,\dots,x_m,hd::ls)\\
&\qquad\to \mathsf{cnfg}(t_1,\dots,t_{i-1},\mathsf{t}(st',p),t_{i+1},\dots,t_n,x_1,\dots,x_{j-1},x'_j,x_{j+1},\dots,x_m,hd::ls\quad)\ [p\geq hd\wedge\varphi]\\
&\mathsf{cnfg}(t_1,\dots,t_{i-1},\mathsf{t}(st,p),t_{i+1},\dots,t_n,x_1,\dots,x_m,hd::ls)\\
&\qquad\to \mathsf{cnfg}(t_1,\dots,t_{i-1},\mathsf{return},\ t_{i+1},\dots,t_n,x_1,\dots,x_m,\qquad ls\qquad)\ [p\geq hd\wedge\varphi]
\end{aligned}$$

where f is a function executed by the i-th thread and $t_1,\dots,t_{i-1},p,t_{i+1},\dots,t_n,x_1,\dots,x_m,x'_j,hd,ls$ are pairwise different variables. Note that each constrained rewrite rule represents a behavior of exactly one thread. For an `assert` statement of the form `assert(`φ`);`, we generate the following rule for the violation of the assertion φ:

$$\mathsf{cnfg}(t_1,\dots,t_{i-1},\mathsf{t}(st,p),t_{i+1},\dots,t_n,x_1,\dots,x_m,hd::ls)\to\mathsf{error}\ [p\geq hd\wedge\neg\varphi]$$

Since the reduction of LCTRSs is non-deterministic, we do not represent the execution of calling `usleep`, i.e., we consider `usleep(`e`);` the skip statement.

The initial configuration is represented by the following term:

$$\mathsf{cnfg}(\mathsf{t}(\mathsf{main},p_0),\mathsf{none},\dots,\mathsf{none},v_1,\dots,v_m,p_0::0::\mathsf{nil})$$

where p_0 is a priority of the `main` function—a positive integer which is larger than any priority specified in the program—and v_j is the initial value of the j-th global variable. Since the `main` function deterministically performs to create a fixed number of threads, to, e.g., reduce the search space of verification, we may omit the execution of the `main` function in the transformed LCTRS. In such a case, we represent the initial configuration by the following term:

$$\mathsf{cnfg}(\mathsf{t}(f_1,p_1),\dots,\mathsf{t}(f_n,p_n),v_1,\dots,v_m,0::\mathsf{nil})$$

where f_i is a function executed by the i-th thread, p_i is a priority of the i-th thread, and v_j is the initial value of the j-th global variable. Descending lists of configuration terms always include a dummy priority 0, and thus the initial configuration above has 0::nil as the last argument. Due to this dummy priority, we do not have to generate any rewrite rule for the case where the last argument of cnfg is nil. This enables us to reduce the number of generated rules.

Example 4.1 Let us consider Program 1 again. The `main` function creates three threads and ends with the termination of the created threads. For the page limitation, the execution of the `main` function is omitted in the transformed LCTRS, i.e., the execution of `main` is not represented by any constrained rewrite rule. Thus, cnfg has six arguments: three threads, two global variables, and a stack of priorities.

For a thread executing `task`, a state just before executing a statement on Line 7 is represented by a term of the form $\mathsf{task}_7(n)$; On Line 6, the local variable `v` is declared with the initial value stored in the global variable `x`. Program 1 is transformed into the following LCTRS:

$$\mathcal{R}_2=\left\{\begin{array}{rll}
\mathsf{cnfg}(\mathsf{t}(\mathsf{task},p),t_2,t_3,x,i,hd{::}ls)\to\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_5,p),t_2,t_3,x,i,p{::}hd{::}ls) & [p\geq hd &]\\
\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_5,p),t_2,t_3,x,i,hd{::}ls)\to\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_5,p),t_2,t_3,x,i,hd{::}ls) & [p\geq hd\wedge i=0 &]\\
\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_5,p),t_2,t_3,x,i,hd{::}ls)\to\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_6,p),t_2,t_3,x,i,hd{::}ls) & [p\geq hd\wedge i\neq 0 &]\\
\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_6,p),t_2,t_3,x,i,hd{::}ls)\to\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_7(v),p),t_2,t_3,x,i,hd{::}ls) & [p\geq hd\wedge v=x &]\\
\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_7(v),p),t_2,t_3,x,i,hd{::}ls)\to\mathsf{cnfg}(\mathsf{return},t_2,t_3,x,i,ls) & [p\geq hd\wedge v=99 &]\\
\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_7(v),p),t_2,t_3,x,i,hd{::}ls)\to\mathsf{error} & [p\geq hd\wedge v\neq 99 &]\\
\mathsf{cnfg}(t_1,\mathsf{t}(\mathsf{mid},p),t_3,x,i,hd{::}ls)\to\mathsf{cnfg}(t_1,\mathsf{t}(\mathsf{mid}_{12},p),t_3,x,i,p{::}hd{::}ls) & [p\geq hd &]\\
\mathsf{cnfg}(t_1,\mathsf{t}(\mathsf{mid}_{12},p),t_3,x,i,hd{::}ls)\to\mathsf{cnfg}(t_1,\mathsf{t}(\mathsf{mid}_{13},p),t_3,x',i,hd{::}ls) & [p\geq hd\wedge x'=99 &]\\
\mathsf{cnfg}(t_1,\mathsf{t}(\mathsf{mid}_{13},p),t_3,x,i,hd{::}ls)\to\mathsf{cnfg}(t_1,\mathsf{t}(\mathsf{mid}_{14},p),t_3,x,i',hd{::}ls) & [p\geq hd\wedge i'=\mathsf{one} &]\\
\mathsf{cnfg}(t_1,\mathsf{t}(\mathsf{mid}_{14},p),t_3,x,i,hd{::}ls)\to\mathsf{cnfg}(t_1,\mathsf{return},t_3,x,i,ls) & [p\geq hd &]\\
\mathsf{cnfg}(t_1,t_2,\mathsf{t}(\mathsf{high},p),x,i,hd{::}ls)\to\mathsf{cnfg}(t_1,t_2,\mathsf{t}(\mathsf{high}_{18},p),x,i,p{::}hd{::}ls) & [p\geq hd &]\\
\mathsf{cnfg}(t_1,t_2,\mathsf{t}(\mathsf{high}_{18},p),x,i,hd{::}ls)\to\mathsf{cnfg}(t_1,t_2,\mathsf{t}(\mathsf{high}_{19},p),x',i,hd{::}ls) & [p\geq hd\wedge x'{=}x{+}1 &]\\
\mathsf{cnfg}(t_1,t_2,\mathsf{t}(\mathsf{high}_{19},p),x,i,hd{::}ls)\to\mathsf{cnfg}(t_1,t_2,\mathsf{return},x,i,ls) & [p\geq hd &]
\end{array}\right\}$$

where a variable i stands for the global variable `init`. The initial configuration is represented by the following term:

$$\mathsf{cnfg}(\mathsf{t}(\mathsf{task},10),\mathsf{t}(\mathsf{mid},20),\mathsf{t}(\mathsf{high},30),0,0,0{::}\mathsf{nil})$$

5 Verification of Safety Properties by All-Path Reachability

As an application of our transformed LCTRSs, we investigate all-path reachability problems (APR problem, for short) for the verification of runtime-errors [8, 10].

A *constrained term* is a pair $\langle t \mid \phi\rangle$ of a term t and a constraint ϕ, which can be considered the set of all instances of t w.r.t. substitutions that respect ϕ: $\{t\gamma \mid \gamma \text{ respects } \phi\}$. An *APR problem* of an LCTRS $\mathcal{R}$ is a pair $\langle s \mid \phi\rangle \Rightarrow \langle t \mid \psi\rangle$ of constrained terms $\langle s \mid \phi\rangle, \langle t \mid \psi\rangle$ for state sets. As in [10, 9, 14], we consider *constant-directed APR problems* of the form $\langle s \mid \phi\rangle \Rightarrow c$ such that c is an irreducible constant. An APR problem $\langle s \mid \phi\rangle \Rightarrow c$ is *demonically valid* w.r.t. $\mathcal{R}$ if every *finite* execution path—a reduction sequence starting with a term in $\langle s \mid \phi\rangle$ and ending with a terminating state (i.e., a normal form)—includes c. Note that execution paths are reduction sequences of *unconstrained* terms, and constrained terms are used just to represent sets of terms. Note also that a usual reachability problem from a set P of terms to a set Q of terms is whether each term in P reaches a term in Q, i.e., the problem is valid if for each term in P, there *exists* a reduction sequence starting with the term and ending with a term in Q. A proof system for APR problems of LCTRSs, DCC, and its weakened but easily-implementable variant cDCC for constant-directed APR problems have been proposed in [3, 8, 9, 10, 12]. We have implemented the variant proof system cDCC in a prototype of Crisys2,[5] an equivalence verification system based on *constrained rewriting induction* for LCTRSs [16, 5]. The implementation attempts to prove and disprove a given APR problem by constructing a proof tree via breadth-first search.

The assertion failure verification for Program 1—whether there exists an execution violating the assertion `assert(v == 99);` on Line 7—is reduced to the following APR problem of the LCTRS $\mathcal{R}_2^{\forall}$

[5] `https://www.trs.css.i.nagoya-u.ac.jp/crisys/rp2024/`

such that $\mathcal{R}_2^{\forall} = \mathcal{R}_2 \cup \{\mathsf{cnfg}(t_1,t_2,t_3,x,i,ls) \to \mathsf{success}\}$:

$$\langle \mathsf{cnfg}(\mathsf{t}(\mathsf{task},10),\mathsf{t}(\mathsf{mid},20),\mathsf{t}(\mathsf{high},30),0,0,0{::}\mathsf{nil}) \mid \mathsf{true}\rangle \Rightarrow \mathsf{success}$$

Since the non-existence of assertion failures is a safety property, as for race condition in [10], the above APR problem is demonically valid if and only if there is no execution path where the assertion on Line 7 is violated.

Regarding the initial term $\langle \mathsf{cnfg}(\mathsf{t}(\mathsf{task},10),\mathsf{t}(\mathsf{mid},20),\mathsf{t}(\mathsf{high},30),0,0,0{::}\mathsf{nil}) \mid \mathsf{true}\rangle$, each thread can be executed first, and no interrupt happens until a thread with a higher priority starts to be executed. For this observation, this APR problem is not demonically valid: There exists at least one execution path with an assertion failure. Using the tool Crisys2, we succeeded in disproving the above APR problem for $\mathcal{R}_2^{\forall}$, which was conducted within a 3,600s timeout on a machine running MacOS 15.4.1 on Apple M2 8 cores with 24GB memory; Z3 (ver. 4.13.3) [18] was used as an external SMT solver. The APR problem for $\mathcal{R}_2^{\forall}$ was expectedly disproved by Crisys2 in 3.867s.

In verifying assertion failures by means of APR problems, when a source program includes exactly one assertion that is executed at most once in any execution, we do not have to take care of the execution after successfully passing the assertion. Viewed in this light, we can replace the right-hand side of the rule for the assertion by success. For example, the `assert` statement on Line 7 of Program 1 is executed at most once, and thus $\mathcal{R}_2$ can be modified to the following one:

$$\mathcal{R}_3 = \left\{ \begin{array}{lll}
\mathsf{cnfg}(\mathsf{t}(\mathsf{task},p),t_2,t_3,x,i,hd{::}ls) \to \mathsf{cnfg}(\mathsf{t}(\mathsf{task}_5,p),t_2,t_3,x,i,p{::}hd{::}ls) & [p \geq hd &] \\
\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_5,p),t_2,t_3,x,i,hd{::}ls) \to \mathsf{cnfg}(\mathsf{t}(\mathsf{task}_5,p),t_2,t_3,x,i,hd{::}ls) & [p \geq hd \wedge i = 0 &] \\
\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_5,p),t_2,t_3,x,i,hd{::}ls) \to \mathsf{cnfg}(\mathsf{t}(\mathsf{task}_6,p),t_2,t_3,x,i,hd{::}ls) & [p \geq hd \wedge i \neq 0 &] \\
\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_6,p),t_2,t_3,x,i,hd{::}ls) \to \mathsf{cnfg}(\mathsf{t}(\mathsf{task}_7(v),p),t_2,t_3,x,i,hd{::}ls) & [p \geq hd \wedge v = x &] \\
\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_7(v),p),t_2,t_3,x,i,hd{::}ls) \to \mathsf{success} & [p \geq hd \wedge v = 99] & \\
\mathsf{cnfg}(\mathsf{t}(\mathsf{task}_7(v),p),t_2,t_3,x,i,hd{::}ls) \to \mathsf{error} & [p \geq hd \wedge v \neq 99] & \\
\vdots & &
\end{array} \right\}$$

We can expect a more efficient verification by means of the LCTRS $\mathcal{R}_3^{\forall}$ obtained by adding the rule $\mathsf{cnfg}(t_1,t_2,t_3,x,i,ls) \to \mathsf{success}$ to the above LCTRS: $\mathcal{R}_3^{\forall} = \mathcal{R}_3 \cup \{\mathsf{cnfg}(t_1,t_2,t_3,x,i,ls) \to \mathsf{success}\}$. In fact, using $\mathcal{R}_3^{\forall}$, Crisys2 succeeded in proving the aforementioned APR problem in 3.541s. The efficiency was not improved very much by the modification above. This is because the process (dis)proving APR problems proceeds by breadth-first search and ends immediately at the detection of error.

To examine the modification above, changing the priorities of the three threads in Program 1, we further made experiments:

$$\langle \mathsf{cnfg}(\mathsf{t}(\mathsf{task},pr_1),\mathsf{t}(\mathsf{mid},pr_2),\mathsf{t}(\mathsf{high},pr_3),0,0,0{::}\mathsf{nil}) \mid \mathsf{true}\rangle \Rightarrow \mathsf{success}$$

Table 1 shows the detail of the experiments: Columns 1 to 3 indicate the values of pr_1,pr_2,pr_3, respectively; Column 4 shows which LCTRS is used; Columns 5 to 7 show results, running times, and heights of proof trees, respectively, where "YES"and "NO" mean that the APR problem is demonically valid and not demonically valid w.r.t. the given LCTRS, respectively; Columns 8 to 12 show the numbers of applying a rule for *case splitting*, generated subgoals, applying *unification*, applying *matching*, and solving SMT problems by the external SMT solver. In applying the rule for case splitting to a constrained term $\langle s \mid \phi\rangle$ of an APR problem $\langle s \mid \phi\rangle \Rightarrow c$ at a non-variable position p of s, for each rewrite rule $\ell \to r\ [\varphi] \in \mathcal{R}$, we first apply syntactic unification to s and ℓ; if they are unifiable with a most general unifier γ, then we check the satisfiability of the constraint $\phi\gamma \wedge \varphi\gamma$; afterwords, we obtain a new

Table 1: Experimental results of the assertion failure verification for $\mathcal{R}_2^\forall$ and $\mathcal{R}_3^\forall$

pr_1	pr_2	pr_3	LCTRS	Result	Time (s)	Ht.	#der	#Goals	#Unif.	#Match.	#SMT
10	20	30	$\mathcal{R}_2^\forall$	NO	3.867	11	78	173	854	3,072	414
			$\mathcal{R}_3^\forall$	NO	3.541	11	72	170	812	2,830	391
10	10	10	$\mathcal{R}_2^\forall$	NO	5.943	9	184	221	1,050	6,093	988
			$\mathcal{R}_3^\forall$	NO	5.444	9	173	216	1,008	5,588	947
10	10	20	$\mathcal{R}_2^\forall$	NO	5.596	10	116	205	1,008	4,540	622
			$\mathcal{R}_3^\forall$	NO	4.786	10	106	197	938	3,995	585
10	20	10	$\mathcal{R}_2^\forall$	NO	3.558	10	92	168	826	3,044	498
			$\mathcal{R}_3^\forall$	NO	3.315	10	86	166	798	2,849	475
10	20	20	$\mathcal{R}_2^\forall$	NO	4.631	11	104	192	938	3,769	564
			$\mathcal{R}_3^\forall$	NO	4.170	11	98	189	896	3,508	541
10	30	20	$\mathcal{R}_2^\forall$	NO	3.581	11	74	168	826	2,868	392
			$\mathcal{R}_3^\forall$	NO	3.201	11	68	165	784	2,632	369
20	10	10	$\mathcal{R}_2^\forall$	NO	5.218	9	112	210	994	4,715	587
			$\mathcal{R}_3^\forall$	NO	4.759	9	102	205	952	4,376	549
20	10	20	$\mathcal{R}_2^\forall$	NO	4.501	9	122	193	924	4,320	596
			$\mathcal{R}_3^\forall$	NO	4.083	9	113	188	882	3,959	564
20	10	30	$\mathcal{R}_2^\forall$	NO	5.573	10	98	205	1,008	4,335	524
			$\mathcal{R}_3^\forall$	NO	4.774	10	88	197	938	3,828	487
20	20	10	$\mathcal{R}_2^\forall$	NO	4.674	10	115	193	938	4,128	562
			$\mathcal{R}_3^\forall$	NO	3.966	10	103	185	868	3,635	513
20	30	10	$\mathcal{R}_2^\forall$	NO	3.165	10	72	161	784	2,686	369
			$\mathcal{R}_3^\forall$	NO	2.936	10	67	159	756	2,525	349
30	20	10	$\mathcal{R}_2^\forall$	NO	4.661	10	91	193	938	3,889	450
			$\mathcal{R}_3^\forall$	NO	3.941	10	79	185	868	3,430	401

APR problem $\langle s[r\gamma]_p \mid \phi\gamma \wedge \varphi\gamma \rangle \Rightarrow c$ as a subgoal. In applying a rule for *circularity* to an APR problem $\langle s \mid \phi \rangle \Rightarrow c$,[6] we try to find an APR problem $\langle s' \mid \phi' \rangle \Rightarrow c$ that has already been applied the rule for case splitting: We first apply matching to s' and s; if s' matches s with a matching substitution θ, then we check the equivalence of ϕ and $\phi'\theta$ by solving the SMT problem $\neg(\phi \Leftrightarrow \phi'\theta)$; if the SMT problem is unsatisfiable (i.e., ϕ and $\phi'\theta$ are equivalent), then we remove the APR problem $\langle s \mid \phi \rangle \Rightarrow c$. The details of solving can be seen in [10, 11]. The numbers of applying unification, matching, and SMT solving affect the efficiency of solving APR problems, and both the number of rewrite rules in the given LCTRS and the height of the resulting proof tree are relevant to the numbers of the applications. For all combinations of priorities, the expected results are "NO" because any thread can be executed first despite the priorities and there exists at least one execution path with an assertion failure. For all combinations of priorities, the modification improved the efficiency.

To get the result "YES", let us consider the case where the tasks `task` and `mid` have the high-priority 30 and the task `task` is executed before others. Note that `high` no longer interrupts the other two tasks. This case is represented by the following APR problem:

$$\langle \mathsf{cnfg}(\mathsf{t}(\mathsf{task}_5, 30), \mathsf{t}(\mathsf{mid}, 30), \mathsf{t}(\mathsf{high}, 10), 0, 0, 30{::}0{::}\mathsf{nil}) \mid \mathsf{true} \rangle \Rightarrow \mathsf{success} \tag{1}$$

[6] The rule for circularity plays a role of the application of induction hypotheses to subgoals (see [3, 10] for detail).

Table 2: Experimental results of the assertion failure verification w.r.t. the APR problem (1).

pr_1	pr_2	pr_3	LCTRS	Result	Time (s)	Ht.	#der	#Goals	#Unif.	#Match.	#SMT
30	30	10	$\mathcal{R}_2^\forall$	YES	0.314	11	20	38	196	315	86
			$\mathcal{R}_3^\forall$	YES	0.187	7	14	32	126	186	59

Note that to represent that `task` has already been executed, the stack in the above configuration has the stack that already has 30 as its head element. In any execution path of $\mathcal{R}_3^\forall$, the task `high` is always executed after terminating the other two, and thus the assertion is no longer violated. Table 2 shows the detail of the experiments. The result of solving the APR problem is "YES" which is an expected one. In proving the above APR problem for $\mathcal{R}_2^\forall$ and $\mathcal{R}_3^\forall$, the modification made proof trees smaller, improving the efficiency significantly. This is because for demonically valid APR problems, all reachable constrained terms are taken into account to prove the validity, and thus the modification significantly reduces the search space.

6 Conclusion

In this paper, we extended the transformation of multithreaded programs into LCTRSs to prioritized threads by introducing a descending list of priorities into terms representing configurations. Then, we showed a verification example in which the APR-based method detects an assertion failure caused by the interrupt of a high-priority thread. We will formulate the extended transformation to prove its correctness. In addition, we will implement the extended transformation and make an empirical evaluation using several examples. A further future direction is to extend the transformation to other constructs and scheduling strategies (e.g., `SCHED_FIFO`) for multithreading.

References

[1] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, doi:10.1145/505863.505888.

[2] Ştefan Ciobâcă, Dorel Lucanu & Andrei-Sebastian Buruiană (2023): *Operationally-based program equivalence proofs using LCTRSs*. *Journal of Logical and Algebraic Methods in Programming* 135, pp. 1–22, doi:10.1016/j.jlamp.2023.100894.

[3] Ştefan Ciobâcă & Dorel Lucanu (2018): *A Coinductive Approach to Proving Reachability Properties in Logically Constrained Term Rewriting Systems*. In Didier Galmiche, Stephan Schulz & Roberto Sebastiani, editors: *Proceedings of the 9th International Joint Conference on Automated Reasoning*, *Lecture Notes in Computer Science* 10900, Springer, pp. 295–311, doi:10.1007/978-3-319-94205-6_20.

[4] Maribel Fernández (2014): *Programming Languages and Operational Semantics – A Concise Overview*. Undergraduate Topics in Computer Science, Springer, doi:10.1007/978-1-4471-6368-8.

[5] Carsten Fuhs, Cynthia Kop & Naoki Nishida (2017): *Verifying Procedural Programs via Constrained Rewriting Induction*. *ACM Transactions on Computational Logic* 18(2), pp. 14:1–14:50, doi:10.1145/3060143.

[6] Yoshiaki Kanazawa & Naoki Nishida (2019): *On Transforming Functions Accessing Global Variables into Logically Constrained Term Rewriting Systems*. In Joachim Niehren & David Sabel, editors: *Proceedings of the 5th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, *Electronic Proceedings in Theoretical Computer Science* 289, Open Publishing Association, pp. 34–52.

[7] Yoshiaki Kanazawa, Naoki Nishida & Masahiko Sakai (2019): *On Representation of Structures and Unions in Logically Constrained Rewriting*. IEICE Technical Report SS2018-38, the Institute of Electronics, Information and Communication Engineers. Vol. 118, No. 385, pp. 67–72, in Japanese.

[8] Misaki Kojima & Naoki Nishida (2022): *On Reducing Non-Occurrence of Specified Runtime Errors to All-Path Reachability Problems of Constrained Rewriting*. In Ştefan Ciobâcă & Keisuke Nakano, editors: *Informal Proceedings of the 9th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, pp. 1–16. Available at `https://easychair.org/publications/preprint/TM7q`.

[9] Misaki Kojima & Naoki Nishida (2023): *From Starvation Freedom to All-Path Reachability Problems in Constrained Rewriting*. In Michael Hanus & Daniela Inclezan, editors: *Proceedings of the 25th International Symposium on Practical Aspects of Declarative Languages*, *Lecture Notes in Computer Science* 13880, Springer Nature Switzerland, pp. 161–179, doi:10.1007/978-3-031-24841-2_11.

[10] Misaki Kojima & Naoki Nishida (2023): *Reducing non-occurrence of specified runtime errors to all-path reachability problems of constrained rewriting*. *Journal of Logical and Algebraic Methods in Programming* 135, pp. 1–19, doi:10.1016/j.jlamp.2023.100903.

[11] Misaki Kojima & Naoki Nishida (2024): *On Representations of Waiting Queues for Semaphores in Logically Constrained Term Rewrite Systems with Constant Destinations*. *Journal of Information Processing* 32, pp. 417–435, doi:10.2197/IPSJJIP.32.417.

[12] Misaki Kojima & Naoki Nishida (2024): *On Solving All-Path Reachability Problems for Starvation Freedom of Concurrent Rewrite Systems Under Process Fairness*. In Laura Kovács & Ana Sokolova, editors: *Proceedings of the 18th International Conference on Reachability Problems*, *Lecture Notes in Computer Science* 15050, Springer, pp. 54–70, doi:10.1007/978-3-031-72621-7_5.

[13] Misaki Kojima, Naoki Nishida & Yutaka Matsubara (2020): *Transforming Concurrent Programs with Semaphores into Logically Constrained Term Rewrite Systems*. In Adrián Riesco & Vivek Nigam, editors: *Informal Proceedings of the 7th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, pp. 1–12.

[14] Misaki Kojima, Naoki Nishida & Yutaka Matsubara (2025): *Transforming concurrent programs with semaphores into logically constrained term rewrite systems*. *Journal of Logical and Algebraic Methods in Programming* 143, pp. 1–23, doi:10.1016/j.jlamp.2024.101033.

[15] Cynthia Kop & Naoki Nishida (2013): *Term Rewriting with Logical Constraints*. In Pascal Fontaine, Christophe Ringeissen & Renate A. Schmidt, editors: *Proceedings of the 9th International Symposium on Frontiers of Combining Systems*, *Lecture Notes in Computer Science* 8152, Springer, pp. 343–358, doi:10.1007/978-3-642-40885-4_24.

[16] Cynthia Kop & Naoki Nishida (2014): *Automatic Constrained Rewriting Induction towards Verifying Procedural Programs*. In Jacques Garrigue, editor: *Proceedings of the 12th Asian Symposium on Programming Languages and Systems*, *Lecture Notes in Computer Science* 8858, Springer, pp. 334–353, doi:10.1007/978-3-319-12736-1_18.

[17] Ayuka Matsumi, Naoki Nishida, Misaki Kojima & Donghoon Shin (2023): *On Singleton Self-Loop Removal for Termination of LCTRSs with Bit-Vector Arithmetic*. In Akihisa Yamada, editor: *Proceedings of the 19th International Workshop on Termination*, pp. 1–6, doi:10.48550/arXiv.2307.14094.

[18] Leonardo Mendonça de Moura & Nikolaj S. Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[19] Naoki Nishida & Sarah Winkler (2018): *Loop Detection by Logically Constrained Term Rewriting*. In Ruzica Piskac & Philipp Rümmer, editors: *Proceedings of the 10th Working Conference on Verified Software: Theories, Tools, and Experiments*, *Lecture Notes in Computer Science* 11294, Springer, pp. 309–321, doi:10.1007/978-3-030-03592-1_18.

[20] Enno Ohlebusch (2002): *Advanced Topics in Term Rewriting*. Springer, doi:10.1007/978-1-4757-3661-8.

[21] Sarah Winkler & Aart Middeldorp (2018): *Completion for Logically Constrained Rewriting*. In Hélène Kirchner, editor: *Proceedings of the 3rd International Conference on Formal Structures for Computation and Deduction*, *Leibniz International Proceedings in Informatics* 108, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 30:1–30:18, doi:10.4230/LIPIcs.FSCD.2018.30.

Probabilistic Lazy PCF with Real-Valued Choice

David Sabel
Hochschule RheinMain Wiesbaden
david.sabel@hs-rm.de

Manfred Schmidt-Schauß
Goethe-University Frankfurt
manfredschauss@gmail.com

The functional language PCF with monomorphic typing, fixed-point operator, a let-construct for sharing and call-by-need evaluation is extended with a binary probabilistic choice operator resulting in probabilistic Lazy PCF. This language is further extended by permitting real-valued computable probabilities for the choice operator. Equivalence of programs is a variant of contextual equivalence, which compares programs based on their expected convergence in all program-contexts. As a tool, distribution equivalence of closed programs of type nat is used, thus simplifying reasoning on equivalence of programs. The main result shows that the extension of binary choice with probability 0.5 to arbitrary real computable numbers in (0,1) is conservative. Programming examples in probabilistic Lazy PCF are given to illustrate the reasoning, modifying and comparing the models of stochastic experiments.

1 Introduction

Pure, lazy[1] functional programming languages such as Haskell allow for a rigorous mathematical treatment of programs and offer many valid program transformations. Implementations of these languages use a call-by-need strategy for program execution, combining lazy evaluation with sharing to avoid the repeated evaluation of sub-expressions (see e.g. [3, 2] for call-by-need lambda calculi).

Probabilistic programming (see e.g. [5]) extends programming languages by adding probabilistic constructs so that they can describe stochastic models. Techniques from programming language semantics can be used to improve reasoning about the programmed probabilistic experiments. Therefore, we are interested in combining lazy evaluation and functional programming with probabilistic programming exploiting call-by-need sharing to have exact control over the side-effects of probabilistic choice.

As a functional core language, we consider Plotkin's language for "Programming Computable Functions" [15], PCF for short. It is a typed functional programming language that extends the simply-typed call-by-name lambda-calculus with natural numbers, branching and number operations, and a fixpoint operator thereby establishing Turing-completeness.

In our programming language "probabilistic lazy PCF" (introduced in [21]), we extended a call-by-need variant of simply-typed PCF with a fair coin toss $s \oplus t$ that chooses between expressions s and t with probability 0.5 for each expression. The language includes `let`-expressions to implement the sharing required for the call-by-need operational semantics. The language has monomorphic types, which allow functions and numbers to be distinguished, and allows flexible and type-safe programming.

Program evaluation is deterministic for all constructs except the binary infix operator $\oplus$, which can choose the left or right argument provided it is a reduction position. Probabilities do not play a role during a single execution, however, they do play a role in the analysis, and also if program execution is used as Monte-Carlo simulations.

[1] With the term 'lazy' we identify call-by-need evaluation to weak head normal form, i.e. not to evaluate to (head) normal form. This is a difference to [16, 13] where 'Lazy PCF' means adding a convergence tester for expressions where convergence to numbers and functions is tested. However, our notion of convergence also tests for numbers and abstractions.

In this probabilistic setting, call-by-value, call-by-name, and call-by-need evaluation are all different. For instance, consider the function call $f\ (1 \oplus 2)\ (3 \oplus \bot)$, where $f\ x\ y = x + x$ and $\bot$ is a diverging expression. Then it may result in $2, 3, 4$ using call-by-name evaluation, since $1 \oplus 2$ is copied for both occurrences of x. Using call-by-need or call-by-value evaluation the possible values are 2 or 4, since the result of $1 \oplus 2$ is shared for both occurrences of x. But, since call-by-value evaluation evaluates all arguments before substituting them in the function body, there is a possibility for divergence (with probability 0.5), while call-by-need (and call-by-name) evaluation cannot diverge in this example.

In this paper we consider an extension of the language by permitting the more general $s \overset{p}{\oplus} t$ instead of $s \oplus t$ for the probabilistic choice between s and t where p is the probability of choosing s and $(1 - p)$ is the probability of choosing t. We allow p to be a computable real number in the open interval $(0, 1)$. This expressivity permits to transform programs into equivalent ones requiring less coin tosses during execution.

Our language permits to program and combine discrete probabilistic experiments in a way that supports the analysis and correct transformation of programs or subprograms. For example, throwing a dice with the six possible results $1, 2, 3, 4, 5$ and 6 with probability $1/6$ each is programmed by $1 \overset{1/6}{\oplus} (2 \overset{1/5}{\oplus} (3 \overset{1/4}{\oplus} (4 \overset{1/3}{\oplus} (5 \oplus 6))))$ where $1/6, 1/5, 1/4, 1/3$ are the probabilities represented as rational numbers. Clearly this is more efficient than using fair coin throws. The non-deterministic execution can be interpreted as the throwing of coins with the respective probabilities, and executing the whole expression corresponds to throwing a dice with the usual distribution. Program evaluation corresponds to a single experiment, and the experiment has a discrete probability distribution.

The intention behind using programs in this setting differs from that in usual deterministic programming: the intention is to provide programs that represent probabilistic experiments, where a single execution corresponds to a single probabilistic experiment. The gain using probabilistic lazy PCF is that there is a rich set of program equivalences that offer approximate executions, and also correct transformations of experiments into other ones with the same distribution. Also analyses justified by program semantics can be applied to reason on the probability distribution.

We call closed expressions that compute natural numbers *stochastic programs*. In [21] we proved that for stochastic programs, distribution-equivalence is the same as contextual equivalence where the latter tests for the expected convergence in all contexts. The notions can easily be transferred to the extended calculus. Also the correspondence between distribution-equivalence and contextual equivalence could be proved to hold in the extended language. The proof is straight-forward and thus we do not repeat it.

In this paper, we address the question, from the perspective of programming language semantics, of whether the real-valued probabilistic operator is necessary, or whether it can be simulated in a smaller language. We give a positive answer to this question, that is, we show that the extended language is a conservative extension of the one that only permits probabilities 0.5. More concretely, we prove that every stochastic program of the extended language is distribution-equivalent to a stochastic programs that only uses fair coin tosses. The core idea of the encoding is well-known (see e.g. [4, Lemma 7.12] for a similar encoding on probabilistic Turing machines): The computability of the real number is exploited by computing the number bit-wise and throwing a fair coin for every bit. This is achieved using a recursive program, resulting in a finite program. In addition to these technical results, we provide several examples of probabilistic programs.

Related Work. Early investigations of probabilistic programming languages are [20, 8]. For example, an operational semantics for a probabilistic lambda calculus has been defined in [12], including call-by-name and call-by-value semantics, but no call-by-need semantics. Contextual equivalence is defined analogously to our definition in [11, 6] for call-by-name and call-by-value calculi. However, both works

differ from our work because in the probabilistic setting the semantics of call-by-name, call-by-value and call-by-need are different. In [10] operational semantics, contextual equivalence, expressiveness and termination of a typed call-by-value PCF are discussed. In terms of the classification in [10], our calculus is a *randomized lambda calculus* (with sharing), since it performs a random evaluation of the choice operator. This paper extends our work in [21]. The combination of probabilistic choice and call-by-need evaluation was also analyzed in [18] where the analyzed language has recursive let, but no numbers, and it is untyped.

Call-by-need lambda-calculi with non-determinism are related to calculi with probabilistic choice. This combination was investigates in several works. For example, in [9] the contextual semantics of an untyped calculus with non-recursive let is investigated. Lambda-calculi with recursive let and non-deterministic operators can be found for instance in [14, 17].

Outline. In Section 2 we recall the language $ProbPCF^{need}$, its operational semantics, contextual equivalence and distribution-equivalence. We illustrate the definitions with examples. In Section 3 we extend the language to real-valued probabilities and provide the necessary adaptations of the definitions. In Section 4 we show that the language extension can be encoded in the original calculus. After explaining the encoding, we show that the encoding is distribution-equivalent. In Section 5 we conclude and give some directions for future work. For space reasons, some proofs are given in a technical appendix.

2 The Language $ProbPCF^{need}$

We recall the syntax and semantics of $ProbPCF^{need}$ from [21].

2.1 Syntax, Typing, and Operational Semantics of $ProbPCF^{need}$

In this section, we first introduce the syntax, followed by the operational semantics presented as a non-deterministic reduction relation. We defer the treatment of probabilities to the next section to ease understanding.

Definition 2.1 (Syntax of Expressions and Types)**.** Let *Var* be an infinite and countable set of variables. We use x, y, z, x_i, y_i, z_i for variables of *Var*. The syntax of *expressions* $s, t, s_t, t_i \in Expr$ and *types* $\tau, \rho, \sigma \in Typ$ is given by the following grammar:

$$\begin{aligned} s,t,s_i,t_i \in Expr ::= {} & x \mid \lambda x.s \mid (s\ t) \mid (s \oplus t) \mid \texttt{let}\ x = s\ \texttt{in}\ t \mid \texttt{if}\ s\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \\ & \mid \texttt{fix}\ s \mid \texttt{pred}\ s \mid \texttt{succ}\ s \mid n \text{ where } n \in \mathbb{N} \\ \tau,\rho,\sigma \in Typ ::= {} & nat \mid \tau \to \rho \end{aligned}$$

In an abstraction $\lambda x.s$ and in a let-expression $\texttt{let}\ x = t\ \texttt{in}\ s$ the variable x is bound with scope s. For an expression t, this induces the usual notion of free variables $FV(t)$, bound variables $BV(t)$, α-renaming, and α-equivalence. If $FV(t) = \emptyset$, the expression t is called *closed* or alternatively a *program*, otherwise, the expression t is *open*. We write $\lambda x_1, x_2, \ldots, x_n.s$ as abbreviation for $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.s$.

The construct $s \oplus t$ is called a *prob-expression*. It will non-deterministically choose between s and t. The operator `fix` is a fix-point operator. Naturals $n \in \mathbb{N} = \{0, 1, 2 \ldots\}$ are built in and `succ` computes the successor of a number, while `pred` computes the predecessor with the exception of $(\texttt{pred}\ 0)$ which results in 0. Branching can be programmed using $\texttt{if}\ s\ \texttt{then}\ t_1\ \texttt{else}\ t_2$ depending on the value of s, where 0 is identified with true and any other natural with false.

The language $ProbPCF^{need}$ is monomorphically typed. The type *nat* represents natural numbers, and the type $\tau \to \rho$ represents the type of a function from the type τ to the type ρ. The typing rules

are standard, so we omit them and refer to [21] for their definition. We write $s : \tau$ if the expression s is well-typed with type τ. A context C is an expression with one hole $[\cdot]$ at expression position. We write $C[\cdot_\sigma] : \tau$ for a context where any expression s of type σ can fill the hole, written $C[s]$, such that $C[s] : \tau$. A closed program of type *nat* is also called a *stochastic program*.

To define operational semantics in form of a reduction relation, called standard reduction, the three context classes of *A*-, *LR*-, and *R*-contexts are used. The *A*-contexts are evaluation contexts as used in the lambda-calculus with call-by-name evaluation (and thus the context hole is in function position of applications). They are adapted to the constructs of the language where evaluation of the first argument of `fix`, `succ`, `pred`, and `if-then-else` is enforced, since these operators are strict in their first argument. The LR-contexts represent an environment consisting of nested `let`-expressions. The reduction contexts R combine the LR- and A-contexts such that the hole position is in the `in`-expressions of `let` and also in the right-hand side of a `let`-binding if it is needed – i.e., if the bound variable occurs in a reduction context. *Flat A-contexts* A^1 are A-contexts with hole-depth 1

$$\begin{aligned} A &::= [\cdot] \mid (A\ s) \mid \texttt{if}\ A\ \texttt{then}\ s\ \texttt{else}\ t \mid \texttt{pred}\ A \mid \texttt{succ}\ A \mid \texttt{fix}\ A \\ A^1 &::= ([\cdot]\ s) \mid \texttt{if}\ [\cdot]\ \texttt{then}\ s\ \texttt{else}\ t \mid \texttt{pred}\ [\cdot] \mid \texttt{succ}\ [\cdot] \mid \texttt{fix}\ [\cdot] \\ \mathrm{LR} &::= [\cdot] \mid \texttt{let}\ x = s\ \texttt{in}\ \mathrm{LR} \\ R &::= \mathrm{LR}[A] \mid \mathrm{LR}[\texttt{let}\ x = A\ \texttt{in}\ R[x]] \end{aligned}$$

A *value* in *ProbPCF*^{need} is a natural or an abstraction and a *weak head normal form* (WHNF) additionally allows an outer LR-context.

$$\text{Values: } v ::= n \mid \lambda x.s \qquad\qquad \text{WHNFs: } w ::= \mathrm{LR}[v]$$

For a WHNF of the form $\mathrm{LR}[n]$ with $n \in \mathbb{N}$, we write $val(\mathrm{LR}[n])$ to denote the number n (without the LR-context).

The standard reduction of *ProbPCF*^{need} is a small-step reduction that respects sharing:

Definition 2.2 (Standard reduction)**.** The *standard reduction* of *ProbPCF*^{need} is denoted by $\xrightarrow{sr}$ and defined by the following rules:

$$\begin{array}{ll}
(sr,fix) & R[\texttt{fix}\ \lambda x.s] \to R[(\lambda x.s)\ (\texttt{fix}\ \lambda x.s)] \\
(sr,lbeta) & R[(\lambda x.s)\ t] \to R[\texttt{let}\ x = t\ \texttt{in}\ s] \\
(sr,succ) & R[\texttt{succ}\ n] \to R[m] \text{ with } m = n+1 \\
(sr,pred) & R[\texttt{pred}\ n] \to R[m] \text{ with } m = \max(0, n-1) \\
(sr,if\text{-}0) & R[\texttt{if}\ 0\ \texttt{then}\ s\ \texttt{else}\ t] \to R[s] \\
(sr,if\text{-}not\text{-}0) & R[\texttt{if}\ n\ \texttt{then}\ s\ \texttt{else}\ t] \to R[t] \text{ if } n \neq 0 \\
(sr,probl) & R[s \oplus t] \to R[s] \\
(sr,probr) & R[s \oplus t] \to R[t] \\
(sr,lflata) & R[A^1[\texttt{let}\ x = s\ \texttt{in}\ t]] \to R[\texttt{let}\ x = s\ \texttt{in}\ A^1[t]] \\
(sr,llet) & \mathrm{LR}[\texttt{let}\ x = (\texttt{let}\ y = s\ \texttt{in}\ t)\ \texttt{in}\ R[x]] \\
 & \to \mathrm{LR}[\texttt{let}\ y = s\ \texttt{in}\ \texttt{let}\ x = t\ \texttt{in}\ R[x]] \\
(sr,cp) & \mathrm{LR}[\texttt{let}\ x = v\ \texttt{in}\ R[x]] \to \mathrm{LR}[\texttt{let}\ x = v\ \texttt{in}\ R[v]]
\end{array}$$

We apply standard reductions only to typed expressions. For a standard reduction step (*sr-step*, for short) $s \xrightarrow{sr} t$, we sometimes write $s \xrightarrow{sr,lab} t$ where (sr,lab) is the name of the applied reduction rule, e.g. $s \xrightarrow{sr,cp} t$. With $\xrightarrow{sr,+}$ ($\xrightarrow{sr,*}$, resp.) we denote the transitive (reflexive-transitive, resp.) closure of $\xrightarrow{sr}$.

We give a short explanation of the reduction rules. The rule $(sr, lbeta)$ is the call-by-need variant of β-reduction: the argument of the application is shared by a new `let`-binding instead of substituting the formal parameter of the abstraction. The copy-rule (sr, cp) performs needed substitutions, where only values are copied. The rule (sr, fix) evaluates the fix-point operator. Computation of operations on numbers is performed by $(sr, succ)$ and $(sr, pred)$, branching is evaluated by rules $(sr, if\text{-}0)$ and $(sr, if\text{-}not\text{-}0)$. The rules $(sr, lflata)$ and $(sr, llet)$ rearrange `let`-bindings w.r.t. other `let`-bindings and flat A-contexts. For instance, In $(\texttt{let}\ x = u\ \texttt{in}\ (\lambda y.s))\ t$ the `let` is shifted over the application, before the application is evaluated:

$$(\texttt{let}\ x = u\ \texttt{in}\ (\lambda y.s)\ t) \xrightarrow{sr,llet} \texttt{let}\ x = u\ \texttt{in}\ ((\lambda y.s)\ t) \xrightarrow{sr,lbeta} \texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = t\ \texttt{in}\ s))$$

Rules $(sr, probl)$ and $(sr, probr)$ evaluate prob-expressions. Standard reduction is non-deterministic: all rules are deterministic except for the rules $\xrightarrow{sr,probl}$ and $\xrightarrow{sr,probr}$ where the left or the right argument of a prob-expression is chosen as next expression.

Example 2.3. We show all reductions for expression $t = (\lambda x.\texttt{if}\ x\ \texttt{then}\ (x \oplus 2)\ \texttt{else}\ \texttt{pred}\ x)\ (0 \oplus 1)$:

i)
$$\begin{array}{ll} t & \xrightarrow{sr,lbeta} \texttt{let}\ x = (0 \oplus 1)\ \texttt{in} \\ & \quad \texttt{if}\ x\ \texttt{then}\ (x \oplus 2)\ \texttt{else}\ \texttt{pred}\ x \\ & \xrightarrow{sr,probl} \texttt{let}\ x = 0\ \texttt{in} \\ & \quad \texttt{if}\ x\ \texttt{then}\ (x \oplus 2)\ \texttt{else}\ \texttt{pred}\ x \\ & \xrightarrow{sr,cp} \texttt{let}\ x = 0\ \texttt{in} \\ & \quad \texttt{if}\ 0\ \texttt{then}\ (x \oplus 2)\ \texttt{else}\ \texttt{pred}\ x \\ & \xrightarrow{sr,if\text{-}0} \texttt{let}\ x = 0\ \texttt{in}\ (x \oplus 2) \\ & \xrightarrow{sr,probl} \texttt{let}\ x = 0\ \texttt{in}\ x \\ & \xrightarrow{sr,cp} \texttt{let}\ x = 0\ \texttt{in}\ 0 \end{array}$$

ii)
$$\begin{array}{ll} t & \xrightarrow{sr,lbeta} \texttt{let}\ x = (0 \oplus 1)\ \texttt{in} \\ & \quad \texttt{if}\ x\ \texttt{then}\ (x \oplus 2)\ \texttt{else}\ \texttt{pred}\ x \\ & \xrightarrow{sr,probl} \texttt{let}\ x = 0\ \texttt{in} \\ & \quad \texttt{if}\ x\ \texttt{then}\ (x \oplus 2)\ \texttt{else}\ \texttt{pred}\ x \\ & \xrightarrow{sr,cp} \texttt{let}\ x = 0\ \texttt{in} \\ & \quad \texttt{if}\ 0\ \texttt{then}\ (x \oplus 2)\ \texttt{else}\ \texttt{pred}\ x \\ & \xrightarrow{sr,if\text{-}0} \texttt{let}\ x = 0\ \texttt{in}\ (x \oplus 2) \\ & \xrightarrow{sr,probr} \texttt{let}\ x = 0\ \texttt{in}\ 2 \end{array}$$

iii)
$$\begin{array}{ll} t & \xrightarrow{sr,lbeta} \texttt{let}\ x = (0 \oplus 1)\ \texttt{in}\ \texttt{if}\ x\ \texttt{then}\ (x \oplus 2)\ \texttt{else}\ \texttt{pred}\ x \\ & \xrightarrow{sr,probr} \texttt{let}\ x = 1\ \texttt{in}\ \texttt{if}\ x\ \texttt{then}\ (x \oplus 2)\ \texttt{else}\ \texttt{pred}\ x \\ & \xrightarrow{sr,cp} \texttt{let}\ x = 1\ \texttt{in}\ \texttt{if}\ 1\ \texttt{then}\ (x \oplus 2)\ \texttt{else}\ \texttt{pred}\ x \\ & \xrightarrow{sr,if\text{-}not\text{-}0} \texttt{let}\ x = 1\ \texttt{in}\ \texttt{pred}\ x \\ & \xrightarrow{sr,cp} \texttt{let}\ x = 1\ \texttt{in}\ \texttt{pred}\ 1 \\ & \xrightarrow{sr,pred} \texttt{let}\ x = 1\ \texttt{in}\ 0 \end{array}$$

2.2 Expected Convergence and Contextual Equivalence

Our operational semantics is non-deterministic, but it does not track the probability of different evaluations or results (in form of the WHNFs). We now introduce probabilities. A *prob-sequence* is a (finite) sequence of pairs $(probl, p)$ and $(probr, p)$ where $p \in (0, 1)$.

For a reduction sequence $s_0 \xrightarrow{sr,a_1} \cdots \xrightarrow{sr,a_n} s_n$, with $\mathsf{PS}(s_0 \xrightarrow{sr,a_1} \cdots \xrightarrow{sr,a_n} s_n)$, we denote the prob-sequence derived from $a_1, \ldots, a_n$, where all labels a_i are removed if $a_i \notin \{probl, probr\}$ and the remaining labels a_j are replaced by $(a_j, 0.5)$. Hence, in *ProbPCF*^{need} the probabilities p in prob-sequences are always 0.5. However, in the next section this will be generalized.

For the reduction sequences in Example 2.3, the prob-sequences are i) $(probl, 0.5), (probl, 0.5)$ ii) $(probr, 0.5), (probr, 0.5)$, and iii) $(probr, 0.5)$.

An evaluation of s is a reduction sequence $s \xrightarrow{sr,a,*} t$ where t is a WHNF. Since the only source of non-determinism is choosing between s or t in $(s \oplus t)$ (i.e. all other reduction rules are deterministic and the redex of every reduction is unique), an evaluation is uniquely determined by s and the prob-sequence $\mathsf{PS}(s \xrightarrow{sr,a,*} t)$. With $WHNF(s,S)$, we denote the WHNF t (up to α-equivalence) that is obtained for expression s and with prob-sequence S, where $WHNF(s,S)$ is undefined, if there is no evaluation for s with prob-sequence S.

The *probability* $\mathsf{P}(S)$ of a prob-sequence $S = (a_1,p_1),\ldots,(a_n,p_n)$ is the product of the probabilities p_i, i.e. $\mathsf{P}(S) = \prod_{(a_i,p_i)\in S} p_i$.

For an expression s, $Eval(s)$ denotes the prob-sequences S of all evaluations of s and the *expected convergence* $\mathrm{ExCv}(s)$ is the (perhaps infinite) sum of the probabilities of these evaluations. In addition, for a natural n, the *expected value convergence of $s : nat$ on value n*, denoted $\mathrm{ExVCv}(s,n)$, only takes into account evaluations resulting in expressions of the form $\mathrm{LR}[n]$ (where n is an integer). Thus:

$$\mathrm{ExCv}(s) := \sum_{S\in Eval(s)} \mathsf{P}(S) \quad \text{and} \quad \mathrm{ExVCv}(s,n) := \sum_{\substack{S\in Eval(s),\\ val(WHNF(s,S))=n}} \mathsf{P}(S)$$

Expected convergence and value-convergence are well-defined, i.e. the limits of the infinite sums always exist and are unique. Note that for all expressions $s : nat$: $\mathrm{ExCv}(s) = \sum_{i\in\mathbb{N}} \mathrm{ExVCv}(s,i)$.

For the expression t from Example 2.3, we have $\mathrm{ExCv}(t) = 1$ and $\mathrm{ExVCv}(t,0) = 0.75$, $\mathrm{ExVCv}(t,2) = 0.25$ and $\mathrm{ExVCv}(t,i) = 0$ for $i \notin \{0,2\}$.

We define the contextual semantics of *ProbPCF^{need}* in terms of contextual equivalence which identifies expressions as equal if their behavior is the same when they are plugged into any context. As behavior we observe the expected convergence and as tests (i.e. the contexts) we only consider natural numbers as output.

Definition 2.4 (Contextual Preorder and Equivalence)**.** Let $s,t : \sigma$. If for all contexts $C[\cdot_\sigma] : nat$, we have $\mathrm{ExCv}(C[s]) \le \mathrm{ExCv}(C[t])$, then $s \le_c t$. We define $s \sim_c t$ iff $s \le_c t$ and $t \le_c s$. The relation $\le_c$ is called *contextual preorder*, and $\sim_c$ is called *contextual equivalence*.

The contextual preorder can be used to compare and order non-equivalent expressions. For instance, diverging programs are least elements of the preorder.

2.3 Distribution-Equivalence

Definition 2.5. A (discrete) probability distribution is a function $p : \mathbb{N} \to [0,1]$ such that $\sum_{i\in\mathbb{N}} p(i) \le 1$. Stochastic programs s of *ProbPCF^{need}* induce a probability distribution by setting $p(n) := \mathrm{ExVCv}(s,n)$. For stochastic program s, let s_d be its *distribution function*, i.e. for all $i \in \mathbb{N}$: $s_d(i) = \mathrm{ExVCv}(s,i)$.

The defect, i.e. $1 - \sum_{i\in\mathbb{N}} p(i)$ is the probability of non-termination, sometimes denoted as $p(\bot)$. As usual, for a program $s : nat$, the *expected value* of s is $E[s] = \sum_{i\in\mathbb{N}} i \cdot s_d(i)$.

Definition 2.6 (Distribution-Equivalence)**.** We say a distribution d' *approximates* a distribution d iff for all $i \in \mathbb{N}$: $d'(i) \le d(i)$. If d' approximates d, then we also write $d' \le d$.

We also use the formulation for the programs that generate the distributions, i.e. for stochastic programs $s,t : nat$, we write $s \le_d t$ (called *distribution approximation*) if $s_d \le t_d$ holds. If and only if $s \le_d t$ and $t \le_d s$ hold, then we write $s \sim_d t$ and say that s and t are *distribution-equivalent*.

On stochastic programs we know that distribution-equivalence is the same as contextual equivalence [21]. We conjecture that also contextual preorder and distribution approximation coincide (i.e. $s \le_c t \iff s \le_d t$), but its proof is work in progress.

2.4 Examples

For the examples, we assume that usual operations on numbers like +, −, ∗, and comparisons like ==, <= etc. are already defined.

A very simple game is tossing a coin with a bet of 1 euro. Head wins and tail loses. For simplicity, let us encode the result tail with 0 and head with 1. The outcome of euro (not including the bet) after playing the simple game can be programmed as:

$$\mathit{simpleGame} = \texttt{if}\ (0 \oplus 1) == 0\ \texttt{then}\ 0\ \texttt{else}\ 1$$

The expected value convergence for 0 and 1 is 0.5, for all other numbers it is 0. Hence the distribution function of *simpleGame* is

$$\mathit{simpleGame}_d(i) = \begin{cases} 0.5, & i \in \{0,1\} \\ 0, & \text{otherwise} \end{cases}$$

The expected value is $E[\mathit{simpleGame}] = 0.5$.

An extension of the simple game is to proceed in case of losing and then iterating this until head occurs. In this case the player has an infinite amount of money to play the game arbitrary often.

The corresponding program to compute the outcome is:

$$\mathit{game1} = \texttt{let}\ \mathit{throw} = \lambda f.\lambda \mathit{bet}.\texttt{if}\ (0 \oplus 1) == 1\ \texttt{then}\ \mathit{bet}\ \texttt{else}\ f\ \mathit{bet}\ \texttt{in}\ \texttt{fix}\ \mathit{throw}\ 1$$

Even there is an infinite sr-reduction sequence of *game1* (if always 0 is the result of $0 \oplus 1$), this experiment terminates with probability 1, i.e. the expected convergence is $\text{ExCv}(\mathit{game1}) = 1$ and the expected outcome is $E[\mathit{game1}] = 1$. The distribution function is

$$\mathit{game1}_d(i) = \begin{cases} 1, & i = 1 \\ 0, & \text{otherwise} \end{cases}$$

Note that our modeling ignores the amount of invested money.

As another variant, we consider the game, where the bet is doubled in the case of losing. Again let us ignore the invested amount and just compute the outcome:

$$\mathit{game2} = \texttt{let}\ \mathit{throw} = \lambda f.\lambda \mathit{bet}.\texttt{if}\ (0 \oplus 1) == 1\ \texttt{then}\ \mathit{bet}\ \texttt{else}\ f\ (2 * \mathit{bet})\ \texttt{in}\ \texttt{fix}\ \mathit{throw}\ 1$$

The expected convergence is $\text{ExCv}(\mathit{game2}) = 1$ the distribution function is

$$\mathit{game2}_d(i) = \begin{cases} \frac{1}{2^j}, & \text{if } i = 2^j,\ j \in \mathbb{N} \\ 0, & \text{otherwise} \end{cases}$$

and the expected value of is infinite, since the infinite sum $\sum_{i \in \mathbb{N}} 2^{i+1} \cdot \frac{1}{2^i} = \sum_{i \in \mathbb{N}} \frac{1}{2}$ has no upper bound.

3 The Language Extension $\mathit{ProbPCF}^{need}_{\mathbb{R}}$

We introduce an extension of $\mathit{ProbPCF}^{need}$ by more flexible coin tosses with real-valued probabilities in the interval $(0,1)$. We only permit probabilities that are computable numbers. The computable numbers include rational numbers and solutions of polynomial equations, for example $0.5\sqrt{2}$.

Definition 3.1 (Syntax and Semantics of $ProbPCF^{need}_{\mathbb{R}}$)**.** The language $ProbPCF^{need}_{\mathbb{R}}$ is defined as an extension of $ProbPCF^{need}$ by permitting a computable probability for every single coin toss. The syntax extension is to permit $s \overset{r}{\oplus} t$, where r is a computable number in $(0,1)$.

The standard reduction is not changed, however, we define a *probability measure* for prob-reductions as follows. We define:

$$\begin{array}{lll} (sr, probl) & R[s \overset{r}{\oplus} t] \to R[s] & \text{has probability measure } r \\ (sr, probr) & R[s \overset{r}{\oplus} t] \to R[t] & \text{has probability measure } 1-r \end{array}$$

For a sr-sequence $s_0 \xrightarrow{sr,a_1} s_1 \cdots \xrightarrow{sr,a_n} s_n$, the *prob-sequence* $PS(s_0 \xrightarrow{sr,a_1} s_1 \cdots \xrightarrow{sr,a_n} s_n)$ is the sequence of pairs (a_i, r_i) derived from $a_1, \ldots, a_n$ as follows: first drop all labels $a_i \notin \{probl, probr\}$ and for the labels $a_i \in \{probl, probr\}$ replace a_i by (a_i, r_i) where r_i is the probability measure of the reduction $s_{i-1} \xrightarrow{sr,a_i} s_i$.

In $ProbPCF^{need}_{\mathbb{R}}$ we use the same definitions as in $ProbPCF^{need}$ for the probability $\mathsf{P}(S)$ of prob-sequences S, the set of evaluations $Eval(s)$ for expression s, the expected convergence $\mathrm{EXCV}(s)$, the expected value convergence $\mathrm{EXVCV}(s,n)$ of s on value n, the contextual preorder and equivalence, and distribution approximation and equivalence.

3.1 Examples

We first give an example with rational probabilities, and then consider an example where real-valued probabilities occur.

Example 3.2 (The Monty-Hall-Problem)**.** The problem is this: There are three doors, behind one door is a very valuable prize like a car, behind the other two doors there are (less valuable) goats. The task is to find (and win) the one with car. The rule is that there is a first choice among the three doors, however, without opening the door. Then you obtain the specific information which one of the remaining other two doors contains a goat. Then you are allowed to choose again. The issue is whether it is better to stick to your first choice, or choose the other unopened door.

We represent winning the car by 0 and use 1 for losing (choosing one of the goats). We also encode the strategy of the player with numbers 0 and 1, where 0 means stay and 1 means changing the decision. Then the problem can be modeled as the following abstraction, that receives the strategy as input:

$$\begin{array}{lll} montyHall := \lambda strategy.\texttt{let}\ firstChoose = 0 \overset{1/3}{\oplus} 1 & & \text{// 1 of 3 doors contains the car} \\ \quad \texttt{in if}\ strategy & & \text{// stay} \\ \quad\quad \texttt{then}\ firstChoose & & \\ \quad\quad \texttt{else} & & \text{// change} \\ \quad\quad\quad \texttt{if}\ firstChoose & & \text{// if firstChoose was the car} \\ \quad\quad\quad\ \texttt{then}\ 1 & & \text{// switch to the goat} \\ \quad\quad\quad\ \texttt{else}\ 0 & & \text{// otherwise, switch from a goat to the car} \end{array}$$

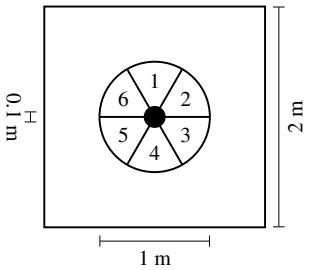
Then the equivalence $montyHall\ 0 \sim_d 0 \overset{1/3}{\oplus} 1$ holds, which means that we win with probability of $1/3$ and we lose with probability of $2/3$. For change our decision, we obtain the equivalence $montyHall\ 1 \sim_d 0 \overset{2/3}{\oplus} 1$, which means that we win with probability of $2/3$ and lose with probability of $1/3$. And thus the recommendation is to change the choice.

Example 3.3. As an example using real-valued probabilities, we consider a simplified dart board.

The dart board has a diameter of 1 meter. The board has six segments (of equal size), the bull's-eye in the middle of the board has a diameter of 0.1 meter. We assume that the board hangs on a wall of 2×2 meters.

Suppose that hitting the bull's-eye is worth 10 points, hitting the $i-th$ segment is worth i points, and hitting the wall gives 0 points.

The result of a random dart throw can be expressed in $ProbPCF^{need}_{\mathbb{R}}$ by the following program (given as an abstraction that ignores its argument):



A darts board on the wall

$$\begin{aligned} throwDart = \lambda d.\texttt{let } wall = 0 \texttt{ in let } segment = 1 \overset{1/6}{\oplus} (2 \overset{1/5}{\oplus} (3 \overset{1/4}{\oplus} (4 \overset{1/3}{\oplus} (5 \overset{1/2}{\oplus} 6)))) \\ \texttt{in let } bullseye = 10 \\ \texttt{in let } board = bullseye \overset{1/100}{\oplus} segment \texttt{ in } board \overset{\pi/16}{\oplus} wall \end{aligned}$$

The distribution of *throwDart* 0 is $(throwDart\ 0)_d(10) = \pi/1600$, $(throwDart\ 0)_d(i) = 99\pi/9600$ for $i = 1,2,3,4,5,6$, $(throwDart\ 0)_d(0) = 1 - \pi/16$ and $(throwDart\ 0)_d(i) = 0$ for $i \notin \{0,1,2,3,4,5,6,10\}$. The program *throwDart* can be used to perform other experiments, like randomly throwing 100 darts etc.

4 Conservativity of the Extension

Clearly, the language $ProbPCF^{need}_{\mathbb{R}}$ is an extension of $ProbPCF^{need}$, since the probability 0.5 can be represented. More formally, consider the sublanguage of $ProbPCF^{need}_{\mathbb{R}}$ where only $s \overset{0.5}{\oplus} t$ is allowed as prob-expression. This sublanguage is isomorphic to $ProbPCF^{need}$ with the same definition of expected (value) convergence and the same contextual semantics. However, as we show, in the sublanguage every $s \overset{r}{\oplus} t$-expression can be simulated by a recursive program and thus the sublanguage and the full language are equivalent w.r.t. contextual equivalence. This immediately implies that $ProbPCF^{need}$ and $ProbPCF^{need}_{\mathbb{R}}$ are equivalent w.r.t. contextual equivalence.

In abuse of notation, we say that expressions $s : nat \in ProbPCF^{need}$ and $t : nat \in ProbPCF^{need}_{\mathbb{R}}$ are distribution-equivalent if they induce the same probability distribution, i.e. for for all $i \in \mathbb{N}$: $s_d(i) = t_d(i)$.

Theorem 4.1. *For every stochastic program $s : nat$ in $ProbPCF^{need}_{\mathbb{R}}$ there exists a distribution-equivalent stochastic program $s' : nat$ in $ProbPCF^{need}$.*

The program s' will be defined by the following encoding *enc*, which we explain in detail before proving Theorem 4.1. The core idea of *enc* is to recursively perform fair coin-tosses for every bit of the bit-expansion of r, which is computable, since r is computable. This approach is well-known, for instance, in the setting of probabilistic Turing machines [4, Lemma 7.12].

Definition 4.2 (Encoding *enc*)**.** Let $r \in (0,1)$ be a computable real number. Let f_r be a computable function (given as a $ProbPCF^{need}$-abstraction not containing any $\oplus$-operator) that computes the bit expansion of r, i.e. we assume that

$$r = \sum_{i=1}^{\infty} \frac{a_i}{2^i} \text{ and that } f_r\ i = \begin{cases} 0, & \text{if } a_i = 1 \\ 1, & \text{if } a_i = 0 \end{cases}$$

Since r is computable and PCF with the fixpoint operator is Turing-complete, f_r exists. The expression $s \overset{r}{\oplus} t$ is encoded as follows:

$$\begin{aligned} enc(s \overset{r}{\oplus} t) = \texttt{let } f_r = \ldots \texttt{ in fix } \big(\lambda g,i,x,y.\texttt{if } (f_r\ i) \texttt{ then } x \oplus (g\ (\texttt{succ } i)\ x\ y) \\ \texttt{else } y \oplus (g\ (\texttt{succ } i)\ x\ y)\big)\ 1\ enc(s)\ enc(t) \end{aligned}$$

We extend the encoding to other program constructs by applying *enc* homomorphically over the term structure, i.e. $enc(F\ s_1\ \ldots\ s_n) = F\ enc(s_1)\ \ldots\ enc(s_n)$ for all language constructs $F \neq \overset{r}{\oplus}$.

Given $s \overset{r}{\oplus} t$, the encoded expression $enc(s \overset{r}{\oplus} t)$ recursively creates prob-expressions $x \oplus y$ where x and y are variables bound to $enc(s)$ and $enc(t)$, respectively. The encoding recursively inspects the bit expansion of r. For every bit position, it calls the function f_r to determine whether the current bit is 0 or 1 in the bit expansion of r. In both cases a prob-expression is generated. If the current bit is set, then the left argument is the variable x, otherwise, it is the variable y, the right argument is the recursive call for inspecting the next bit position of r. If we inspect any evaluation of the encoding, then we verify that it does recursive steps until it does a *probl*-step and then chooses x or y and then proceeds with the evaluation of $enc(s)$ or $enc(t)$, resp.

We sum the probabilities of all reduction sequences starting with the encoded expression and after the first *probl*-reduction. We split into two cases: if we include the sequences where the *probl*-reduction results in x, then it is the infinite sum $\sum_{i=1}^{\infty} \frac{a_i}{2^i}$ which is equal to r. If only the sequences are included where the *probl*-reduction results in y, then the probability is $\sum_{i=1}^{\infty} \frac{|1-a_i|}{2^i}$ which is equal to $1-r$.

Example 4.3. We first consider a rational probability. Let $m \neq n$. The expression $m \overset{1/3}{\oplus} n$ is encoded as

$$s = enc(m \overset{1/3}{\oplus} n) = \texttt{let}\ f_{1/3} = \lambda i.(i \bmod 2)\ \texttt{in}\ \texttt{fix}\left(\lambda g,i,x,y.\texttt{if}\ (f_{1/3}\ i)\ \texttt{then}\ x \oplus (g\ (\texttt{succ}\ i)\ x\ y)\ \texttt{else}\ y \oplus (g\ (\texttt{succ}\ i)\ x\ y)\right)\ 1\ m\ n$$

and unfolds to $n \oplus (m \oplus (n \oplus (m \oplus (n \oplus \ldots$. This shows $\text{ExVCv}(s,m) = \sum_{i\in\mathbb{N}} \frac{1}{2^{2(i+1)}} = \sum_{i\in\mathbb{N}} (\frac{1}{4})^{i+1} = \frac{1}{3}$ and $\text{ExVCv}(s,n) = \sum_{i\in\mathbb{N}} \frac{1}{2^{2i+1}} = \sum_{i\in\mathbb{N}} \frac{1}{2^i} - \sum_{i\in\mathbb{N}} \frac{1}{2^{2(i+1)}} = 1 - \frac{1}{3} = \frac{2}{3}$.

As an example with an irrational probability, consider $(a \overset{\tau}{\oplus} b)$ where τ is the Prouhet–Thue–Morse constant (see e.g. [7, Sect. 6.8]) and $a \neq b$ are natural numbers. Number τ is computable and the i^{th} bit of τ (for $i = 1,2,\ldots$) can be computed by $h\ (\texttt{pred}\ i)$ where h is recursively defined as $h = def_h$ with $def_h = \lambda i.\texttt{if}\ i\ \texttt{then}\ 0\ \texttt{else if}\ i \bmod 2\ \texttt{then}\ h\ (i\ \texttt{div}\ 2)\ \texttt{else}\ 1 - (h\ (i\ \texttt{div}\ 2))$. Thus:

$$s = enc(a \overset{\tau}{\oplus} b) = \texttt{let}\ f_\tau = \lambda i.\texttt{let}\ h = def_h\ \texttt{in}\ 1 - (h\ (\texttt{pred}\ i))\ \texttt{in}\ \texttt{fix}\left(\lambda g,i,x,y.\texttt{if}\ (f_\tau\ i)\ \texttt{then}\ x \oplus (g\ (\texttt{succ}\ i)\ x\ y)\ \texttt{else}\ y \oplus (g\ (\texttt{succ}\ i)\ x\ y)\right)\ 1\ a\ b$$

Unfolding s results in $(b \oplus (a \oplus (a \oplus (b \oplus (a \oplus (b \oplus (b \oplus (a \oplus \ldots$ and $\text{ExVCv}(s,a) = \tau$ and $\text{ExVCv}(s,b) = 1-\tau$. A problem where probability τ occurs is the following game: Alice and Bob repeatedly toss a fair coin taking turns according to the Thue-Morse-sequence [1]: Alice starts, then Bob. After that, the sequence is extended by the inverted sequence that has already been constructed (where inverted means exchanging Alice with Bob and vice versa). The sequence begins with Alice, Bob, Bob, Alice, Bob, Alice, Alice, Bob, and so on. The first person to flip heads wins. Then the probability that Bob wins is τ. Encoding Alice as 2 and Bob as 1, the above problem can be modeled as $(1 \overset{\tau}{\oplus} 2)$.

Our encoding *enc* applies the following two program equivalences (indefinitely) for $r' = 0.5$:

Lemma 4.4. *In* $ProbPCF^{need}_{\mathbb{R}}$ *the following equivalences hold:*

1. $s \overset{r}{\oplus} t \sim_c s \overset{r'}{\oplus} (s \overset{\frac{r-r'}{1-r'}}{\oplus} t)$ *if* $r' \leq r$ *and* $r' \in (0,1)$ *2.* $s \overset{r}{\oplus} t \sim_c t \overset{r'}{\oplus} (s \overset{\frac{r}{1-r'}}{\oplus} t)$ *if* $r' > r$ *and* $r' \in (0,1)$

The proof of the lemma is sketched in Appendix A). It uses a so-called context lemma for contextual equivalence, which is says that the inequation $\text{ExCv}(C[s]) \leq \text{ExCv}(C[t])$ holds, for all $s,t : \sigma$ and

contexts $C[\cdot_\sigma] : nat$ provide that $\forall k \geq 0$ and for all reduction contexts $R[\cdot_\sigma] : nat$ there exists $d \geq 0$, such that $\text{ExCv}(R[s],k) \leq \text{ExCv}(R[t],k+d)$. Here $\text{ExCv}(u,k)$ means the expected convergence of expression u where only at most k prob-reductions are allowed. The bound on the number of prob-reductions is helpful to perform inductive proofs.

We sketch the proof of Theorem 4.1. It uses the following two propositions. Both are proved in Appendix A. The first proposition shows the correctness of the encoding (w.r.t. distribution equivalence) for all closing reduction contexts.

Proposition 4.5. *Let R be a reduction context, s, t be prob-free expressions, $r \in (0,1)$ be a computable real number. Let $R[s \overset{r}{\oplus} t]$ be a stochastic program. Then $R[s \overset{r}{\oplus} t] \sim_d R[enc(s \overset{r}{\oplus} t)]$.*

The following proposition is similar to a context lemma for distribution-equivalence. It allows us to transfer the result on all reduction contexts to all contexts. In [21] we have shown a restricted form of the proposition in *ProbPCF*^{need}, where s and t had to be closed and of type *nat* (i.e. stochastic programs). We provide a generalisation for *ProbPCF*$^{need}_{\mathbb{R}}$ where s and t might be open.

Proposition 4.6. *Let $C[\cdot_{1,\sigma} \ldots, \cdot_{n,\sigma}] : nat$ be a context with n holes, $s,t : \sigma$ such that $C[s,\ldots,s]$ and $C[t,\ldots,t]$ are closed. If for $s,t : \sigma$, and all reduction contexts $R[\cdot_\sigma] : nat$ such that $R[s]$ and $R[t]$ are closed, we have $R[s] \sim_d R[t]$. Then $C[s,\ldots,s] \leq_d C[t,\ldots,t]$.*

We finally prove Theorem 4.1 by combining Propositions 4.5 and 4.6:

Proof of Theorem 4.1. Let s be a stochastic program with several prob-expressions. Then we can replace each prob-expression with its encoding, starting with the innermost expression. By Proposition 4.6, this replacement will preserve the distribution-equivalence, since the original expression and its encoding are distribution-equivalent by Proposition 4.5. □

5 Conclusion

We extended our previously introduced probabilistic call-by-need language with computable real-valued probabilities and have shown that the extension can be encoded in the smaller language without changing the distribution for closed programs of type *nat*.

A result from [4, Lemma 7.12] on probabilistic Turing machines can be reformulated for our setting as: The *expected* number of reduction steps and probabilistic fair choices to simulate a choice with real-valued probability can be estimated as $O(1)$ provided the number of steps for computation of the i^{th} bit is polynomial in i.

For future work we may investigate algorithmic approximations of probabilistic (closed) programs: this may include semantic approximations by restricting the number of prob-reductions in evaluations, or by restricting the number of sr-steps and then stopping with no result. This will approximate the real distribution of the program. We may also program this approximation, for instance, by providing approximating encodings $enc_\approx$ similar to *enc* that stop after performing a limit of prob-steps. In this case the programs can be compared w.r.t. distribution approximation, where $enc_\approx(s) \leq_d s$ should hold.

As a technical detail we should prove that distribution approximation is equivalent to contextual preorder (for stochastic programs). This proof is work in progress.

References

[1] J.-P. Allouche & J. O. Shallit (1998): *The Ubiquitous Prouhet-Thue-Morse Sequence*. In: *SETA 1998*, DMTCS, Springer, pp. 1–16, doi:10.1007/978-1-4471-0551-0_1.

[2] Z. M. Ariola & M. Felleisen (1997): *The call-By-need lambda calculus*. *JFP* 7(3), pp. 265–301.

[3] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky & P. Wadler (1995): *A call-by-need lambda calculus*. In: *POPL 1995*, ACM, pp. 233–246, doi:10.1145/199448.199507.

[4] Sanjeev Arora & Boaz Barak (2009): *Computational Complexity: A Modern Approach*. Cambridge University Press.

[5] G. Barthe, J.-P. Katoen & A. Silva, editors (2020): *Foundations of Probabilistic Programming*. Cambridge University Press, doi:10.1017/9781108770750.

[6] R. Crubillé & U. Dal Lago (2014): *On Probabilistic Applicative Bisimulation and Call-by-Value λ-Calculi*. In: *ESOP 2014*, *LNCS* 8410, Springer, pp. 209–228, doi:10.1007/978-3-642-54833-8_12.

[7] S. R. Finch (2005): *Mathematical Constants*. *Encyclopedia of Mathematics and its Applications* 94, Cambridge University Press.

[8] C. Jones & G. D. Plotkin (1989): *A Probabilistic Powerdomain of Evaluations*. In: *LICS 1989*, IEEE Computer Society, pp. 186–195, doi:10.1109/LICS.1989.39173.

[9] A. Kutzner & M. Schauß (1998): *A Non-Deterministic Call-by-Need Lambda Calculus*. In: *ICFP 1998*, ACM, pp. 324–335, doi:10.1145/289423.289462.

[10] U. Dal Lago (2020): *On Probabilistic Λ-Calculi*, pp. 121–144. Cambridge University Press, doi:10.1017/9781108770750.005.

[11] U. Dal Lago, D. Sangiorgi & M. Alberti (2014): *On coinductive equivalences for higher-order probabilistic functional programs*. In: *POPL 2014*, ACM, pp. 297–308, doi:10.1145/2535838.2535872.

[12] U. Dal Lago & M. Zorzi (2012): *Probabilistic operational semantics for the lambda calculus*. *RAIRO-Theor. Inf. Appl.* 46(3), pp. 413–450, doi:10.1051/ita/2012012.

[13] J. C. Mitchell (1991): *On Abstraction and the Expressive Power of Programming Languages*. In: *TACS '91*, *LNCS* 526, Springer, pp. 290–310, doi:10.1007/3-540-54415-1_51.

[14] A. Moran, D. Sands & M. Carlsson (2003): *Erratic Fudgets: a semantic theory for an embedded coordination language*. *Sci. Comput. Program.* 46(1-2), pp. 99–135, doi:10.1016/S0167-6423(02)00088-6.

[15] G. D. Plotkin (1977): *LCF Considered as a Programming Language*. *Theor. Comput. Sci.* 5(3), pp. 223–255, doi:10.1016/0304-3975(77)90044-5.

[16] J. G. Riecke (1991): *Fully Abstract Translations between Functional Languages*. In: *POPL 1991*, ACM, pp. 245–254, doi:10.1145/99583.99617.

[17] D. Sabel & M. Schmidt-Schauß (2008): *A Call-by-Need Lambda-Calculus with Locally Bottom-Avoiding Choice: Context Lemma and Correctness of Transformations*. *Math. Structures Comput. Sci.* 18(03), pp. 501–553, doi:10.1017/S0960129508006774.

[18] D. Sabel, M. Schmidt-Schauß & L. Maio (2022): *Contextual Equivalence in a Probabilistic Call-by-Need Lambda-Calculus*. In: *PPDP 2022*, ACM, pp. 4:1–4:15, doi:10.1145/3551357.3551374.

[19] D. Sabel, M. Schmidt-Schauß & L. Maio (2022): *A Probabilistic Call-by-Need Lambda-Calculus – Extended Version*. *CoRR*, doi:10.48550/ARXIV.2205.14916.

[20] N. Saheb-Djahromi (1978): *Probabilistic LCF*. In: *MFCS 1978*, *LNCS* 64, Springer, pp. 442–451, doi:10.1007/3-540-08921-7_92.

[21] M. Schmidt-Schauß & D. Sabel (2023): *Program equivalence in a typed probabilistic call-by-need functional language*. *J. Log. Algebraic Methods Program.* 135, p. 100904, doi:10.1016/J.JLAMP.2023.100904.

A Proofs

For an expression $s \in ProbPCF^{need}_{\mathbb{R}}$, let $\mathrm{ExCv}(s,k)$ and $\mathrm{ExVCv}(s,n,k)$ be the *expected (value) convergence of s restricted to evaluations that do not use more than k prob-reductions*, i.e.

$$\mathrm{ExCv}(s,k) = \sum_{S \in Eval(s), |S| \le k} \mathsf{P}(S) \text{ and } \mathrm{ExVCv}(s,n,k) = \sum_{S \in Eval(s), val(WHNF(s,S)) = m, |S| \le k} \mathsf{P}(S)$$

Since $(\mathrm{ExCv}(s,k))_{k=0}^{\infty}$ and $(\mathrm{ExVCv}(s,n,k))_{k=0}^{\infty}$ are monotonically increasing and bounded by $\mathrm{ExCv}(s)$, or $\mathrm{ExVCv}(s,n)$, we have $\lim_{k\to\infty} \mathrm{ExCv}(s,k) = \mathrm{ExCv}(s)$ and $\lim_{k\to\infty} \mathrm{ExVCv}(s,n,k) = \mathrm{ExVCv}(s,n)$.

The following context lemma was proved for $ProbPCF^{need}$, but also holds in $ProbPCF^{need}_{\mathbb{R}}$:

Theorem A.1 (Context Lemma)**.** *Let σ be a type, let $s,t : \sigma$ such that $\forall k \ge 0$ and for all reduction contexts $R[\cdot_\sigma] : nat$ there exists $d \ge 0$, such that $\mathrm{ExCv}(R[s],k) \le \mathrm{ExCv}(R[t],k+d)$, and let $C[\cdot_\sigma] : nat$ be a context. Then the inequation $\mathrm{ExCv}(C[s]) \le \mathrm{ExCv}(C[t])$ holds.*

Proof of Lemma 4.4. We demonstrate the proof for the first equivalence. Let $r \in (0,1)$, $r' \in (0,1)$, $r' \le r$ and $s,t : \sigma$. Let $R[\cdot_\sigma] : nat$ be a reduction context. We have to show two directions. First let $k \ge 0$ and $\mathrm{ExCv}(R[s \overset{r}{\oplus} t],k) = p$. Then we choose $d = 1$ and show $\mathrm{ExCv}(s \overset{r'}{\oplus} (s \overset{\frac{r-r'}{1-r'}}{\oplus} t), k+d) = q \ge p$. First, consider the case $k = 0$. Then $p = 0$ and $p \le q$ holds. Now assume $k > 0$. Then

$$\begin{aligned} p = \mathrm{ExCv}(R[s \overset{r}{\oplus} t],k) &= r\mathrm{ExCv}(R[s],k-1) + (1-r)\mathrm{ExCv}(R[t],k-1) \\ &\le r\mathrm{ExCv}(R[s],k-1) + (1-r)\mathrm{ExCv}(R[t],k-1) \\ &\quad + r'\mathrm{ExCv}(R[s],k) - r'\mathrm{ExCv}(R[s],k-1) = \mathrm{ExCv}(R[s \overset{r'}{\oplus} (s \overset{\frac{r-r'}{1-r'}}{\oplus} t)],k+1) = q \end{aligned}$$

The context lemma shows $s \overset{r}{\oplus} t \le_c s \overset{r'}{\oplus} (s \overset{\frac{r-r'}{1-r'}}{\oplus} t)$. For the other direction of the claim, we show $q' = \mathrm{ExCv}(R[s \overset{r'}{\oplus} (s \overset{\frac{r-r'}{1-r'}}{\oplus} t)],k) \le \mathrm{ExCv}(s \overset{r}{\oplus} t,k) = p'$ for all k. For $k \ge 2$, we have

$$\begin{aligned} q' &= \mathrm{ExCv}(R[s \overset{r'}{\oplus} (s \overset{\frac{r-r'}{1-r'}}{\oplus} t)],k) \\ &= r\mathrm{ExCv}(R[s],k-2) + (1-r)\mathrm{ExCv}(R[t],k-2) + r'\mathrm{ExCv}(R[s],k-1) - r'\mathrm{ExCv}(R[s],k-2) \\ &\le r\mathrm{ExCv}(R[s],k-1) + (1-r)\mathrm{ExCv}(R[t],k-1) + r'\mathrm{ExCv}(R[s],k-1) - r'\mathrm{ExCv}(R[s],k-1) \\ &= r\mathrm{ExCv}(R[s],k-1) + (1-r)\mathrm{ExCv}(R[t],k-1) = \mathrm{ExCv}(R[s \overset{r}{\oplus} t],k) = p' \end{aligned}$$

For $k = 0$, the inequation $q' \le p'$ also holds, since $q' = 0$. For $k = 1$, the inequation $q' \le p'$ also holds, since $\mathrm{ExCv}(R[s \overset{r'}{\oplus} (s \overset{\frac{r-r'}{1-r'}}{\oplus} t)],k) = r'\mathrm{ExCv}(R[s],0) \le r\mathrm{ExCv}(R[s],0) + (1-r)\mathrm{ExCv}(R[t],0) = \mathrm{ExCv}(R[s \overset{r}{\oplus} t],1)$ since $r' \le r$. Thus, the context lemma shows $s \overset{r'}{\oplus} (s \overset{\frac{r-r'}{1-r'}}{\oplus} t) \le_c s \overset{r}{\oplus} t$. □

The following lemmas show how to transfer results with bounds to the limit. The proofs are straightforward (see [18, 19] for similar proofs).

Lemma A.2. *Let $s,t : \tau$. (i) If $\forall k \ge 0 : \exists d : \mathrm{ExCv}(s,k) \le \mathrm{ExCv}(t,k+d)$, then $\mathrm{ExCv}(s) \le \mathrm{ExCv}(t)$.*
(ii) If $\forall k \ge 0 : \exists d : \mathrm{ExVCv}(s,n,k) \le \mathrm{ExVCv}(t,n,k+d)$, then $\mathrm{ExVCv}(s,n) \le \mathrm{ExVCv}(t,n)$.

Lemma A.3. *Let $s,t : \tau$. (i) If $\forall k \ge 0 : \mathrm{ExCv}(s,k) \le \mathrm{ExCv}(t)$ and for every $\varepsilon > 0$ there exists $k_\varepsilon \ge 0$ with $\mathrm{ExCv}(t) - \mathrm{ExCv}(s,k_\varepsilon) < \varepsilon$, then $\mathrm{ExCv}(s) = \mathrm{ExCv}(t)$.*
(ii) If $\forall k \ge 0 : \mathrm{ExVCv}(s,n,k) \le \mathrm{ExVCv}(t,n)$ and for every $\varepsilon > 0$ there exists $k_\varepsilon \ge 0$ with $\mathrm{ExVCv}(t,n) - \mathrm{ExVCv}(s,n,k_\varepsilon) < \varepsilon$, then $\mathrm{ExVCv}(s,n) = \mathrm{ExVCv}(t,n)$.

Proof of Proposition 4.5. For proving distribution equivalence, it suffices to show $\mathrm{ExVCv}(R[s \overset{r}{\oplus} t], n) = \mathrm{ExVCv}(R[enc(s \overset{r}{\oplus} t)], n)$ for all $n \in \mathbb{N}$. Let n be arbitrary but fixed. Using Lemma A.3 it suffices to show

1. For all $k \in \mathbb{N}$: $\mathrm{ExVCv}(R[enc(s \overset{r}{\oplus} t)], n, k) \le \mathrm{ExVCv}(R[s \overset{r}{\oplus} t], n)$
2. For every $\varepsilon > 0$ there exists k_ε: $\mathrm{ExVCv}(R[s \overset{r}{\oplus} t], n) - \mathrm{ExVCv}(R[enc(s \overset{r}{\oplus} t)], n, k_\varepsilon) < \varepsilon$

We prove Item 1: In any reduction context R the first sr-step of $R[s \overset{r}{\oplus} t]$ reduces the prob-expression in R. This justifies the equation $\mathrm{ExVCv}(R[s \overset{r}{\oplus} t, n) = r \cdot \mathrm{ExVCv}(R[s], n) + (1-r) \cdot \mathrm{ExVCv}(R[t], n)$. We now show $\mathrm{ExVCv}(R[enc(s \overset{r}{\oplus} t)], n, k) \le \mathrm{ExVCv}(R[s \overset{r}{\oplus} t], n)$ by induction on k: If $k = 0$, then $\mathrm{ExVCv}(R[enc(s \overset{r}{\oplus} t)], n, 0) = 0$. If $k > 0$, then the reduction of $R[enc(s \overset{r}{\oplus} t)]$ reduces $enc(s \overset{r}{\oplus} t)$. The recursive function g is called and a *probl-* or *probr*-step is made. There are two cases:

- $r \ge 0.5$: the function g of the encoding *enc* inspects $f_r(i)$ starting from the current $i = 1$. Since $r \ge 0.5$ the first bit of of the bit-expansion of r is 1 and thus for the recursive call the bit is removed and i is increased by one. This means to shift the number to the next bit. We verify that this is equivalent to subtracting 0.5 from r and then multiplying the result by 2 (for the shift). Thus:

$$\begin{aligned}
&\mathrm{ExVCv}(enc(s \overset{r}{\oplus} t), n, k) = 0.5 \cdot \mathrm{ExVCv}(R[s], n, k-1) + 0.5 \cdot (R[enc(s \overset{r}{\oplus} t)], n, k-1) \\
\overset{I.H.}{\le}\ & 0.5 \cdot \mathrm{ExVCv}(R[s], n, k-1) + 0.5 \cdot (\mathrm{ExVCv}(R[s \overset{r}{\oplus} t], n)) \\
\le\ & 0.5 \cdot \mathrm{ExVCv}(R[s], n) + 0.5 \cdot (\mathrm{ExVCv}(R[s \overset{2(r-0.5)}{\oplus} t], n)) \\
=\ & 0.5 \cdot \mathrm{ExVCv}(R[s], n) + 0.5\big(2(r-0.5) \cdot \mathrm{ExVCv}(R[s], n) + (1-2(r-0.5)) \cdot \mathrm{ExVCv}(R[t], n)\big) \\
=\ & 0.5 \cdot \mathrm{ExVCv}(R[s], n) + (r-0.5)\mathrm{ExVCv}(R[s], n) + (1-r)\mathrm{ExVCv}(R[t], n) \\
=\ & r \cdot \mathrm{ExVCv}(R[s], n) + (1-r) \cdot \mathrm{ExVCv}(R[t], n) = \mathrm{ExVCv}(R[s \overset{r}{\oplus} t], n)
\end{aligned}$$

- $r < 0.5$: The recursive call of g only increases i by one, which means shifting the bit representation of r by one or equivalently multiplying r by 2. So in this case we compute as follows:

$$\begin{aligned}
&\mathrm{ExVCv}(enc(s \overset{r}{\oplus} t, n, k)) = 0.5 \cdot \mathrm{ExVCv}(R[t], n, k-1) + 0.5 \cdot (R[enc(s \overset{2r}{\oplus} t)], n, k-1) \\
\overset{I.H.}{\le}\ & 0.5 \cdot \mathrm{ExVCv}(R[t], n, k-1) + 0.5 \cdot (\mathrm{ExVCv}(R[s \overset{2r}{\oplus} t], n)) \\
\le\ & 0.5 \cdot \mathrm{ExVCv}(R[t], n) + 0.5 \cdot (\mathrm{ExVCv}(R[s \overset{2r}{\oplus} t], n)) \\
=\ & 0.5 \cdot \mathrm{ExVCv}(R[t], n) + 0.5(2r\mathrm{ExVCv}(R[s], n) + (1-2r) \cdot \mathrm{ExVCv}(R[t], n)) \\
=\ & 0.5 \cdot \mathrm{ExVCv}(R[t], n) + r\mathrm{ExVCv}(R[s], n) + (0.5-r)\mathrm{ExVCv}(R[t], n) \\
=\ & r \cdot \mathrm{ExVCv}(R[s], n) + (1-r) \cdot \mathrm{ExVCv}(R[t], n) = \mathrm{ExVCv}(R[s \overset{r}{\oplus} t], n)
\end{aligned}$$

For Item 2, we have: for $\delta > 0$, there exists k_δ such that $\mathrm{ExVCv}(R[s \overset{r}{\oplus} t], n) - \mathrm{ExVCv}(R[s \overset{r}{\oplus} t], n, k) < \delta$, since $\lim_{k \to \infty} \mathrm{ExVCv}(R[s \overset{r}{\oplus} t], n, k) = \mathrm{ExVCv}(R[s \overset{r}{\oplus} t], n)$. With $\mathrm{ExVCv}(R[enc(s \overset{r}{\oplus} t), n, (k_1, k_2)])$ we denote the expected value convergence of $R[enc(s \overset{r}{\oplus} t)]$ where at most k_1 prob-reductions are allowed for evaluating $enc(s \overset{r}{\oplus} t)$ and at most $k_2 + 1$ prob-reductions are allowed for other prob-reductions, i.e. inside R, perhaps using the result of $enc(s \overset{r}{\oplus} t)$. Then $\forall k_1, k_2 : \mathrm{ExVCv}(R[enc(s \overset{r}{\oplus} t), n, (k_1, k_2)]) \le \mathrm{ExVCv}(R[s \overset{r}{\oplus} t], k_2)$, since s and t are prob-free. Assume that k_2 is fixed. Then $\mathrm{ExVCv}(R[s \overset{r}{\oplus} t], k_2) - \mathrm{ExVCv}(R[enc(s \overset{r}{\oplus} t), n, (k_1, k_2)]) < \frac{1}{2^{k_1}}$, since $enc(s \overset{r}{\oplus} t)$ can evaluate the k_1 bits of the bit-expansion of r. This shows that for every $\xi > 0$ there exists a k_ξ with $\mathrm{ExVCv}(R[s \overset{r}{\oplus} t], k_2) - \mathrm{ExVCv}(R[enc(s \overset{r}{\oplus} t), n, (k_\xi, k_2)]) < \xi$. This implies $\lim_{k_1 \to \infty} \mathrm{ExVCv}(R[enc(s \overset{r}{\oplus} t)], n, (k_1, k_2)) = \mathrm{ExVCv}(R[s \overset{r}{\oplus} t], k_2)$.

Finally, let $\delta = \varepsilon/2$, then there exists k_2 with $\text{ExVCv}(R[s \overset{r}{\oplus} t], n) - \text{ExVCv}(R[s \overset{r}{\oplus} t], n, k_2) < \varepsilon/2$. Let $\xi = \varepsilon/2$ there exists a k_3 with $\text{ExVCv}(R[s \overset{r}{\oplus} t], k_2) - \text{ExVCv}(R[enc(s \overset{r}{\oplus} t), n, (k_3, k_2)]) < \varepsilon/2$ and thus $\text{ExVCv}(R[s \overset{r}{\oplus} t], n) - \text{ExVCv}(R[enc(s \overset{r}{\oplus} t)], n, (k_3, k_2)) < \varepsilon$. Since $\text{ExVCv}(R[enc(s \overset{r}{\oplus} t)], n, (k_3, k_2))$ $< \text{ExVCv}(R[enc(s \overset{r}{\oplus} t)], n, k_3 + k_2) < \text{ExVCv}(R[enc(s \overset{r}{\oplus} t)], n) < \text{ExVCv}(R[s \overset{r}{\oplus} t], n)$, this also shows $\text{ExVCv}(R[s \overset{r}{\oplus} t], n) - \text{ExVCv}(R[enc(s \overset{r}{\oplus} t)], n, k_3 + k_2) < \varepsilon$. □

Proof of Proposition 4.6. Let $u = C[s, \ldots, s]$ and $v = C[t, \ldots, t]$. From Lemma A.2 it suffices to show for all $n \in nat$: for all $\forall k : \text{ExVCv}(u, n, k) \le \text{ExVCv}(v, n)$. For the remaining proof, we fix s, t, and the maximal number k of prob-reductions permitted in evaluations of $u = C[s, \ldots, s]$. With $v = C[t, \ldots, t]$, we show that $\text{ExVCv}(u, n, k) \le \text{ExVCv}(v, n)$ for all closing C. The proof is by induction on (i) the maximal number k of prob-reductions of $C[s, \ldots, s]$, (ii) the number of all reduction steps of $C[s, \ldots, s]$ until the next prob-reduction; and 0 if the reduction does not contain any prob-reduction steps; and (iii) the number of holes of the context C. We consider evaluations (resp. sr-steps) of the respective expressions.

- One base case is that $C[\cdot, \ldots, \cdot]$ is a WHNF and there are no sr-steps. Then $C[s, \ldots, s]$ as well as $C[t, \ldots, t]$ are WHNFs. Due to the type, these are the same natural number (in an LR-context). Hence we have $\text{ExVCv}(C[s, \ldots, s], n) = \text{ExVCv}(C[t, \ldots, t], n])$.
- There is no evaluation of $C[s, \ldots, s]$ under the restriction on k and without prob-reduction steps: for example it may be non-terminating or get stuck since a prob-expression cannot be evaluated. Then the contribution of $C[s, \ldots, s]$ is 0, and hence $\text{ExVCv}(C[s, \ldots, s], n, k) \le \text{ExVCv}(C[t, \ldots, t], n)$.
- If $C[s, \ldots, \cdot_{i,\sigma}, \ldots, s]$ is a reduction context for some i. Then there is j such that $C[r_1, \ldots, \cdot_{j,\sigma}, \ldots, r_n]$ is a reduction context for all expressions r_k. We investigate the two pairs
  - $C[s, \ldots, [s]_j, \ldots, s]$ and $C[t, \ldots, t, [s]_j, t \ldots, t]$,
  - $C[t, \ldots, t, [s]_j, t \ldots, t]$ and $C[t, \ldots, [t]_j, \ldots, t]$.

  The precondition on $R[s] \sim_d R[t]$ shows that the expressions of the latter pair are equivalent w.r.t. $\sim_d$. For the first pair, we use the context $C[\cdot, \ldots, [s]_j, \ldots, \cdot]$ with $n - 1$ holes and use the induction hypothesis (on the number of holes) to show that $\text{ExVCv}(C[s, \ldots, [s]_j, \ldots, s], n, k) \le$ $\text{ExVCv}(C[t, \ldots, t, [s]_j, t \ldots, t], n)$. Applying the inequation $\text{ExVCv}(C[t, \ldots, t, [s]_j, t \ldots, t], n) \le$ $\text{ExVCv}(C[t, \ldots, t, [t]_j, t \ldots, t], n)$, we conclude $\text{ExVCv}(C[s, \ldots, s], n, k) \le \text{ExVCv}(C[t, \ldots, t], n)$.
- $C[s, \ldots, [\cdot_{i,\sigma}], \ldots, s]$ is not a reduction context for any hole i.

  Then the same holds for $C[t, \ldots, [\cdot_{i,\sigma}], \ldots, t]$. If there is no sr-reduction step, then the expected convergence of u and v is 0. If expression u has an *sr*-reduction step, then the same reduction step at the same position can be used for $C[t, \ldots, t]$. There are two cases:
  - The *sr*-reduction is not a prob-reduction, and results are $C'[s, \ldots, s]$ and $C'[t, \ldots, t]$, resp. We can use induction on the number of sr-steps of $C'[s, \ldots, s]$ to obtain the desired inequality.
  - The sr-step of $C[s, \ldots, s]$ is a prob-reduction with $r \in \mathbb{R}$. We apply the same prob-reduction to $C[t, \ldots, t]$. We obtain $C'[s, \ldots, s]$ and $C'[t, \ldots, t]$, with a contribution of r and $1 - r$ to the computed distribution. The other possibility of the prob-reduction results in $C''[s, \ldots, s]$ and $C''[t, \ldots, t]$. Induction on the number of prob-reductions shows $\text{ExVCv}(C'[s, \ldots, s], n, k - 1) \le \text{ExVCv}(C'[t, \ldots, t], n)$, $\text{ExVCv}(C''[s, \ldots, s], n, k - 1) \le \text{ExVCv}(C''[t, \ldots, t], n)$, resp. and so

$$\begin{aligned} & r \cdot \text{ExVCv}(C'[s, \ldots, s], n, k-1) + (1-r) \cdot \text{ExVCv}(C''[s, \ldots, s], n, k-1) \\ \le\ & r \cdot \text{ExVCv}(C'[t, \ldots, t], n) - (1-r) \cdot \text{ExVCv}(C''[t, \ldots, t], n) \\ =\ & \text{ExVCv}(C[t, \ldots, t], n) \end{aligned}$$

□

Towards a verified compiler for Distributed PlusCal

Ghilain Bergeron Horatiu Cirstea Stephan Merz
Université de Lorraine, CNRS, Inria, LORIA,
F-54000 Nancy, France

Formal verification techniques are frequently used to ensure correctness properties of distributed systems and algorithms. However, languages used for formal modeling and verification are often substantially different from languages used in software development, and verifying an abstract representation of an algorithm does not ensure that its handwritten implementation will be correct. This paper presents work in progress on a verified compiler for an extension of Lamport's PlusCal with threads and communication channels. Its syntax and semantics are formalized in the Lean 4 proof assistant and the passes compiling PlusCal algorithms into Go code are implemented in the underlying Lean 4 programming language. This paper gives formal semantics for the first pass of the compiler and outlines its mechanically verified correctness proof.

1 Introduction

Distributed systems and the distributed algorithms that these systems implement are notoriously difficult to design and to verify. This is due to the high number of potential executions that interleave steps of system components (distributed nodes, threads, messaging subsystem) executing independently, leading to bugs that are difficult to reproduce. Formal verification techniques such as model checking or theorem proving help ensure correctness properties of algorithms and of programs. They can be applied at different levels of abstraction. In particular, verifying formal specifications of distributed algorithms at high levels of abstraction allows designers to identify errors that would be very costly to correct during later development stages. However, languages used for formal modeling and verification are often substantially different from languages used in software development. Moreover, verifying an abstract representation of an algorithm does not ensure that its implementation will be correct if the translation from a verified specification to a concrete implementation is manual and thus, error-prone and potentially introducing subtle bugs.

Whereas the translation of sequential programs, including formal compiler verification, is nowadays a well-understood problem [Ler09], similar techniques for concurrent and distributed programs have been studied to a much lesser extent. Preliminary work exists for Erlang [ZBF20, HB24], DistAlgo [LSL17] (a domain-specific language extension of Python), Modular PlusCal [HHC+23] (a superset of PlusCal), but the full formal verification of such translations has not been one of the primary goals. The verification aspect is more significant in Verdi [WWP+15], a framework for implementing distributed systems using OCaml and formally verifying their properties in Coq.

A step towards bridging the gap between the formal specification and the corresponding implementation is the use of modeling languages that are closer to programming languages. In the context of TLA+ [Lam02], a specification formalism that is based on mathematical set theory and temporal logic, PlusCal [Lam09] was proposed as an algorithmic language intended for the description of concurrent and distributed algorithms and systems. PlusCal combines the "look and feel" of imperative pseudo-code with the full power of mathematical set theory, used for modeling the data structures manipulated by the algorithm. It is therefore well suited to modeling systems at an intermediate level between abstract TLA+

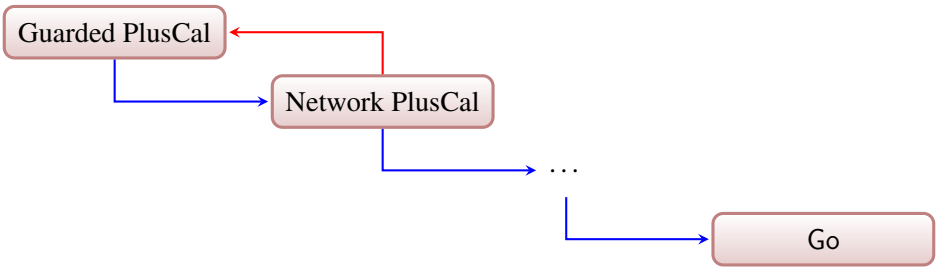


Figure 1: Outline of the compilation.

specifications and executable code. PlusCal algorithms are translated to TLA$^+$ specifications and can be verified using the TLA$^+$ model checkers [YML99, KKT19] or the proof assistant TLAPS [CDL$^+$12], thus allowing a system designer to obtain high confidence in the correctness of the algorithm.

This work specifically targets Distributed PlusCal [CM23], an extension of PlusCal for describing distributed algorithms. The overall objective is to design a verified compiler from Distributed PlusCal to Go. Following standard practice, our compiler is structured as a sequence of transformation passes. Each pass introduces more concrete intermediate representations, systematically bridging the semantic gap between the high-level input and the target executable language by introducing refinements of the semantics of the original specification. The compiler is developed in the Lean 4 proof assistant [MU21], which serves both as a functional programming language and as a proof environment. Figure 1 illustrates the compilation phases. As we will explain in Section 3, the compiler takes as input programs in the Guarded PlusCal fragment of Distributed PlusCal. The downward (blue) arrows in the diagram represent transformations that have been programmed in Lean. The main contribution of this paper are the formal definition of the semantics of Guarded PlusCal and thus of Network PlusCal, which is a fragment of Guarded PlusCal, and a mechanized correctness proof of the first phase of compilation, represented by the upward (red) arrow in the diagram. We expect to apply the same overall methodology for the remaining phases.

Outline. We introduce in Section 2 Distributed PlusCal and present in Section 3 the formal syntax and semantics of Guarded PlusCal. Section 4 describes the first pass of the compiler and its correctness proof. We finally discuss the next steps in the development of the compiler.

2 Distributed PlusCal

TLA$^+$ [Lam02] is a formalism for describing algorithms and systems at a high level of abstraction. It is based on mathematical set theory for representing data structures in terms of sets and functions, and on the Temporal Logic of Actions (TLA) for representing executions of systems. TLA$^+$ specifications usually consist of a predicate describing the possible initial states, a predicate that represents the possible state transitions (w.r.t. all state variables appearing in the specification), and some liveness or fairness property expressed as a formula of temporal logic. The formal verification of TLA$^+$ specifications is supported by the explicit-state model checker TLC, the SMT-based symbolic model checker Apalache, and the proof assistant TLAPS. Konnov et al. [KKM22] present an overview of the different verification tools applied to a common case study.

PlusCal [Lam09] was designed as a language for describing algorithms, providing a syntax that resembles imperative pseudo-code, including a process abstraction for modeling concurrent programs.

It relies on the rich language of expressions of TLA$^+$, which allows for a high level of expressiveness, but also means that PlusCal algorithms are in general not executable. Distributed PlusCal [CM23] extends PlusCal with threads and also with communication channels. Both extensions are relevant for describing distributed algorithms: nodes of distributed systems are modeled by Distributed PlusCal processes and communicate by message passing, and they may contain threads that share access to the node's local memory. For example, a node could have a main thread of execution and a second thread for handling incoming messages in the background. The (Distributed) PlusCal translator generates a TLA$^+$ specification from the algorithm and inserts it within the module containing the algorithm. Properties of algorithms are expressed as TLA$^+$ formulas and are verified using the standard TLA$^+$ tools.

The overall structure of a Distributed PlusCal algorithm consists of sections that declare global variables, operators, macros, and procedures (each of which may be empty) and a final section which contains either process declarations as shown in Listing 2.1 or a single body of statements representing a sequential algorithm.

Distributed PlusCal statements include `skip` (which does nothing), assignments, conditional statements, while loops, procedure calls, and `goto`. Processes and threads may synchronize using the `await` instruction that blocks until a predicate becomes true. Distributed PlusCal also includes two forms of non-deterministic control structures: `either` ... `or` ... can be used to introduce a choice between a fixed number of alternative branches, whereas the statement `with` $x \in S$ expresses a choice among the values in a set S. In particular, guarded commands can be expressed by combining `either` ... `or` and `await` statements. Channels are declared using the keyword `fifo`; messages will be received in the order in which they are sent. The operation `send`$(chan, expr)$ sends a message corresponding to the expression *expr* on channel *chan* and the operation `receive`$(chan, var)$ receives a message from channel *chan* and stores it in the variable *var*.

An important concern when describing concurrent algorithms is to model the "grain of atomicity", i.e., which statements are assumed to execute without interleaving with statements of other processes. PlusCal (and Distributed PlusCal) use statement labels to this effect: all statements appearing between two labels are executed atomically.

As an example, Listing 2.1 shows a simple Distributed PlusCal algorithm that implements a ping-pong protocol between two processes. The process `Pong` sends a message `"Ping"` on channel `ping` and waits for a message `"Pong"` on channel `pong` before starting again from the beginning, while process `Ping` proceeds symmetrically. The `await` statements are not really useful for this simple example, and users would typically use `while` loops instead of low-level `goto` statements. The example is written in this way in order to illustrate key aspects in the compilation strategy. The algorithm of Listing 2.1 introduces single instances (with identifiers 0 and 1) of the two process types. Alternatively, a fixed number of instances may be introduced by writing `process (p \in S)` where `S` must evaluate to a finite set. For the purpose of compiling towards an executable typed language, we assume that channels and variables are ascribed a type annotation (not shown in this listing) as defined by the Snowcat typechecker [KKT19] and that each process instance is assigned a dedicated channel, referred to as its mailbox, the only one through which it receives messages.

3 The Guarded PlusCal fragment

As mentioned in the previous section, the atomicity of statements (in a block) is a central concern when describing concurrent algorithms. Blocking statements like `await` and `receive` are one source of complexity when compiling interleaving blocks towards executable code. For example, an `await` statement

```
--algorithm PingPong {
    fifos ping, pong;

    (* @mailbox: ping; *)
    process (Ping = 0)
        variable tmp1 = "";
    {
    rcv1:   receive(ping, tmp1);
            await tmp1 = "Ping";
            goto pong;
    pong:   send(pong, "Pong");
            goto rcv1;
    }
                                        }
    (* @mailbox: pong; *)
    process (Pong = 1)
        variable tmp2 = "";
    {
    ping:   send(ping, "Ping");
            goto rcv2;
    rcv2:   receive(pong, tmp2);
            await tmp2 = "Pong";
            goto ping;
    }   }
```

Listing 2.1: Ping-pong algorithm in Distributed PlusCal

is executed only if the expression it contains evaluates to `TRUE`, in which case the control is passed to the next statement and otherwise, the execution of the containing block is aborted and tried again later. A direct implementation of such a behavior in executable code would require, in case of failure, rolling back all changes performed from the beginning of the block until the `await` statement. Indeed, this is the strategy employed in PGo, the compiler for Modular PlusCal, but provisioning for failure and potential rollback introduces significant overhead at execution.

We restrict here to a fragment of Distributed PlusCal that we call Guarded PlusCal that imposes several restrictions. First, Guarded PlusCal algorithms may contain process-local variables, but no global variables other than channel declarations: processes communicate exclusively by message passing. Second, a process receives messages only from a single channel, its mailbox[1]. Third and most importantly, any `receive` and `await` statements must be placed at the top of their respective blocks. This restriction by itself is not sufficient for avoiding rollbacks: for example, in the case of two `receive` operations, the second one might block, requiring the first one to be undone. However, we will see in Section 4 that it is a stepping stone for allowing us to avoid cancellation mechanisms built into the compiled program. Although the constraints might appear quite restrictive, we believe that Distributed PlusCal algorithms can generally be rewritten to satisfy them. For example, we checked that this is true for the algorithms contained in the TLA$^+$ examples repository [LKM$^+$]. Our Ping-Pong example is already written in Guarded PlusCal. For simplicity, we further assume that `while` statements are replaced with equivalent `if` and `goto` statements, and that `if` statements are rewritten as guarded commands, using a combination of `either ... or` and `await`.

3.1 Syntax of Guarded PlusCal

As for Distributed PlusCal, a Guarded PlusCal algorithm is composed of a section containing declarations (including the declaration of communication channels), and a section containing the different processes. Each process consists of some local variable declarations followed by the definition of the behavior of each of its threads, where each thread corresponds to a list of atomic blocks. Each block groups all the *guard* statements at the beginning and contains a final `goto` statement at the end. Note that there are no global variables in Guarded PlusCal; processes communicate exclusively through channels.

[1] Observe that channel declarations could therefore be omitted, and the process identity could be used instead of the mailbox channel. We do not do this here so that Guarded PlusCal algorithms remain valid Distributed PlusCal algorithms.

P	::=	$p \star (x = e)^* \star (\{(l : \texttt{either}\ B_1\ \texttt{or}\ \ldots\ \texttt{or}\ B_n)^*\})^*$	Process
B	::=	$G^*; E^*; \texttt{goto}\ l$	Atomic block
G	::=	$\texttt{await}\ e$	Precondition
	|	$\texttt{receive}(c, x)$	Message reception
	|	$\texttt{with}\ x = e \mid \texttt{with}\ x \in e$	Block-local binding
E	::=	$\texttt{skip}$	Inaction
	|	$x := e$	Local variable assignment
	|	$\texttt{print}\ e$	Local value output
	|	$\texttt{assert}\ e$	Dynamic assertion
	|	$\texttt{send}(c, e)$	Message sending

where $l \in \mathit{Label} \uplus \{\texttt{Done}\}$ denotes labels, $x, c \in \mathit{Var}$ denote (channel) variables and e denotes (non-temporal) TLA$^+$ expressions. Channels and variables can be indexed.

A process, identified by its identifier p[2], consists of a declaration of local variables—each initialized by an expression—and a list of threads, where each thread is defined as a sequence of labeled non-deterministic choices of atomic blocks (as will be discussed in the next section, any of the alternative branches can be selected for execution). For the sake of simplicity, we write $l_i : B_i$ instead of $l_i : \texttt{either}\ B_i$ when there is only one block. We denote by $\mathit{mailbox}_p$ the unique mailbox on which (the threads of) process p can receive messages. For the algorithm in the Listing 2.1, the process identifiers are respectively 0 and 1, and their mailboxes are respectively *ping* and *pong*.

An atomic block B (or simply block) is composed of three kinds of statements:

- guard statements G determine when the block is *enabled*, i.e. has a chance to be executed, and may non-deterministically select values for some of the variables;
- execution statements E define the actual behavior of the block;
- a $\texttt{goto}\ l$ statement indicates the next block to be executed (or none, if l is the reserved $\texttt{Done}$).

We consider $S ::= G \mid E$ the set of statements (guards or execution statements).

3.2 Semantics of Guarded PlusCal

A *state* is a pair $\langle M, C \rangle$ where M is a mapping from variables to values and C is a mapping from channels to sequences of values. The execution of a statement on a state produces a new state and the execution of an atomic block results in a *final state* which also indicates the label of the next block to execute. $\mathit{State} = \mathit{State}_\bot \uplus \mathit{State}_\top$ denotes the indexed family of all the states:

$$\mathit{State}_\bot = (\mathit{Var} \rightharpoonup \mathit{Value}) \times (\mathit{Var} \rightharpoonup \mathit{Value}^*)$$
$$\mathit{State}_\top = \mathit{State}_\bot \times \mathit{Label}$$

$\mathit{State}_\top$ represents execution states reached after a $\texttt{goto}$ terminal statement, while $\mathit{State}_\bot$ represents execution states obtained when executing the statements inside an atomic block. For any given $\sigma \in \mathit{State}_\bot$, we denote M_σ and C_σ its first and second projections respectively. $M(r \leftarrow v)$ denotes the mapping obtained by replacing the value of variable r with v in M, and $C(c \leftarrow vs)$ denotes the mapping obtained by replacing the sequence of values of channel c with vs in C.

To represent the actions performed by a process that are observable by other processes we consider the set *Trace* of sequences of process events of the form $\texttt{out}\ v$ (process output) or $c!v$ (channel sending).

[2] We omit in the syntax the process name.

The symbol ε denotes the empty trace, $*$ the concatenation of two traces and $\leq$ the prefix relation on traces.

We follow the approach proposed in [CWWC24], and define the semantics of Guarded PlusCal by two functions $[\![\cdot]\!]$ and $[\![\cdot]\!]^{\perp}$ describing respectively all the possible non-erroneous executions and all executions leading to an error. The semantics of atomic blocks (and threads) in case of non-erroneous executions is a set of triples $\langle \sigma, \mathfrak{t}, \sigma' \rangle$ where $\sigma \in \mathit{State}_{\perp}$ is the initial state, $\mathfrak{t} \in \mathit{Trace}$ is the trace of events produced by the statements in the block and $\sigma' \in \mathit{State}_{\top}$ is the final state. For statements, the semantics is also a set of triples $\langle \sigma, \mathfrak{t}, \sigma' \rangle$ but the state $\sigma' \in \mathit{State}_{\perp}$ is not final.

We present below the semantics of the Guarded PlusCal constructs used in the rest of the paper; the semantics of all the constructs is given in the appendix.

Semantics of (non-temporal) expressions. The expression language of TLA$^+$ is very rich and expressive, but not all expressions have a defined meaning. For example, the expression $1 \cup \text{TRUE}$ is a well-formed expression of TLA$^+$, but its meaning is not specified, and it is rejected by the model checkers TLC and Apalache.

We introduce the two reduction predicates $M \vdash e \Downarrow v$ and $M \vdash e \lightning$ that respectively describe when an expression is meaningful (i.e. can be reduced to a normal form v, given a memory M) and when an expression is meaningless (when e admits no normal form in a given memory M). The set *Value* of normal forms consists of numerals, strings, booleans, sets, sequences, tuples, records, and functions. Note that some expressions may not always be meaningful depending on the memory in which they are reduced: for example the TLA$^+$ expression $x \cup \{\text{TRUE}\}$ evaluates to $\{\text{TRUE}\}$ if $M(x) = \{\text{TRUE}\}$ but does not evaluate to a value if $M(x) = 0$.

Semantics of guards. For an `await` statement, if the corresponding expression evaluates to `TRUE`, the execution continues with no changes to the state and no impact on the trace. When the expression evaluation fails, the execution is aborted. For a `receive` statement, the first value of the channel (mapping) is removed and assigned to the variable r. Such a statement never fails. For a `with` statement, the result of evaluating the expression (or one of the elements if the result is a set) is assigned to the variable in the memory. When the expression evaluation fails, the execution is aborted.

$$
\begin{aligned}
[\![\texttt{await}\ e]\!] &= \{\langle \sigma, \varepsilon, \sigma \rangle \mid \sigma \in \mathit{State}_{\perp} \wedge M_{\sigma} \vdash e \Downarrow \texttt{TRUE}\} \\
[\![\texttt{receive}(c, r)]\!] &= \{\langle \sigma, \varepsilon, \langle M_{\sigma}(r \leftarrow v), C_{\sigma}(c \leftarrow vs) \rangle \rangle \mid \sigma \in \mathit{State}_{\perp} \wedge C_{\sigma}(c) = v \cdot vs\} \\
[\![\texttt{with}\ x = e]\!] &= \{\langle \sigma, \varepsilon, \langle M_{\sigma}(x \leftarrow v), C_{\sigma} \rangle \rangle \mid \sigma \in \mathit{State}_{\perp} \wedge M_{\sigma} \vdash e \Downarrow v\} \\
[\![\texttt{with}\ x \in e]\!] &= \{\langle \sigma, \varepsilon, \langle M_{\sigma}(x \leftarrow v), C_{\sigma} \rangle \rangle \mid \sigma \in \mathit{State}_{\perp} \wedge v \in S \wedge M_{\sigma} \vdash e \Downarrow S\} \\
[\![\texttt{await}\ e]\!]^{\perp} &= \{\langle \sigma, \varepsilon \rangle \mid \sigma \in \mathit{State}_{\perp} \wedge M_{\sigma} \vdash e \lightning\} \\
[\![\texttt{receive}(c, r)]\!]^{\perp} &= \emptyset \\
[\![\texttt{with}\ x = e]\!]^{\perp} &= \{\langle \sigma, \varepsilon \rangle \mid \sigma \in \mathit{State}_{\perp} \wedge M_{\sigma} \vdash e \lightning\} \\
[\![\texttt{with}\ x \in e]\!]^{\perp} &= \{\langle \sigma, \varepsilon \rangle \mid \sigma \in \mathit{State}_{\perp} \wedge M_{\sigma} \vdash e \lightning\}
\end{aligned}
$$

Note that we have a blocking semantics for all the guards: there is no transition when the expression in `await` evaluates to anything else than `TRUE`, when the channel in `receive` is empty, and when the expression in `with` evaluates to an empty set.

Semantics of execution statements. An assignment $r := e$ updates the value of the variable r to the value of the expression e and produces no trace event. If the evaluation of e fails so does the assignment.

$$[\![r := e]\!] = \{\langle \sigma, \varepsilon, \langle M_\sigma(r \leftarrow v), C_\sigma\rangle\rangle \mid \sigma \in State_\perp \wedge M_\sigma \vdash e \Downarrow v\}$$
$$[\![r := e]\!]^\perp = \{\langle \sigma, \varepsilon\rangle \mid \sigma \in State_\perp \wedge M_\sigma \vdash e \not\downarrow\}$$

An `assert` statement has no effect on the state and does not produce any trace event if the expression it contains evaluates to `TRUE`, otherwise the execution fails. The statements `print`, `send` and `skip` behave as usually. In particular, successful executions of `print` and `send` produce trace events. The formal semantics of all statements is given in the appendix.

Semantics of atomic blocks. Reducing an atomic block simply amounts to checking if it is enabled (i.e. if all its guards are satisfied), and then executing all the statements following the guards as well as the terminal `goto` statement. Given the (forward) composition defined by

$$R_1 \circ_2 R_2 = \{\langle s_1, \mathfrak{t}_1 * \mathfrak{t}_2, s_3\rangle \mid \langle s_1, \mathfrak{t}_1, s_2\rangle \in R_1 \wedge \langle s_2, \mathfrak{t}_2, s_3\rangle \in R_2\},$$

the semantics of a block is a set of triples of the form $State_\perp \times Trace \times State_\top$ defined inductively by:

$$[\![\texttt{goto}\ l]\!] = \{\langle \sigma, \varepsilon, \langle \sigma, l\rangle\rangle \mid \sigma \in State_\perp\}$$
$$[\![S\,; B]\!] = [\![S]\!] \circ_2 [\![B]\!]$$

An atomic block aborts whenever one of its guards or statements aborts (`goto` never aborts).

Semantics of threads and processes. We define the semantics of threads as a set of triples $State_\top \times Trace \times State_\top$, where the label in the first state indicates which block is allowed to be reduced, by

$$[\![T]\!] = \{\langle\langle M, C, l\rangle, \mathfrak{t}, \sigma'\rangle \mid \sigma' \in State_\top \wedge \langle\langle M, C\rangle, \mathfrak{t}, \sigma'\rangle \in \bigcup_{B \in T(l)} [\![B]\!]\}$$
$$[\![T]\!]^\perp = \{\langle\langle M, C, l\rangle, \mathfrak{t}\rangle \mid \langle\langle M, C\rangle, \mathfrak{t}\rangle \in \bigcup_{B \in T(l)} [\![B]\!]^\perp\}$$

where $T(l) = \{B_i \mid 1 \le i \le n \wedge l : \texttt{either}\ B_1\ \texttt{or}\ \ldots\ \texttt{or}\ B_n \in T\}$ is the collection of blocks at label l in the thread T. The semantics of threads is merely defined as the set of all the possible choices between the enabled blocks at the given label l in the thread (in their respective memory), effectively modelling the angelic non-determinism exhibited by an `either` statement in PlusCal.

The semantics of processes is then defined as the iteration of the composition of the semantics of threads (in any order), starting from the initial configuration where the memory is initialized to the initial value of local variables and each thread starts in its first block.

4 Towards compiling Guarded PlusCal into executable code

In this section, we describe the first pass of our compiler, from Guarded PlusCal to a restricted language better suitable for compilation into executable code. The other passes have also been implemented but not yet verified.

4.1 General compiler correctness

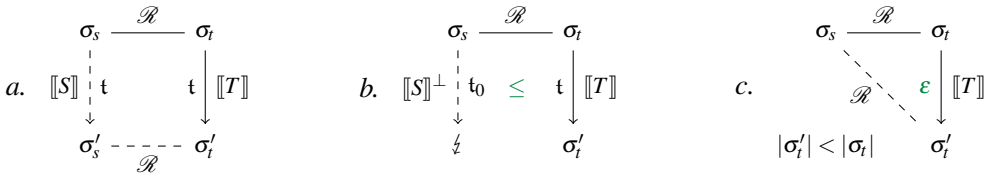
We start by recalling some standard techniques used to prove compiler correctness, which we will then apply in the following subsection. In the following, we consider two arbitrary languages $\mathscr{L}_s$ (the source language) and $\mathscr{L}_t$ (the target language) whose semantics [CWWC24] are defined by the functions $[\![\cdot]\!]$ and $[\![\cdot]\!]^\perp$ exhibiting all the possible (non-)erroneous executions in terms of tuples $(\sigma, \mathfrak{t}, \sigma')$, respectively $(\sigma, \mathfrak{t})$, with σ, σ' language specific configurations and $\mathfrak{t}$ observational traces. In the case of Guarded PlusCal, the configurations are the states of the program and the traces are sequences of events that can be observed, resulting from `print` and `send` statements. We also assume that the semantics of $\mathscr{L}_s$ and $\mathscr{L}_t$ are related by a (matching) relation $\mathscr{R}$ between configurations of the two languages and that the target language $\mathscr{L}_t$ has a well-founded measure $|\cdot|$ on its configurations.

Definition 4.1 (Behavior refinement [CWWC24])**.** *Given two programs $S \in \mathscr{L}_s$ and $T \in \mathscr{L}_t$, T refines the behavior of S, denoted $[\![T]\!] \sqsubseteq_{\mathscr{R}} [\![S]\!]$, if for all configurations σ_s of $\mathscr{L}_s$, σ_t, σ_t' of $\mathscr{L}_t$ and trace $\mathfrak{t}$ such that $(\sigma_s, \sigma_t) \in \mathscr{R}$ and $(\sigma_t, \mathfrak{t}, \sigma_t') \in [\![T]\!]$, either*

a. there exists a configuration σ_s' of $\mathscr{L}_s$ such that $(\sigma_s', \sigma_t') \in \mathscr{R}$ and $(\sigma_s, \mathfrak{t}, \sigma_s') \in [\![S]\!]$, or

b. there exists a trace $\mathfrak{t}_0 \leq \mathfrak{t}$ such that $(\sigma_s, \mathfrak{t}_0) \in [\![S]\!]^\perp$, or

c. $(\sigma_s, \sigma_t') \in \mathscr{R}$ and $\mathfrak{t} = \varepsilon$ and $|\sigma_t'| < |\sigma_t|$.



Behavior refinement states that every (non-erroneous) behavior that T can exhibit is either a behavior that can also be exhibited by S, or corresponds to a failure of S (perhaps for a shorter trace: T could have performed some additional logging actions), or is an internal action that decreases the measure (and therefore T cannot indefinitely perform such actions). Therefore, any execution of T that does not fail corresponds to a successful execution of S or to an execution of S that fails at some point. Of course, the second case cannot happen if S has been proved to be failure-free. Observe that nothing is required if T fails and therefore the definition does not ensure completeness of the compiler.

Theorem 4.2 (Refinement of sequential composition)**.** *If both $[\![S]\!] \sqsubseteq_{\mathscr{R}} [\![T]\!]$ and $[\![U]\!] \sqsubseteq_{\mathscr{R}} [\![V]\!]$ hold, then $[\![S\,;U]\!] \sqsubseteq_{\mathscr{R}} [\![T\,;V]\!]$ holds as well.*

A compiler $\mathscr{C} : \mathscr{L}_s \rightharpoonup \mathscr{L}_t$, i.e. a partial function from $\mathscr{L}_s$ to $\mathscr{L}_t$, is said to be partially correct if every program we obtain as output of $\mathscr{C}$ refines the behavior of the original program:

Definition 4.3 (Compiler partial correctness)**.** *$\mathscr{C}$ preserves the semantics of $\mathscr{L}_s$ iff for all program $P \in \mathscr{L}_s$ such that $\mathscr{C}(P)$ is defined, $[\![\mathscr{C}(P)]\!] \sqsubseteq_{\mathscr{R}} [\![P]\!]$.*

Proving full compiler partial correctness is non-trivial, but is more easily tackled by considering each pass as a separate compiler. Each pass can then be proven partially correct separately, and all the correctness results can be tied back together by simple composition of partial functions and semantics.

Theorem 4.4 (Composition of partially correct compilers)**.** *Given two compilers $\mathscr{C}_1 : \mathscr{L}_1 \rightharpoonup \mathscr{L}_2$ and $\mathscr{C}_2 : \mathscr{L}_2 \rightharpoonup \mathscr{L}_3$ that respectively preserve the semantics of $\mathscr{L}_1$ and $\mathscr{L}_2$, their functional composition $\mathscr{C}_2 \circ \mathscr{C}_1 : \mathscr{L}_1 \rightharpoonup \mathscr{L}_3$ also preserves the semantics of $\mathscr{L}_1$.*

4.2 From Guarded PlusCal to Network PlusCal

Compilation of Distributed PlusCal specifications into executable programs is a complex task essentially because of the atomicity of block execution combined with the blocking statements `await` and `receive`. A naive compilation of `receive`$(mailbox_p, x)$ into an executable language featuring asynchronous channels (e.g. into Go's `x := <-mailbox`$_p$) can lead to behaviors that are not consistent with the semantics of Distributed PlusCal.

Consider the block `receive`$(mailbox_p, x)$; `await` $x > 50$; `goto Done`. When compiled naively by simply replacing the first statement with `x := <-mailbox`$_p$, the corresponding program would receive a value from a channel `mailbox`$_p$, without actually knowing if this value is greater than 50. Note that in Go it is not possible to peek at the channel's content without actually receiving the message. To preserve the intended semantics of PlusCal, we should roll back and replace the received value back as the first value in the channel, which is not possible in Go.

We address this problem by performing a first compilation pass $\mathscr{C}_{G\to N}$ that essentially replaces all `receive` guards with `await` statements. We call Network PlusCal the restriction of Guarded PlusCal where all `receive` guards have been removed. Since every process p receives only from a single mailbox, we can use a new (fresh) variable $inbox_p$ in the local process state which handles the temporary buffering of a received message, and use it to encode the `receive` behavior. Continuing the previous example, we replace the first statements of the block by `await` $Len(inbox_p) > 0 ; x := Head(inbox_p) ; inbox_p := Tail(inbox_p) ;$ `await` $x > 50$. However, this introduces an assignment preceding an `await` guard, and the statements must be reordered to obtain `await` $Len(inbox_p) > 0 ;$ `await` $Head(inbox_p) > 50 ; x := Head(inbox_p) ; inbox_p := Tail(inbox_p)$. Additionally, we introduce a separate thread to handle the transfer of messages from the mailbox to the local variable $inbox_p$. These transformations are performed by rewrite rules, as defined below.

The compiler $\mathscr{C}_{G\to N}$. The goal of this compiler pass is to get rid of any potential blocking that may be incurred by `receive` guards by using an explicitly handled sequence $inbox_p$; channel reception then becomes indexing into $inbox_p$ if that sequence is nonempty.

The compilation of `receive` statements is done by normalizing w.r.t. the one rule rewrite system $\mathscr{E}$:

$$p \star V^* \star T_1^* \{ S_1^* ; \texttt{receive}(mailbox_p, x) ; S_2^* ; \texttt{goto } l ; \} T_2^*$$
$$\Longrightarrow_{\mathscr{E}}$$
$$p \star V^*, inbox_p = \langle\rangle \star T_1^* \{ S_1^* ; \texttt{await } Len(inbox_p) > 0 ; x := Head(inbox_p) ; inbox_p := Tail(inbox_p) ; S_2^* ; \texttt{goto } l ; \} T_2^*$$

where V^*, S^* and T^* stand for arbitrary sequences of variables, statements and threads, respectively. The standard operators *Len*, *Head* and *Tail* on TLA$^+$ sequences denote the number of elements in the sequence, the first element of the sequence (provided it is nonempty) and the sequence without its first element, and $\langle\rangle$ denotes the empty sequence.

As explained above, the compilation of `receive` guards introduces new assignments that need to be reordered in order to produce a valid Network PlusCal block. This reordering is done by replacing the variables involved in the newly introduced assignments with the corresponding expressions. Since these expressions only depend on the freshly generated variables $inbox_p$, no special care needs to be taken in the replacement process to avoid possible variable captures. This is done by normalizing w.r.t. the rewrite

system $\mathscr{S}$:

$$\begin{aligned}
S_1^*\,;x \coloneqq e_1\,;\texttt{await}\ e_2\,;S_2^*\,;\texttt{goto}\ l\,; &\quad\Longrightarrow_{\mathscr{S}}\quad S_1^*\,;\texttt{await}\ e_2[e_1/x]\,;x \coloneqq e_1\,;S_2^*\,;\texttt{goto}\ l\,;\\
S_1^*\,;x \coloneqq e_1\,;\texttt{with}\ y = e_2\,;S_2^*\,;\texttt{goto}\ l\,; &\quad\Longrightarrow_{\mathscr{S}}\quad S_1^*\,;\texttt{with}\ y = e_2[e_1/x]\,;x \coloneqq e_1\,;S_2^*\,;\texttt{goto}\ l\,;\\
S_1^*\,;x \coloneqq e_1\,;\texttt{with}\ y \in e_2\,;S_2^*\,;\texttt{goto}\ l\,; &\quad\Longrightarrow_{\mathscr{S}}\quad S_1^*\,;\texttt{with}\ y \in e_2[e_1/x]\,;x \coloneqq e_1\,;S_2^*\,;\texttt{goto}\ l\,;
\end{aligned}$$

Because we are replacing `receive` guards with indexing into a new local variable $inbox_p$, we also need to insert a separate thread that handles the transfer of messages from $mailbox_p$ to that variable. This is accomplished by applying once the following rewrite rule on all processes that have an associated mailbox:

$$p \star V^* \star T^* \Longrightarrow_{\mathscr{M}} p \star V^*, tmp = v \star T^*\{rx_p : \texttt{receive}(mailbox_p, tmp)\,; inbox_p \coloneqq Append(inbox_p, tmp)\,; \texttt{goto}\ rx_p\,;\}$$

where *tmp* is a fresh process-local variable that is initialized to some value v of the same type as the type contained in the mailbox, and *Append* is the operator from the TLA$^+$ standard library that denotes insertion of a value at the end of a sequence. Note that the choice of the initial value v does not matter because *tmp* will always be overwritten before its first use.

We then define the compilation process by $\mathscr{C}_{G\to N} = \downarrow_{\mathscr{E}} \circ \downarrow_{\mathscr{S}} \circ \Longrightarrow_{\mathscr{M}}$ with $\downarrow_R$ denoting the reduction to normal form w.r.t. a rewrite system R. The result of applying our compiler $\mathscr{C}_{G\to N}$ to the example of 2.1 is presented in Listing 4.1.

```
--algorithm PingPong {
    fifos ping, pong;

    process (Ping = 0)
        variables
            tmp1 = "", rx_inbox_Ping = "",
                       inbox_Ping = <<>>;
    {
    rcv1:   await Len(inbox_Ping) > 0;
            await Head(inbox_Ping) = "Ping";
            tmp1 := Head(inbox_Ping);
            inbox_Ping := Tail(inbox_Ping);
            goto pong;
    pong:   send(pong, "Pong");
            goto rcv1;
    }
    {
    rx1:    receive(ping, rx_inbox_Ping);
            inbox_Ping :=
                Append(inbox_Ping, rx_inbox_Ping);
            goto rx;
    }

    process (Pong = 1)
        variables
            tmp2 = "", rx_inbox_Pong = "",
                       inbox_Pong = <<>>;
    {
    ping:   send(ping, "Ping");
            goto rcv2;
    rcv2:   await Len(inbox_Pong) > 0;
            await Head(inbox_Pong) = "Pong";
            tmp2 := Head(inbox_Pong);
            inbox_Pong := Tail(inbox_Pong);
            goto ping;
    }
    {
    rx2:    receive(pong, rx_inbox_Pong);
            inbox_Pong :=
                Append(inbox_Pong, rx_inbox_Pong);
            goto rx;
    }
}
```

Listing 4.1: Ping-pong algorithm in Network PlusCal

On the partial correctness of $\mathscr{C}_{G\to N}$. The different languages involved in the compilation (Distributed PlusCal, Guarded PlusCal, Network PlusCal) are specified as types in the Lean proof assistant, and the rewrite rules defining the passes are implemented as Lean functions. Because in Lean, functions must terminate and respect their types, each transformation step is terminating, and the resulting normal forms contain only well-formed Network PlusCal blocks.

Theorem 4.5 (Termination). *$\mathscr{C}_{G\to N}$ is terminating and the produced normal forms are valid Network PlusCal processes.*

Concerning the correctness of this compilation pass, since the semantics of processes and threads does not change between Guarded PlusCal and Network PlusCal, it suffices to show that the compilation of each atomic block is correct. To establish this property, we first define the relation $\langle M_s, C_s\rangle \sim_{mailbox_p} \langle M_t, C_t\rangle$, which connects configurations of Guarded PlusCal and Network PlusCal in the presence of a mailbox:

$$\langle M_s, C_s\rangle \sim_{mailbox_p} \langle M_t, C_t\rangle \iff \bigwedge \begin{cases} \forall x \neq inbox_p.\ M_s(x) = M_t(x), \\ \forall c \neq mailbox_p.\ C_s(c) = C_t(c), \\ C_s(mailbox_p) = M_t(inbox_p) +\!\!+ C_t(mailbox_p) \end{cases}$$

where $+\!\!+$ denotes sequence concatenation in TLA$^+$. In words, the values of variables other than the newly added variable $inbox_p$ and of channels other than $mailbox_p$ must match in the configurations of the source and target programs. Moreover, the content of $mailbox_p$ in the source configuration must correspond to the concatenation of the sequences contained in $inbox_p$ and in $mailbox_p$ in the target configuration, since some messages from the mailbox may already have been transferred to the local variable $inbox_p$. In particular, in the absence of a mailbox, the source and target configurations must match exactly.

The semantics of all languages involved in the compilation process have been defined in Lean and the partial correctness of each transformation step has been mechanically established. The partial correctness of the compilation pass $\mathscr{C}_{G\to N}$ is then obtained by composition of these results.

Theorem 4.6 (Partial correctness of $\mathscr{C}_{G\to N}$). *For any Guarded PlusCal process P, we have that $[\![\mathscr{C}_{G\to N}(P)]\!] \sqsubseteq_{\sim_{mailbox_p}} [\![P]\!]$.*

5 Conclusion

We have formally defined the semantics of the Distributed PlusCal specification language, an extension of PlusCal featuring processes, threads, and message-passing communications, in Lean 4, and we have started the development of a compiler into an executable language.

The first pass of the compiler, which transforms programs written in the Guarded PlusCal fragment of Distributed PlusCal into the Network PlusCal fragment by separating `await` and `receive` statements through the introduction of a helper thread and reordering statements if necessary, has been specified using rewrite rules and implemented in Lean 4. The rigorous description of Guarded PlusCal and Network PlusCal, using the type system of Lean 4, allows us to automatically establish the termination of the translation process and to guarantee that the resulting programs have the expected shape. We have also used the Lean 4 proof assistant to mechanically verify the correctness of the translation process.

The next step concerns the translation into Go programs. The syntax of Go has already been defined in Lean 4, and the translation from Network PlusCal into Go has been implemented. However, we will also need to define a formal semantics of Go in Lean 4 in order to state the correctness of the compilation. The preliminary compilation pass from Guarded PlusCal into Network PlusCal significantly simplifies the compilation into executable code since each block begins with a sequence of (side effect free) guards that must all be satisfied before changing the process-local environment. This allows us to avoid cancellation mechanisms (such as rollbacks in transactions) when compiling a block, because the enabledness of a block is completely determined by its guards. However, we still need to introduce synchronization mechanisms (such as locks) in order to prevent *data races* between multiple concurrently executing blocks. As before, we obtain for free the termination of the pass as well as the well-formedness of the resulting Go programs, but the correctness of the translation still needs to be established.

References

[CDL+12] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts & Hernán Vanzetto (2012): *TLA+ Proofs*. In Dimitra Giannakopoulou & Dominique Méry, editors: *18th Intl. Symp. Formal Methods (FM 2012)*, *LNCS* 7436, Springer, Paris, France, pp. 147–154.

[CM23] Horatiu Cirstea & Stephan Merz (2023): *Extending PlusCal for Modeling Distributed Algorithms*. In Paula Herber & Anton Wijs, editors: *iFM 2023 - 18th International Conference, iFM 2023, Leiden, The Netherlands, November 13-15, 2023, Proceedings*, *Lecture Notes in Computer Science* 14300, Springer, pp. 321–340, doi:10.1007/978-3-031-47705-8_17.

[CWWC24] Zhang Cheng, Jiyang Wu, Di Wang & Qinxiang Cao (2024): *Denotation-based Compositional Compiler Verification*. arXiv:2404.17297.

[HB24] Marian Hristov & Annette Bieniusa (2024): *Erla+: Translating TLA+ Models into Executable Actor-Based Implementations*. In: *Proceedings of the 23rd ACM SIGPLAN International Workshop on Erlang*, Erlang 2024, Association for Computing Machinery, New York, NY, USA, p. 13–23, doi:10.1145/3677995.3678190. Available at https://doi.org/10.1145/3677995.3678190.

[HHC+23] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do & Ivan Beschastnikh (2023): *Compiling Distributed System Models with PGo*. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, Association for Computing Machinery, New York, NY, USA, p. 159–175, doi:10.1145/3575693.3575695. Available at https://doi.org/10.1145/3575693.3575695.

[KKM22] Igor Konnov, Markus Kuppe & Stephan Merz (2022): *Specification and Verification with the TLA+ Trifecta: TLC, Apalache, and TLAPS*. In Tiziana Margaria & Bernhard Steffen, editors: *11th Intl. Symp. Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2022)*, *Lecture Notes in Computer Science* 13701, Springer, Rhodes, Greece, pp. 88–105.

[KKT19] Igor Konnov, Jure Kukovec & Thanh-Hai Tran (2019): *TLA+ model checking made symbolic*. *Proc. ACM Program. Lang.* 3(OOPSLA), doi:10.1145/3360549. Available at https://doi.org/10.1145/3360549.

[Lam02] Leslie Lamport (2002): *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, USA.

[Lam09] Leslie Lamport (2009): *The PlusCal Algorithm Language*. In M. Leucker & C. Morgan, editors: *6th Intl. Coll. Theor. Asp. Comp. (ICTAC 2009)*, *LNCS* 5684, Springer, Kuala Lumpur, Malaysia, pp. 36–60.

[Ler09] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Communications of the ACM* 52(7), pp. 107–115. Available at http://xavierleroy.org/publi/compcert-CACM.pdf.

[LKM+] Leslie Lamport, Markus A. Kuppe, Stephan Merz, Andrew Helwer, William Schultz, Jeff Hemphill, Mariusz Ryndzionek, Igor Konnov, Thanh Hai Tran, Josef Widder, Jim Gray, Murat Demirbas, Guanzhou Hu, Giuliano Losa, Ron Pressler, Younes Akhouayri, Luming Dong, Zhi Niu, Lim Ngian Xin Terry, Gaurav Gandhi, Isaac DeFrain, Martin Harrison, Santhosh Raju, Cherry G. Mathew, Fransisca Andriani & Ludovic Yvoz: *TLA+ Examples*. https://github.com/tlaplus/examples/.

[LSL17] Yanhong A. Liu, Scott D. Stoller & Bo Lin (2017): *From Clarity to Efficiency for Distributed Algorithms*. *ACM Transactions on Programming Languages and Systems* 39(3), pp. 1–41.

[MU21] Leonardo de Moura & Sebastian Ullrich (2021): *The Lean 4 Theorem Prover and Programming Language*. In: *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, Springer-Verlag, Berlin, Heidelberg, p. 625–635, doi:10.1007/978-3-030-79876-5_37. Available at https://doi.org/10.1007/978-3-030-79876-5_37.

[WWP+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst & Thomas Anderson (2015): *Verdi: a framework for implementing and formally verifying distributed systems*. *SIGPLAN Not.* 50(6), p. 357–368, doi:10.1145/2813885.2737958. Available at https://doi.org/10.1145/2813885.2737958.

[YML99] Yuan Yu, Panagiotis Manolios & Leslie Lamport (1999): *Model Checking TLA+ Specifications*. In Laurence Pierre & Thomas Kropf, editors: *Correct Hardware Design and Verification Methods*, Springer, Bad Herrenalb, Germany, pp. 54–66.

[ZBF20] Peter Zeller, Annette Bieniusa & Carla Ferreira (2020): *Teaching practical realistic verification of distributed algorithms in Erlang with TLA+*. In Annette Bieniusa & Viktória Fördós, editors: *Erlang Workshop*, ACM, pp. 14–23.

Appendix

A Normal forms of TLA$^+$ expressions

We define the set *Value* of all normal forms v by:

v	$::=$	n	Number literal
	$\mid$	$"s"$	String literal
	$\mid$	TRUE $\mid$ FALSE	Boolean literal
	$\mid$	$\{v_1, \ldots, v_n\}$	Set literal
	$\mid$	$\langle v_1, \ldots, v_n \rangle$	Sequence/tuple literal
	$\mid$	$[x_1 : v_1, \ldots, x_n : v_n]$	Record literal
	$\mid$	$v_1 :> v_2$ @@ $\ldots$ @@ $v_{n-1} :> v_n$	Function literal
	$\mid$	LAMBDA $x_1 \ldots x_n.\ e$	Operator literal

B Definition of the semantics of Guarded PlusCal

Complete semantics of execution statements A skip statement has no effect on the state and does not produce any trace event. A print statement produces an output event containing the value of the expression it contains. An assert statement has no effect on the state and does not produce any trace event if the expression it contains evaluates to TRUE, otherwise it aborts the execution. A send statement appends the value of the expression it contains to the sequence of values of the channel it sends to, and produces a sending event. An assignment statement $r := e$ updates the value of the variable r to the value of the expression e and produces no trace event. A skip could not lead to a failure, all the other statements lead to a failure if the corresponding expression does so and in this case, no trace event is produced.

$$
\begin{aligned}
[\![\texttt{skip}]\!] &= \{\langle \sigma, \varepsilon, \sigma \rangle \mid \sigma \in \mathit{State}_\perp\} \\
[\![\texttt{print}\ e]\!] &= \{\langle \sigma, \texttt{out}\ v, \sigma \rangle \mid \sigma \in \mathit{State}_\perp \wedge M_\sigma \vdash e \Downarrow v\} \\
[\![\texttt{assert}\ e]\!] &= \{\langle \sigma, \varepsilon, \sigma \rangle \mid \sigma \in \mathit{State}_\perp \wedge M_\sigma \vdash e \Downarrow \texttt{TRUE}\} \\
[\![\texttt{send}(c, e)]\!] &= \{\langle \sigma, c!v, \langle M_\sigma, C_\sigma(c \leftarrow vs \cdot v) \rangle \rangle \mid \sigma \in \mathit{State}_\perp \wedge C_\sigma(c) = vs \wedge M_\sigma \vdash e \Downarrow v\} \\
[\![r := e]\!] &= \{\langle \sigma, \varepsilon, \langle M_\sigma(r \leftarrow v), C_\sigma \rangle \rangle \mid \sigma \in \mathit{State}_\perp \wedge M_\sigma \vdash e \Downarrow v\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{skip}]\!]^\perp &= \emptyset \\
[\![\texttt{print}\ e]\!]^\perp &= \{\langle \sigma, \varepsilon \rangle \mid \sigma \in \mathit{State}_\perp \wedge M_\sigma \vdash e \not\downarrow\} \\
[\![\texttt{assert}\ e]\!]^\perp &= \{\langle \sigma, \varepsilon \rangle \mid \sigma \in \mathit{State}_\perp \wedge (M_\sigma \vdash e \not\downarrow \vee M_\sigma \vdash e \Downarrow \texttt{FALSE})\} \\
[\![\texttt{send}(c, e)]\!]^\perp &= \{\langle \sigma, \varepsilon \rangle \mid \sigma \in \mathit{State}_\perp \wedge M_\sigma \vdash e \not\downarrow\} \\
[\![r := e]\!]^\perp &= \{\langle \sigma, \varepsilon \rangle \mid \sigma \in \mathit{State}_\perp \wedge M_\sigma \vdash e \not\downarrow\}
\end{aligned}
$$

Complete semantics of atomic blocks Let the (forward) composition of semantics sets $(\circ_2) : (\mathit{State} \times \mathit{Trace} \times \mathit{State}) \times (\mathit{State} \times \mathit{Trace} \times \mathit{State}) \to \mathit{State} \times \mathit{Trace} \times \mathit{State}$ defined by

$$
R_1 \circ_2 R_2 = \{\langle s_1, \mathrm{t}_1 * \mathrm{t}_2, s_3 \rangle \mid \langle s_1, \mathrm{t}_1, s_2 \rangle \in R_1 \wedge \langle s_2, \mathrm{t}_2, s_3 \rangle \in R_2\},
$$

the semantics of an atomic block is a set of triples of the form $State_\bot \times Trace \times State_\top$ defined inductively by:

$$\begin{aligned}[\![\texttt{goto}\ l]\!] &= \{\langle \sigma, \varepsilon, \langle \sigma, l\rangle\rangle \mid \sigma \in State_\bot\} \\ [\![S;B]\!] &= [\![S]\!] \circ_2 [\![B]\!]\end{aligned}$$

Let the (forward) composition $(\circ_1) : (State \times Trace \times State) \times (State \times Trace) \to State \times Trace$ defined by

$$R \circ_1 X = \{\langle s_1, t_1 * t_2\rangle \mid (s_1, t_1, s_2) \in R \wedge (s_2, t_2) \in X\},$$

the aborting semantics of an atomic block is then defined inductively by:

$$\begin{aligned}[\![\texttt{goto}\ l]\!]^\bot &= \emptyset \\ [\![S;B]\!]^\bot &= [\![S]\!]^\bot \cup ([\![S]\!] \circ_1 [\![B]\!]^\bot)\end{aligned}$$

Reducing an atomic block simply amounts to checking if it is enabled (i.e. if all its guards are satisfied), and then executing all the statements following the guards as well as the terminal `goto` statement. An atomic block aborts whenever one of its guards or statements aborts (`goto` never aborts).

On Merging Constrained Rewrite Rules of Induction Hypotheses in Constrained Rewriting Induction

Naoki Nishida
Nagoya University, Nagoya Japan
nishida@i.nagoya-u.ac.jp

Nozomi Taira*
Nagoya University, Nagoya, Japan

In proving constrained equations by rewriting induction (RI, for short), the Expand rule which is one of the main inference rules of RI makes an orientable constrained equation a constrained rewrite rule, called an IH rewrite rule, which can be used as an induction hypothesis. Such an IH rewrite rule may be applied to constrained equations in order to simplify them. The broader the scope of the application of IH rewrite rules, the greater the possibilities for proving constrained equations. In this paper, we modify the Expand rule of RI in order to make the scope of the application of the IH rewrite rule broader. To be more precise, in orienting a constrained equation as an IH rewrite rule, if there already exists an IH rewrite rule having the same left- and right-hand sides, then we disjunct the constraint of the orienting equation with the constraint of the existing IH rewrite rule; otherwise, the orienting equation is added to the set of IH rewrite rules as a new IH rewrite rule. We show that this modification does not lose the proof power of RI.

1 Introduction

Recently, approaches to program verification by means of logically constrained term rewrite systems (LCTRSs, for short) [13] are well investigated [6, 19, 4, 16, 7, 8, 11, 9, 10, 12, 15]. LCTRSs are extensions of *term rewrite systems* (TRSs, for short) by allowing rewrite rules to have guard constraints which are evaluated under equipped built-in theories. LCTRSs combine classic *term rewriting* (see, e.g., [1, 17]) with built-in data types and constraints from user-specified first-order theories, specifically those supported by modern Satisfiability Modulo Theories (SMT) solvers (cf. [2, 3]). This allows for a high expressivity that is useful for representing many programming language constructs directly, together with robust tool support, e.g., the tool Ctrl [14], for automated reasoning. For instance, equivalence checking by means of LCTRSs is useful to ensure the correctness of terminating functions (cf. [6]). Due to these features, LCTRSs are known to be useful computational models of not only functional but also imperative programs.

In previous work [6, 7, 8, 11], as a verification technique, *rewriting induction* (RI, for short) [18, 6] is used to prove equivalence of two programs by reducing equivalence to an *inductive theorem* which is a valid constrained equation of terms w.r.t. the reduction of the corresponding LCTRS. In the field of term rewriting, RI is one of the most powerful principles to prove equations to be inductive theorems of rewrite systems. RI has been extended to several kinds of rewrite systems, including LCTRSs.

RI consists of inference rules that are applied to RI states $(\mathcal{E}, \mathcal{H})$, where $\mathcal{E}$ is a finite set of (constrained) equations and $\mathcal{H}$ is a (constrained) rewrite system. The application of rewrite rules in $\mathcal{H}$ corresponds to the application of induction hypotheses to subsequent RI states. We call rules in $\mathcal{H}$ *IH rewrite rules*. An *RI proof* of a finite set $\mathcal{E}_0$ of (constrained) equations w.r.t. a rewrite system $\mathcal{R}$ is a sequence of RI states, which starts with $(\mathcal{E}_0, \emptyset)$, is obtained by the application of inference rules, and ends with

*Graduated in March 2025.

Program 1: a C program to compute summation.

```
1 int sumx(int n) {
2   if( n == 0 )
3     return 0;
4   else
5     return n + sumx(n + (n > 0 ? -1 : 1));
6 }
```

$(\emptyset, \mathcal{H}')$ for some rewrite system $\mathcal{H}'$. If we find an RI proof of $\mathcal{E}_0$ w.r.t. $\mathcal{R}$, then all constrained equations in $\mathcal{E}_0$ are proved to be inductive theorems of $\mathcal{R}$. Ctrl [14][1]—a tool to automatically prove properties of LCTRSs—is one of the known LCTRS tools for proving constrained equations to be inductive theorems of LCTRSs (cf. [6]).

In proving constrained equations to be inductive theorems of a given constrained rewrite system, constrained equations are sometimes decomposed to some constrained equations by disjuctively splitting constraints. To be more precise, we split a constrained equation $s \approx t \ [\phi_1 \vee \cdots \vee \phi_n]$ to n constrained equations $s \approx t \ [\phi_1], \ldots, s \approx t \ [\phi_n]$. RI for constrained rewrite systems has an inference rule for such splitting of constrained equations, which is a natural and useful way in proving equational claims.

In proving constrained equations by RI, the EXPAND rule which is one of the main inference rules of RI makes an orientable (constrained) equation an IH rewrite rule. To be more precise, for an RI state $(\mathcal{E} \uplus \{s \approx t \ [\phi]\}, \mathcal{H})$ of an LCTRS $\mathcal{R}$, if $\mathcal{R} \cup \mathcal{H} \cup \{s \to t \ [\phi]\}$ is terminating, then the EXPAND rule induces an RI state of the form $(\mathcal{E} \cup \mathcal{E}', \mathcal{H} \cup \{s \to t \ [\phi]\})$ for some set $\mathcal{E}'$ of constrained equations. Such an IH rewrite rule may be applied to constrained equations in subsequent RI states in order to simplify them. The broader the scope of the application of IH rewrite rules, the greater the possibilities for proving constrained equations.

Example 1.1 Let us consider Program 1 written in the C language, which is considered a SIMP$^+$ program. SIMP$^+$ [7] is an extension of a small imperative language, so-called SIMP [5], to global variables and function calls. Unlike the C language, the range of variables with type `int` is not the 32-bit integers but the integers. Given a non-negative integer n, the function `sumx` returns the summation from 0 to n, i.e., $\texttt{sumx}(n) = \sum_{i=1}^{n} i$. In addition, given a negative integer n', `sumx` return the summation from n' to 0, i.e., $\texttt{sumx}(n') = \sum_{i=n'}^{-1} i$. The program is transformed into the following LCTRS:

$$\mathcal{R}_1 = \left\{ \begin{array}{ll} \mathsf{sumx}(n) \to 0 & [n = 0 \quad\quad\quad\quad\ ] \\ \mathsf{sumx}(n) \to n + \mathsf{sumx}(n + one) & [n > 0 \wedge one = -1] \\ \mathsf{sumx}(n) \to n + \mathsf{sumx}(n + one) & [n < 0 \wedge one = 1 \quad] \end{array} \right\}$$

The above constrained equation represents the following conditional equational claim:

For any integer n, if $n \geq 0$, then $\mathsf{sumx}(n) = \frac{n(n+1)}{2}$, and otherwise (i.e., $n < 0$), $\mathsf{sumx}(n) = -\frac{n(n-1)}{2}$

To prove the correctness of `sumx`, let us try to prove the following constrained equation to be an inductive theorem of $\mathcal{R}_1$:

$$\mathsf{sumx}(n) \approx m \ \ [(n \geq 0 \wedge m = (n \times (n+1)) \,\mathsf{div}\, 2) \vee (n < 0 \wedge m = -((n \times (n-1)) \,\mathsf{div}\, 2))] \qquad (1)$$

[1] `http://cl-informatik.uibk.ac.at/software/ctrl/`

Ctrl succeeds in automatically prove the above constrained equation (1). Let us now consider to split the above equation regarding $\vee$, i.e., split to the following two constrained equations:

$$\mathsf{sumx}(n) \approx m \ \ [n \geq 0 \wedge m = (n \times (n+1))\,\mathsf{div}\,2] \tag{2}$$

$$\mathsf{sumx}(n) \approx m \ \ [n < 0 \wedge m = -((n \times (n-1))\,\mathsf{div}\,2)] \tag{3}$$

Ctrl succeeds in automatically proving the constrained equation (2), but fails to prove (3) in the automatic mode. Note that we can succeed in manually proving (3) by Ctrl. To prove (3), as for (2), let us first apply EXPAND to $(\{(3)\},\emptyset)$, obtaining the following RI state:

$$(\left\{\begin{array}{c} 0 \approx m \ [n = 0 \wedge \Psi_{m,n}] \\ n + \mathsf{sumx}(n + \mathit{one}) \approx m \ [n > 0 \wedge \mathit{one} = -1 \wedge \Psi_{m,n}] \\ n + \mathsf{sumx}(n + \mathit{one}) \approx m \ [n < 0 \wedge \mathit{one} = 1 \wedge \Psi_{m,n}] \end{array}\right\}, \{\mathsf{sumx}(n) \to m \ [\Psi_{m,n}]\})$$

where $\Psi_{m,n} = (n < 0 \wedge m = -((n \times (n-1))\,\mathsf{div}\,2))$. The first equation can be deleted because $m = 0$ is valid under the constraint $n = 0 \wedge \Psi_{m,n}$. The second equation can be deleted because the constraint is unsatisfiable. The third equation can be simplified by replacing $n + \mathit{one}$ with a fresh variable y and by conjuncting $y = n + \mathit{one}$ with the constraint. Thus, we obtain the following RI state:

$$(\{n + \mathsf{sumx}(y) \approx m \ [n < 0 \wedge \mathit{one} = 1 \wedge \Psi_{m,n} \wedge y = n + \mathit{one}]\}, \{\mathsf{sumx}(n) \to m \ [\Psi_{m,n}]\})$$

Unfortunately, in contrast to (2), we cannot further simplify the above equation via $\mathcal{R}_1$ or the IH rewrite rule $\mathsf{sumx}(n) \to m \ [\Psi_{m,n}]$. By focusing $\mathsf{sumx}(y)$ in the left-hand side, we can apply EXPAND to the equation and then all the derived equations can automatically be proved by Ctrl.

In this paper, we modify the EXPAND rule of RI for LCTRSs in order to make the scope of the application of IH rewrite rules broader. Roughly speaking, in orienting a constrained equation as an IH rewrite rule, if there already exists an IH rewrite rule having the same left- and right-hand sides, then we disjunct the constraint of the orienting equation with the constraint of the existing IH rewrite rule; otherwise, the orienting equation is added to the set of IH rewrite rules as a new IH rewrite rule. To be more precise, for an RI state $(\mathcal{E} \uplus \{s \approx t \ [\phi]\}, \mathcal{H})$ w.r.t. an LCTRS $\mathcal{R}$, if $\mathcal{R} \cup \mathcal{H} \cup \{s \to t \ [\phi]\}$ is terminating, then our new EXPAND rule induces an RI state of the form $(\mathcal{E} \cup \mathcal{E}', \mathcal{H}')$ for some set $\mathcal{E}'$ of constrained equations, where

- if $\mathcal{H}$ includes $s \to t \ [\psi]$ with $\mathcal{V}ar(\psi) = \mathcal{V}ar(\phi)$, then $\mathcal{H}' = (\mathcal{H} \setminus \{s \to t \ [\psi]\}) \cup \{s \to t \ [\psi \vee \phi]\}$, and
- otherwise, $\mathcal{H}' = \mathcal{H} \cup \{s \to t \ [\phi]\}$.

Since constraints of constrained equations are guard conditions, $s \approx t \ [\phi]$ can be considered $\phi \Rightarrow (s = t)$. Viewed in this light, having the induction hypotheses $s \approx t \ [\phi_1]$ and $s \approx t \ [\phi_2]$ is theoretically equivalent to having the induction hypothesis $s \approx t \ [\phi_1 \vee \phi_2]$. However, induction hypotheses are represented as constrained rewrite rules in the RI setting, and they are applied to constrained equations. From the viewpoint of applying constrained rewrite rules, having the induction hypothesis $s \approx t \ [\phi_1 \vee \phi_2]$ as a constrained rewrite rule is more powerful than having the induction hypotheses $s \approx t \ [\phi_1]$ and $s \approx t \ [\phi_2]$ as constrained rewrite rules. To be more precise, we have that $\to_{\{s \to t \ [\phi_1 \vee \phi_2]\}} \supsetneq \to_{\{s \to t \ [\phi_1],\ s \to t \ [\phi_2]\}}$, but the other inclusion does not hold in general. We show that the modification of EXPAND does not lose the proof power of RI.

Example 1.2 Let us consider $\mathcal{R}_1$ and constrained equations (2) and (3) in Example 1.1 again. Applying our new EXPAND rule to $(\{(2),(3)\},\emptyset)$ regarding (2), we obtain the following RI state:

$$(\left\{\begin{array}{c} 0 \approx m\ [n=0 \wedge \Phi_{m,n}] \\ n+\mathsf{sumx}(n+one) \approx m\ [n>0 \wedge one=-1 \wedge \Phi_{m,n}] \\ n+\mathsf{sumx}(n+one) \approx m\ [n<0 \wedge one=1 \wedge \Phi_{m,n}] \\ (3)\ \ \mathsf{sumx}(n) \approx m\ [\Psi_{m,n}] \end{array}\right\}, \{\mathsf{sumx}(n) \to m\ [\Phi_{m,n}]\})$$

where $\Phi_{m,n} = (n \geq 0 \wedge m = (n \times (n+1))\,\mathsf{div}\,2)$. The first to third equations are proved by the application of some RI inference rules, obtaining the following RI state:

$$(\{\mathsf{sumx}(n) \approx m\ [\Psi_{m,n}]\}, \{\mathsf{sumx}(n) \to m\ [\Phi_{m,n}]\})$$

Then, $\mathcal{R}_1 \cup \{\mathsf{sumx}(n) \to m\ [\Phi_{m,n}],\ \mathsf{sumx}(n) \to m\ [\Psi_{m,n}]\}$ is terminating, and thus we apply our new EXPAND rule to the above RI state. Unlike the original EXPAND rule, we merge $\mathsf{sumx}(n) \to m\ [\Phi_{m,n}]$ and $\mathsf{sumx}(n) \to m\ [\Psi_{m,n}]$ as a single rule, obtaining the following RI state:

$$(\left\{\begin{array}{c} 0 \approx m\ [n=0 \wedge \Psi_{m,n}] \\ n+\mathsf{sumx}(n+one) \approx m\ [n>0 \wedge one=-1 \wedge \Psi_{m,n}] \\ n+\mathsf{sumx}(n+one) \approx m\ [n<0 \wedge one=1 \wedge \Psi_{m,n}] \end{array}\right\}, \{\mathsf{sumx}(n) \to m\,[\Phi_{m,n} \vee \Psi_{m,n}]\})$$

As in Example 1.1, the first and second equations can be deleted, and the third can be simplified:

$$(\{n+\mathsf{sumx}(y) \approx m\ [n<0 \wedge one=1 \wedge \Psi_{m,n} \wedge y=n+one]\}, \{\mathsf{sumx}(n) \to m\,[\Phi_{m,n} \vee \Psi_{m,n}]\})$$

The equation above can be simplified to the following one by the IH rewrite rule in the RI state:

$$n+m' \approx m\ [n<0 \wedge one=1 \wedge \Psi_{m,n} \wedge y=n+one \wedge (\Phi_{m',y} \vee \Psi_{m',y})]$$

By conjuncting the negation of $n+m'=m$ with the constraint part, an RI inference rule (EQ-DELETE in Section 3) transforms the above equation into the following one:

$$n+m' \approx m\ [n<0 \wedge one=1 \wedge \Psi_{m,n} \wedge y=n+one \wedge (\Phi_{m',y} \vee \Psi_{m',y}) \wedge n+m' \neq m]$$

The constraint of the above equation is unsatisfiable, and thus the above equation is deleted. Therefore, using our new EXPAND *twice*, we succeed in proving both (2) and (3) to be inductive theorems of $\mathcal{R}_1$. Note that to prove (2) and (3), we used the original EXPAND *three times* in Example 1.1.

2 Preliminaries

In this section, we briefly recall LCTRSs [13, 6]. Familiarity with basic notions and notations on term rewriting [1, 17] is assumed.

To define an LCTRS [13, 6] over an $\mathcal{S}$-sorted signature Σ, we consider the following sorts, signatures, mappings, and constants: *Theory sorts* in $\mathcal{S}_{theory}$ and *term sorts* in $\mathcal{S}_{term}$ such that $\mathcal{S} = \mathcal{S}_{theory} \uplus \mathcal{S}_{term}$; a *theory signature* Σ_{theory} and a *term signature* Σ_{terms} such that $\Sigma = \Sigma_{theory} \cup \Sigma_{terms}$ and $\iota_1, \ldots, \iota_n, \iota \in \mathcal{S}_{theory}$ for any symbol $f : \iota_1 \times \cdots \times \iota_n \to \iota \in \Sigma_{theory}$; a mapping $\mathcal{I}$ that assigns to each theory sort ι a (non-empty) set $\mathcal{A}_\iota$, so-called the universe of ι (i.e., $\mathcal{I}(\iota) = \mathcal{A}_\iota$); a mapping $\mathcal{J}$, so-called an interpretation for Σ_{theory}, that assigns to each function symbol $f : \iota_1 \times \cdots \times \iota_n \Rightarrow \iota \in \Sigma_{theory}$ a function $f^{\mathcal{J}}$ in $\mathcal{I}(\iota_1) \times$

$\cdots \times \mathcal{I}(\iota_n) \to \mathcal{I}(\iota)$ (i.e., $\mathcal{J}(f) = f^{\mathcal{J}}$); a set $\mathcal{V}al_\iota \subseteq \Sigma_{theory}$ of *value-constants* $a : \iota$ for each theory sort ι such that $\mathcal{J}$ gives a bijection from $\mathcal{V}al_\iota$ to $\mathcal{I}(\iota)$ $(= \mathcal{A}_\iota)$. We denote $\bigcup_{\iota \in \mathcal{S}_{theory}} \mathcal{V}al_\iota$ by $\mathcal{V}al$. Note that $\mathcal{V}al \subseteq \Sigma_{theory}$. For readability, we may not distinguish $\mathcal{V}al_\iota$ and $\mathcal{I}(\iota)$ $(= \mathcal{A}_\iota)$, i.e., for each $v \in \mathcal{V}al_\iota$, v and $\mathcal{J}(v)$ may be identified. We require that $\Sigma_{terms} \cap \Sigma_{theory} \subseteq \mathcal{V}al$. Symbols in $\Sigma_{theory} \setminus \mathcal{V}al$ are *calculation symbols*, for which we may use infix notation. A term in $\mathcal{T}(\Sigma_{theory}, \mathcal{V})$ is called a *theory term*, where $\mathcal{V}$ is a (countably infinite) set of variables. We define the *interpretation* $[\![\cdot]\!]_{\mathcal{J}}$ of ground theory terms as $[\![f(s_1,\ldots,s_n)]\!]_{\mathcal{J}} = \mathcal{J}(f)([\![s_1]\!]_{\mathcal{J}},\ldots,[\![s_n]\!]_{\mathcal{J}})$. Note that for every ground theory term s, there is a unique value-constant c such that $[\![s]\!]_{\mathcal{J}} = [\![c]\!]_{\mathcal{J}}$.

We typically choose a theory signature $\mathcal{S}_{theory}$ such that $\mathcal{S}_{theory} \supseteq \mathcal{S}_{core} = \{bool\}$, $\mathcal{V}al_{bool} = \{\mathsf{true}, \mathsf{false} : bool\}$, $\Sigma_{theory} \supseteq \Sigma_{core} = \mathcal{V}al_{bool} \cup \{\wedge, \vee, \Rightarrow, \Leftrightarrow : bool \times bool \Rightarrow bool,\ \neg : bool \Rightarrow bool\} \cup \{=_\iota, \neq_\iota : \iota \times \iota \Rightarrow bool \mid \iota \in \mathcal{S}_{theory}\}$, $\mathcal{I}(bool) = \{\top, \bot\}$, and $\mathcal{J}$ interprets these symbols as expected: $\mathcal{J}(\mathsf{true}) = \top$ and $\mathcal{J}(\mathsf{false}) = \bot$. We omit the sort subscripts ι from $=_\iota$ and $\neq_\iota$ when they are clear from the context. A theory term with sort *bool* is called a *constraint*. A substitution γ which is a sort-preserving mapping from $T(\Sigma, \mathcal{V})$ to $T(\Sigma, \mathcal{V})$ is said to *respect* a constraint ϕ if $x\gamma \in \mathcal{V}al$ for all $x \in \mathcal{V}ar(\phi)$ and $[\![\phi\gamma]\!]_{\mathcal{J}} = \top$, where $\mathcal{V}ar(\phi)$ denotes the set of variables appearing in ϕ. A constraint ϕ is said to be *valid* (*satisfiable*, resp.) if $[\![\phi\gamma]\!] = \top$ for any (some, resp.) substitution γ respecting ϕ.

A *constrained rewrite rule* is a triple $\ell \to r\ [\varphi]$ such that ℓ and r are terms of the same sort, φ is a constraint, and ℓ is not a theory term (i.e., not a variable). If $\varphi = \mathsf{true}$, then we may write $\ell \to r$. We define $\mathcal{LV}ar(\ell \to r\ [\varphi])$ as $\mathcal{V}ar(\varphi) \cup (\mathcal{V}ar(r) \setminus \mathcal{V}ar(\ell))$, the set of *logical variables in* $\ell \to r\ [\varphi]$ which are variables instantiated with values in rewriting terms. We say that a substitution γ *respects* $\ell \to r\ [\varphi]$ if $\gamma(x) \in \mathcal{V}al$ for all $x \in \mathcal{LV}ar(\ell \to r\ [\varphi])$ and $[\![\varphi\gamma]\!]_{\mathcal{J}} = \top$. Regarding the signature of $\mathcal{R}$, we denote the set $\{f(x_1,\ldots,x_n) \to y\ [y = f(x_1,\ldots,x_n)] \mid f \in \Sigma_{theory} \setminus \mathcal{V}al,\ x_1,\ldots,x_n,y \in \mathcal{V} \text{ are pairwise distinct}\}$ by $\mathcal{R}_{calc}$. The elements of $\mathcal{R}_{calc}$ are called *calculation rules* and we often deal with them as constrained rewrite rules even though their left-hand sides are theory terms. The *rewrite relation* $\to_{\mathcal{R}}$ is a binary relation over terms, defined as follows: For a term s, $s[\ell\gamma]_p \to_{\mathcal{R}} s[r\gamma]_p$ if and only if $\ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc}$ and γ respects $\ell \to r\ [\varphi]$. We may say that the reduction *occurs at position* p and may write $\to_{p,\mathcal{R}}$ or $\to_{p,\ell\to r\ [\varphi]}$ instead of $\to_{\mathcal{R}}$. A reduction step with $\mathcal{R}_{calc}$ is called a *calculation*. A *logically constrained term rewrite system* (LCTRS, for short) is defined as an abstract reduction system $(\mathcal{T}(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$, simply denoted by $\mathcal{R}$, where $\mathcal{R}$ is a set of constrained rewrite rules. An LCTRS is usually given by supplying Σ, $\mathcal{R}$, and an informal description of $\mathcal{I}$ and $\mathcal{J}$ if these are not clear from the context.

The *standard integer signature* Σ_{int} is $\Sigma_{core} \cup \{+, -, \times, \mathsf{exp}, \mathsf{div}, \mathsf{mod} : int \times int \Rightarrow int\} \cup \{\geq, > : int \times int \Rightarrow bool\} \cup \mathcal{V}al_{int}$ where $\mathcal{S}_{theory} \supseteq \{int, bool\}$, $\mathcal{V}al_{int} = \{\mathsf{n} \mid n \in \mathbb{Z}\}$, $\mathcal{I}(int) = \mathbb{Z}$, and $\mathcal{J}(\mathsf{n}) = n$ for any $n \in \mathbb{Z}$—we use n (in sans-serif font) as the value-constant for $n \in \mathbb{Z}$ (in *math* font). We define $\mathcal{J}$ in a natural way. An LCTRS over a signature $\Sigma \supseteq \Sigma_{int}$ with $\Sigma_{theory} = \Sigma_{int}$ is called an *integer LCTRS*.

Example 2.1 $\mathcal{R}_1$ is an integer LCTRS. We have, e.g., the following reduction of $\mathcal{R}_1$: $\mathsf{sum}\times(10) \to_{\mathcal{R}_1} 10 + \mathsf{sum}\times(10 + (-1)) \to_{\mathcal{R}_1} 10 + \mathsf{sum}\times(9) \to_{\mathcal{R}_1} 10 + (9 + \mathsf{sum}\times(9 + (-1))) \to_{\mathcal{R}_1} \cdots \to_{\mathcal{R}_1} 55$.

A function symbol f is called a *defined symbol* of $\mathcal{R}$ if there exists a rule $f(\ell_1,\ldots,\ell_n) \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc}$; non-defined elements of Σ_{theory} are called *constructors* of $\mathcal{R}$. Note that all values are constructors of $\mathcal{R}$. We denote the sets of defined symbols and constructors of $\mathcal{R}$ by $\mathcal{D}_{\mathcal{R}}$ and $\mathcal{C}_{\mathcal{R}}$, respectively: $\mathcal{D}_{\mathcal{R}} = \{f \mid f(\ell_1,\ldots,\ell_n) \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc}\}$ and $\mathcal{C}_{\mathcal{R}} = \Sigma \setminus \mathcal{D}_{\mathcal{R}}$. A $\mathcal{C}_{\mathcal{R}}$-term in $\mathcal{T}(\mathcal{C}_{\mathcal{R}}, \mathcal{V})$ is called a *constructor term* of $\mathcal{R}$. A term of the form $f(t_1,\ldots,t_n)$ with $f \in \mathcal{D}_{\mathcal{R}}$ and constructor terms $t_1,\ldots,t_n$ is called *basic*. We call $\mathcal{R}$ a *constructor system* if the left-hand side of each rule $\ell \to r\ [\varphi] \in \mathcal{R}$ is basic. For any $x \in \mathcal{D}om(\gamma)$, if $\gamma(x)$ (ground) constructor term, then γ is called *(ground) constructor substitution*. An LCTRS $\mathcal{R}$ is called *quasi-reducible* if every ground basic term is a redex of $\mathcal{R}$.

Finally, we define constrained rewriting [13, 6]. A *constrained term* is a pair $s\,[\phi]$ of a term s and a constraint ϕ. The set of instances of s w.r.t. substitutions respecting ϕ is denoted by $\mathscr{I}(s\,[\phi])$: $\mathscr{I}(s\,[\phi]) = \{s\gamma \mid \gamma \text{ is a substitution respecting } \phi\}$. We say that two constrained terms $s\,[\phi]$ and $t\,[\psi]$ are *equivalent*, written as $s\,[\phi] \sim t\,[\psi]$, if both of the following hold: for any substitution γ respecting ϕ, there exists a substitution δ such that δ respects ψ and $s\gamma = t\delta$, and vice versa. Note that $s\,[\phi] \sim t\,[\psi]$ if and only if $\mathscr{I}(s\,[\phi]) = \mathscr{I}(t\,[\psi])$ [6]. We define the *constrained base-rewriting* $\to_{base,\mathcal{R}}$ of an LCTRS $\mathcal{R}$ as follows: $s\,[\phi] \to_{base,\mathcal{R}} t\,[\psi]$ if and only if there exists a freshly-renamed[2] rule $\ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc}$, a position p of s, and a substitution γ such that $\mathcal{D}om(\gamma) \subseteq \mathcal{V}ar(\ell, r, \varphi)$, $s|_p = \ell\gamma$, $t = s[r\gamma]_p$, $\psi = \phi$, $x\gamma \in \mathcal{V}al \cup \mathcal{V}ar(\phi)$ for any variable $x \in \mathcal{LV}ar(\ell \to r\ [\varphi])$, and $(\phi \Rightarrow \varphi\gamma)$ is valid. We define the *constrained rewriting* $\rightharpoonup_{\mathcal{R}}$ as $\sim \cdot \to_{base,\mathcal{R}} \cdot \sim$. When we make the applied position p and rule $\ell \to r\ [\varphi]$ explicit, we may write $\to_{base,p,\mathcal{R}}$ or $\to_{base,p,\ell\to r\ [\varphi]}$ for $\to_{base,\mathcal{R}}$, and $\rightharpoonup_{p,\mathcal{R}}$ or $\rightharpoonup_{p,\ell\to r\ [\varphi]}$ for $\rightharpoonup_{\mathcal{R}}$.

3 Constrained Rewriting Induction

In this section, we recall *constrained rewriting induction* [6] which is rewriting induction for LCTRSs. In the rest of this paper, the abbreviation "RI" stands for "constrained rewriting induction".

A constrained equation $s \approx t\ [\phi]$ is called an *inductive theorem* of a terminating and quasi-reducible LCTRS $\mathcal{R}$ if $s\gamma \leftrightarrow^*_{\mathcal{R}} t\gamma$ for every ground constructor substitution γ such that $\mathcal{D}om(\gamma) \supseteq \mathcal{V}ar(s,t,\phi)$ and γ respects ϕ. By definition, an equation $s \approx t\ [\phi]$ is trivially an inductive theorem of $\mathcal{R}$ if $s = t$ or ϕ is not satisfiable.

We do not distinguish the left- and right-hand sides of $\approx$ (i.e., $\approx$ is symmetric), and thus we do not distinguish $s \approx t\ [\phi]$ and $t \approx s\ [\phi]$. The notation $s \simeq t\ [\phi]$ denotes $s \approx t\ [\phi]$ or $t \approx s\ [\phi]$. In other words, $s \simeq t\ [\phi]$ specifies an equation with a constraint ϕ such that one side of the equation is s and the other is t. In applying $\rightharpoonup$ to constrained equations, we consider $\approx$ a binary constructor which is not commutative.

RI comprises some inference rules over *RI states* of the form $(\mathcal{E}, \mathcal{H})$, where $\mathcal{E}$ is a finite set of equations and $\mathcal{H}$ is an LCTRS. For an RI state $(\mathcal{E} \uplus \{s \simeq t\ [\phi]\}, \mathcal{H})$, the inference rules induce another RI state of the form $(\mathcal{E} \cup \mathcal{E}', \mathcal{H}')$ such that $\mathcal{H} \subseteq \mathcal{H}'$. The main fundamental inference rules of RI for LCTRSs are EXPAND, CASE, SIMPLIFY, DELETE, and EQ-DELETE [6]:

EXPAND

$$(\mathcal{E} \uplus \{s \simeq t\ [\phi]\}, \mathcal{H}) \vdash_{\mathrm{RI}} (\mathcal{E} \cup \mathit{Expd}_{\mathcal{R}}(p,s,t,\phi), \mathcal{H} \cup \{s \to t\ [\phi]\}) \text{ if } \mathcal{R} \cup \mathcal{H} \cup \{s \to t\ [\phi]\} \text{ is terminating}$$

where $s|_p$ is basic for some position p of s, and $\mathit{Expd}_{\mathcal{R}}(p,s,t,\phi) = \{s' \approx t'\ [\psi] \mid \ell \to r\ [\varphi]$ is a freshly renamed variant of a rule in $\mathcal{R}$, γ is a most general unifier of $s|_p$ and ℓ, $\mathcal{R}an(\gamma|_{\mathcal{V}ar(\phi,\varphi)}) \subseteq \mathcal{V}al \cup \mathcal{V}$, $(\phi \wedge \varphi)\gamma$ is satisfiable, $(s\gamma \approx t\gamma\ [(\phi \wedge \varphi)\gamma]) \rightharpoonup_{1.p,\ell\to r\ [\varphi]} (s' \approx t'\ [\psi])\}$.

CASE

$$(\mathcal{E} \uplus \{s \simeq t\ [\phi]\}, \mathcal{H}) \vdash_{\mathrm{RI}} (\mathcal{E} \cup \mathit{Expd}_{\mathcal{R}}(p,s,t,\phi), \mathcal{H})$$

where $s|_p$ is basic for some position p of s.

SIMPLIFY

$$(\mathcal{E} \uplus \{s \approx t\ [\phi]\}, \mathcal{H}) \vdash_{\mathrm{RI}} (\mathcal{E} \cup \{s' \approx t'\ [\phi']\}, \mathcal{H})$$

where $(s \approx t\ [\phi]) \rightharpoonup_{\mathcal{R}\cup\mathcal{H}} (s' \approx t'\ [\phi'])$.

[2] We assume that $\mathcal{V}ar(s,\phi,t,\psi) \cap \mathcal{V}ar(\ell,r,\varphi) = \emptyset$.

DELETE

$$(\mathcal{E} \uplus \{s \approx t\ [\phi]\}, \mathcal{H}) \vdash_{\mathrm{RI}} (\mathcal{E}, \mathcal{H}) \quad \text{if } s = t \text{ or } \phi \text{ is unsatisfiable.}$$

EQ-DELETE

$$(\mathcal{E} \uplus \{s \approx t\ [\phi]\}, \mathcal{H}) \vdash_{\mathrm{RI}} (\mathcal{E} \cup \{s \approx t\ [\phi \wedge \neg(\textstyle\bigwedge_{i=1}^{n}(s_i = t_i))]\}, \mathcal{H})$$

where $s = s[s_1, \ldots, s_n]_{p_1,\ldots,p_n}$, $t = s[t_1, \ldots, t_n]_{p_1,\ldots,p_n}$ for some positions $p_1, \ldots, p_n$ of s and terms $s_1, \ldots, s_n, t_1, \ldots, t_n$ in $\mathcal{T}(\Sigma_{theory}, \mathcal{V}ar(\phi))$.

The EXPAND rule makes a case analysis at an exhaustive case-splitting position p of s by adding the case-split equations $Expd_{\mathcal{R}}(p, s, t, \phi)$ to $\mathcal{E}$ and by adding the rule $s \to t\ [\phi]$ to $\mathcal{H}$ as an induction hypothesis. The CASE rule also makes a case analysis at an exhaustive case-splitting position p, but does not add any induction hypothesis to $\mathcal{H}$. The SIMPLIFY rule applies a rule in $\mathcal{R} \cup \mathcal{H}$ to $s \approx t\ [\phi]$; the application of a rule in $\mathcal{R}$ corresponds to *the simplification by definition*, and the application of a rule in $\mathcal{H}$ corresponds to *the application of induction hypotheses*. The DELETE rule drops an equation that is *trivially* an inductive theorem of $\mathcal{R}$. The EQ-DELETE rule removes from the set of substitutions respecting ϕ some substitutions by conjuncting $\neg(\bigwedge_{i=1}^{n}(s_i = t_i))$ with ϕ: When $s = s[s_1, \ldots, s_n]_{p_1,\ldots,p_n}$ and $t = s[t_1, \ldots, t_n]_{p_1,\ldots,p_n}$ with $s_1, t_1, \ldots, s_n, t_n \in \mathcal{T}(\Sigma_{theory}, \mathcal{V}ar(\phi))$, for any substitution γ respecting $\bigwedge_{i=1}^{n}(s_i = t_i)$, we have that $s\sigma = t\sigma$; thus, it suffices to show that $s \approx t\ [\phi \wedge \neg(\bigwedge_{i=1}^{n}(s_i = t_i))]$ is an inductive theorem of $\mathcal{R}$.

To make it explicit which inference rule is applied, instead of $\vdash_{\mathrm{RI}}$, we write $\vdash_{\mathrm{E}}$, $\vdash_{\mathrm{C}}$, $\vdash_{\mathrm{S}}$, $\vdash_{\mathrm{D}}$, and $\vdash_{\mathrm{Q}}$ for the application of EXPAND, CASE, SIMPLIFY, DELETE, and EQ-DELETE, respectively. We call constrained rewrite rules obtained by the application of EXPAND *IH rewrite rules*. Given a terminating and quasi-reducible LCTRS $\mathcal{R}$ and a finite set $\mathcal{E}_0$ of constrained equations to be proved, RI starts with $(\mathcal{E}_0, \emptyset)$ and attempts to induce $(\emptyset, \mathcal{H})$ for some LCTRS $\mathcal{H}$ by means of the application of inference rules to RI states: If $(\mathcal{E}_0, \emptyset) \vdash_{\mathrm{RI}}^{*} (\emptyset, \mathcal{H})$, then all constrained equations in $\mathcal{E}_0$ are inductive theorems of $\mathcal{R}$.

Example 3.1 The constrained equation (3) in Example 1.1 is proved by RI to be an inductive theorem of $\mathcal{R}_1$ as follows:

$$(\{(3)\ \ \mathsf{sumx}(n) \approx m\ [\Phi_{m,n}]\}, \emptyset)$$

$$\vdash_{\mathrm{E}} \left(\left\{ \begin{array}{c} 0 \approx m\ [n = 0 \wedge \Phi_{m,n}] \\ n + \mathsf{sumx}(n + one) \approx m\ [n > 0 \wedge one = -1 \wedge \Phi_{m,n}] \\ n + \mathsf{sumx}(n + one) \approx m\ [n < 0 \wedge one = 1 \wedge \Phi_{m,n}] \end{array} \right\}, \{\mathsf{sumx}(n) \to m\ [\Phi_{m,n}]\} \right)$$

$$\vdash_{\mathrm{Q}} \left(\left\{ \begin{array}{c} 0 \approx m\ [n = 0 \wedge \Phi_{m,n} \wedge \neg(0 = m)] \\ n + \mathsf{sumx}(n + one) \approx m\ [n > 0 \wedge one = -1 \wedge \Phi_{m,n}] \\ n + \mathsf{sumx}(n + one) \approx m\ [n < 0 \wedge one = 1 \wedge \Phi_{m,n}] \end{array} \right\}, \{\mathsf{sumx}(n) \to m\ [\Phi_{m,n}]\} \right)$$

$$\vdash_{\mathrm{D}}^{2} (\{n + \mathsf{sumx}(n + one) \approx m\ [n > 0 \wedge one = -1 \wedge \Phi_{m,n}]\}, \{\ldots\})$$

$$\vdash_{\mathrm{S}} (\{n + \mathsf{sumx}(y) \approx m\ [n > 0 \wedge one = -1 \wedge \Phi_{m,n} \wedge y = n + one]\}, \{\mathsf{sumx}(n) \to m\ [\Phi_{m,n}]\})$$

$$\vdash_{\mathrm{S}} (\{n + m' \approx m\ [n > 0 \wedge one = -1 \wedge \Phi_{m,n} \wedge y = n + one \wedge \Phi_{m',y}]\}, \{\ldots\})$$

$$\vdash_{\mathrm{Q}} (\{n + m' \approx m\ [n > 0 \wedge one = -1 \wedge \Phi_{m,n} \wedge y = n + one \wedge \Phi_{m',y} \wedge n + m' \neq m]\}, \{\ldots\})$$

$$\vdash_{\mathrm{D}} (\emptyset, \{\mathsf{sumx}(n) \to m\ [\Phi_{m,n}]\})$$

Recall that $\Phi_{m,n} = (n \geq 0 \wedge m = (n \times (n+1))\,\mathsf{div}\,2)$ and $\Psi_{m,n} = (n < 0 \wedge m = -((n \times (n-1))\,\mathsf{div}\,2))$.

4 Merging IH Rewriting Rules at EXPAND Steps

The application of the SIMPLIFY rule w.r.t. IH rewrite rules is the application of induction hypotheses, which usually plays the most important role to succeed in proving given constrained equations to be inductive theorems. Viewed in this light, the broader the scope of the application of IH rewrite rule, the greater the possibilities for proving constrained equations. In this section, by merging IH rewrite rules, we modify the EXPAND rule in order to make the scope of the application of IH rewrite rule broader.

Definition 4.1 *For constrained rewrite rules $\ell \to r\ [\varphi]$ and $\ell' \to r'\ [\varphi']$, we write $(\ell \to r\ [\varphi]) \gtrsim (\ell' \to r'\ [\varphi'])$ if there exists a renaming δ such that*

- $\mathcal{D}om(\delta) \subseteq \mathcal{V}ar(\ell, r, \varphi)$,
- $\ell' = \ell\delta$,
- $r' = r\delta$,
- $\mathcal{V}ar(\ell', r') \cap \mathcal{V}ar(\varphi') = \mathcal{V}ar(\ell\delta, r\delta) \cap \mathcal{V}ar(\varphi\delta)$, *and*
- $(\varphi' \Rightarrow (\exists \vec{x}.\ \varphi\delta))$ *is valid, where* $\{\vec{x}\} = \mathcal{V}ar(\varphi\delta) \setminus \mathcal{V}ar(\ell\delta, r\delta)$.

Note that $(\varphi' \Rightarrow (\exists \vec{x}.\ \varphi\delta))$ is valid if and only if $((\exists \vec{y}.\ \varphi') \Rightarrow (\exists \vec{x}.\ \varphi\delta))$ is valid, where $\{\vec{y}\} = \mathcal{V}ar(\varphi') \setminus \mathcal{V}ar(\ell', r')$. To make δ explicit, we often write $(\ell \to r\ [\varphi]) \gtrsim_\delta (\ell' \to r'\ [\varphi'])$.

Roughly speaking, rules $\ell \to r\ [\varphi]$ and $\ell' \to r'\ [\varphi']$ with $(\ell \to r\ [\varphi]) \gtrsim (\ell' \to r'\ [\varphi'])$ have the same rule part $\ell\delta \to r\delta$ $(= (\ell' \to r'))$ but the constraint part φ' is weaker than or equal to $\varphi\delta$.

Example 4.2 Regarding the constrained rewrite rules in Example 1.2, all of the following hold:

- $(\mathsf{sumx}(n) \to m\,[\Phi_{m,n} \vee \Psi_{m,n}]) \gtrsim (\mathsf{sumx}(n) \to m\,[\Phi_{m,n}])$,
- $(\mathsf{sumx}(n) \to m\,[\Phi_{m,n} \vee \Psi_{m,n}]) \gtrsim (\mathsf{sumx}(n) \to m\,[\Psi_{m,n}])$. and
- $(\mathsf{sumx}(n) \to m\,[\Phi_{m,n}]) \not\gtrsim (\mathsf{sumx}(n) \to m\,[\Psi_{m,n}])$.

Proposition 4.3 *Let $\mathcal{R}$ be an LCTRS and $\ell \to r\ [\varphi], \ell' \to r'\ [\varphi']$ constrained rewrite rules in $\mathcal{R}$. If $(\ell \to r\ [\varphi]) \gtrsim (\ell' \to r'\ [\varphi'])$, then $\to_{\ell \to r\ [\varphi]} \supseteq \to_{\ell' \to r'\ [\varphi']}$.*

Proof. Assume that $(\ell \to r\ [\varphi]) \gtrsim_\delta (\ell' \to r'\ [\varphi'])$ for some renaming δ. Then, by definition, we have that $\mathcal{D}om(\delta) \subseteq \mathcal{V}ar(\ell, r, \varphi)$, $\ell' = \ell\delta$, $r' = r\delta$, $\mathcal{V}ar(\ell', r') \cap \mathcal{V}ar(\varphi') = \mathcal{V}ar(\ell\delta, r\delta) \cap \mathcal{V}ar(\varphi\delta)$, and $(\varphi' \Rightarrow (\exists \vec{x}.\ \varphi\delta))$ is valid, where $\{\vec{x}\} = \mathcal{V}ar(\varphi\delta) \setminus \mathcal{V}ar(\ell\delta, r\delta)$. Suppose that $s\ [\phi] \sim s'[\ell'\gamma']_p\ [\phi'] \to_{base, \ell' \to r'\ [\varphi']} s'[r'\gamma']_p\ [\phi'] \sim t\ [\psi]$, where $\mathcal{D}om(\gamma') \subseteq \mathcal{V}ar(\ell', r', \varphi')$, $x\gamma' \in \mathcal{V}al \cup \mathcal{V}ar(\phi')$ for any variable $x \in \mathcal{LV}ar(\ell' \to r'\ [\varphi'])$, and $(\phi' \Rightarrow \varphi'\gamma')$ is valid. Hence, we have that $\mathcal{V}ar(\phi') = \mathcal{V}ar(\varphi'\gamma')$. Since $\{\vec{x}\} = \mathcal{V}ar(\varphi\delta) \setminus \mathcal{V}ar(\ell\delta, r\delta) = \mathcal{V}ar(\varphi\delta) \setminus \mathcal{V}ar(\ell', r')$, we have that $\mathcal{V}ar(\phi'\gamma') = \mathcal{V}ar((\exists \vec{x}.\ \varphi\delta)\gamma')$. It follows from validity of $(\varphi' \Rightarrow (\exists \vec{x}.\ \varphi\delta))$ and $(\phi' \Rightarrow \varphi'\gamma')$ that ϕ', $\phi' \wedge \varphi'\gamma$, and $\phi' \wedge (\exists \vec{x}.\ \varphi\delta)\gamma'$ are equivalent. Thus, we have that $s'[\ell'\gamma']_p\ [\phi'] \sim s'[\ell'\gamma']_p\ [\phi' \wedge (\exists \vec{x}.\ \varphi\delta)\gamma')]$ and $s'[r'\gamma']_p\ [\phi'] \sim s'[r'\gamma']_p\ [\phi' \wedge (\exists \vec{x}.\ \varphi\delta)\gamma')]$. Therefore, $s\ [\phi] \sim s'[\ell'\gamma']_p\ [\phi' \wedge (\exists \vec{x}.\ \varphi\delta)\gamma')] = s'[\ell\delta\gamma']_p\ [\phi' \wedge (\exists \vec{x}.\ \varphi)\delta\gamma')] \to_{base, \ell \to r\ [\varphi]} s'[r\delta\gamma']_p\ [\phi' \wedge (\exists \vec{x}.\ \varphi)\delta\gamma')] \sim t\ [\psi]$, and thus $s\ [\phi] \to_{\ell \to r\ [\varphi']} t\ [\psi]$. □

We say that an LCTRS $\mathcal{H}$ is *minimal* if for any rule $\ell \to r\ [\varphi] \in \mathcal{H}$, there is no constrained rewrite rule $\ell' \to r'\ [\varphi] \in \mathcal{H} \setminus \{\ell \to r\ [\varphi]\}$ such that $(\ell' \to r'\ [\varphi']) \gtrsim (\ell \to r\ [\varphi])$. A minimal LCTRS $\mathcal{H}$ is said to be a *minimal* system of an LCTRS $\mathcal{H}'$ if both of the following statements hold:

- $\to_{\mathcal{H}} = \to_{\mathcal{H}'}$, and

- for any constrained rewrite rule $\ell' \to r' \ [\varphi']$ in $\mathcal{H}'$, there exists a constrained rewrite rule $\ell \to r \ [\varphi]$ in $\mathcal{H}$ such that $(\ell \to r \ [\varphi]) \gtrsim (\ell' \to r' \ [\varphi'])$.

Proposition 4.4 *Let $\mathcal{H}, \mathcal{H}'$ be LCTRSs such that $\mathcal{H}$ is a minimal system of $\mathcal{H}'$. Then, $\to_{\mathcal{H}} \supseteq \to_{\mathcal{H}'}$.*

Proof. Trivial by Proposition 4.3. □

We now define an operation that merges a constrained rewrite rule to an LCTRS.

Definition 4.5 (*merge*) *We say that a constrained rewrite rule $\ell \to r \ [\varphi]$* can be merged with *a constrained rewrite rule $\ell' \to r' \ [\varphi']$* via *a renaming δ if all of the following statements hold:*

- $\ell = \ell'\delta$,
- $r = r'\delta$,
- $\mathcal{V}ar(\varphi) \cap \mathcal{V}ar(\ell, r) = \mathcal{V}ar(\varphi'\delta) \cap \mathcal{V}ar(\ell'\delta, r'\delta)$, *and*
- $(\mathcal{V}ar(\varphi'\delta) \setminus \mathcal{V}ar(\ell'\delta, r'\delta)) \cap \mathcal{V}ar(\ell, r, \varphi) = \emptyset$.

We define an operation merge, which takes a constrained rewrite rule and an LCTRS as input and returns an LCTRS, as follows:

- $\mathit{merge}(\ell \to r \ [\varphi], \emptyset) = \{\ell \to r \ [\varphi]\}$,
- $\mathit{merge}(\ell \to r \ [\varphi], \{\ell' \to r' \ [\varphi']\} \uplus \mathcal{H}) = \{\ell \to r \ [\varphi \vee \varphi'\delta]\} \cup \mathcal{H}$ *if $\ell \to r \ [\varphi]$ can be merged with $\ell' \to r' \ [\varphi']$ via a renaming δ, and*
- *otherwise,* $\mathit{merge}(\ell \to r \ [\varphi], \{\ell' \to r' \ [\varphi']\} \uplus \mathcal{H}) = \{\ell' \to r' \ [\varphi']\} \cup \mathit{merge}(\ell \to r \ [\varphi], \mathcal{H})$.

By definition, it is clear that if $\ell \to r \ [\varphi]$ can be merged with $\ell' \to r' \ [\varphi']$ via a renaming δ, then $(\ell \to r \ [\varphi \vee \varphi'\delta]) \gtrsim_{\varnothing} (\ell \to r \ [\varphi])$, $(\ell \to r \ [\varphi \vee \varphi'\delta]) \gtrsim_{\varnothing} (\ell \to r \ [\varphi'\delta])$, and $(\ell \to r \ [\varphi \vee \varphi'\delta]) \gtrsim_{\delta|_{\mathcal{V}ar(\ell,r,\varphi)}} (\ell' \to r' \ [\varphi'])$, where $\varnothing$ is the identity substitution.

Example 4.6 Let us consider the constrained rewrite rules in Example 1.2 again. The constrained rewrite rule $\mathsf{sum}\times(n) \to m \ [\Phi_{m,n}]$ can be merged with $\mathsf{sum}\times(n) \to m \ [\Psi_{m,n}]$ via the identity renaming $\varnothing$.

The merge operation has the following property.

Proposition 4.7 *Let $\mathcal{H}$ be a minimal LCTRS, and $\ell \to r \ [\varphi]$ a constrained rewrite rule. Then, $\mathit{merge}(\ell \to r \ [\varphi], \mathcal{H})$ is unique and a minimal LCTRS of $\{\ell \to r \ [\varphi]\} \cup \mathcal{H}$.*

Proof. Since $\mathcal{H}$ is minimal, there exists *at most one* constrained rewrite rule in $\mathcal{H}$ that can be merged with $\ell \to r \ [\varphi]$. We make a case analysis depending on whether there exists a constrained rewrite rule $\ell' \to r' \ [\varphi']$ in $\mathcal{H}$ such that $\ell \to r \ [\varphi]$ can be merged with $\ell' \to r' \ [\varphi']$.

- Case where there is no such a rule in $\mathcal{H}$. In this case, we have that $\mathit{merge}(\ell \to r \ [\varphi], \mathcal{H}) = \{\ell \to r \ [\varphi]\} \cup \mathcal{H}$. Recall that $\mathcal{H}$ is minimal. Thus, $\mathit{merge}(\ell \to r \ [\varphi], \mathcal{H})$ is unique and $\to_{\{\ell \to r \ [\varphi]\} \cup \mathcal{H}} = \to_{\mathit{merge}(\ell \to r \ [\varphi], \mathcal{H})}$. Therefore, the claim trivially holds.
- Case where there exists such a rule in $\mathcal{H}$. Let $\ell' \to r' \ [\varphi'] \in \mathcal{H}$ be a constrained rewrite rule that can be merged with $\ell \to r \ [\varphi]$ via a renaming δ. Let $\mathcal{H}' = \mathcal{H} \setminus \{\ell' \to r' \ [\varphi']\}$. Then, by definition, we have that $\mathit{merge}(\ell \to r \ [\varphi], \mathcal{H}) = \{\ell \to r \ [\varphi \vee \varphi'\delta]\} \cup \mathcal{H}'$ and there is no rule in $\mathcal{H}'$ that can be merged with $\ell \to r \ [\varphi]$. Thus, $\mathit{merge}(\ell \to r \ [\varphi], \mathcal{H})$ is unique. Since $\mathcal{H}$ is minimal,

$\mathcal{H}'$ is also minimal and there is no rule in $\mathcal{H}'$ that can be merged with $\ell \to r\ [\varphi \vee \varphi'\delta]$. Thus, $\mathit{merge}(\ell \to r\ [\varphi], \mathcal{H})$ is minimal. It is clear that $\ell \to r\ [\varphi] \lesssim \ell \to r\ [\varphi \vee \varphi'\delta]$ and $(\ell' \to r'\ [\varphi']) \lesssim (\ell \to r\ [\varphi \vee \varphi'\delta])$. It follows from Proposition 4.3 that $(\to_{\ell \to r\ [\varphi]} \cup \to_{\ell' \to r'\ [\varphi']}) \subseteq \to_{\ell \to r\ [\varphi \vee \varphi'\delta]}$, and thus $\to_{\{\ell \to r\ [\varphi]\} \cup \mathcal{H}} \subseteq \to_{\mathit{merge}(\ell \to r\ [\varphi], \mathcal{H})}$. Hence, $\mathit{merge}(\ell \to r\ [\varphi], \mathcal{H})$ is unique and $\to_{\{\ell \to r\ [\varphi]\} \cup \mathcal{H}} \subseteq \to_{\mathit{merge}(\ell \to r\ [\varphi], \mathcal{H})}$. By definition, it is trivial that $\to_{\{\ell \to r\ [\varphi]\} \cup \mathcal{H}} \subseteq \to_{\mathit{merge}(\ell \to r\ [\varphi], \mathcal{H})}$. Thus, we show that $\to_{\{\ell \to r\ [\varphi]\} \cup \mathcal{H}} \supseteq \to_{\mathit{merge}(\ell \to r\ [\varphi], \mathcal{H})}$. Assume that $s \to_{\mathit{merge}(\ell \to r\ [\varphi], \mathcal{H})} t$. Since the case where the applied rule to the step is not $\ell \to r\ [\varphi \vee \varphi'\delta]$ is trivial, we consider the remaining case, i.e., we assume that $\ell \to r\ [\varphi \vee \varphi'\delta]$ is applied at a position p of s. Then, there exists a substitution γ such that $s|_p = \ell\gamma$, $t = s[r\gamma]_p$, and γ respects $\varphi \vee \varphi'\delta$, and thus $[\![(\varphi \vee \varphi'\delta)\gamma]\!] = \top$. By definition, $\ell \mathit{tor}\ [\varphi]$ or $\ell' \to r'\ [\varphi'] \in \mathcal{H}$ are applicable to s at p, and thus we have that $s \to_{\{\ell \to r\ [\varphi]\} \cup \mathcal{H}} t$. Therefore, the claim holds. □

We introduce the merge operation to EXPAND.

EXPAND$_{\mathit{merge}}$

$$(\mathcal{E} \uplus \{s \simeq t\ [\phi]\}, \mathcal{H}) \vdash_{\mathsf{RIm}} (\mathcal{E} \cup \mathit{Expd}_{\mathcal{R}}(p, s, t, \phi), \mathit{merge}(s \to t\ [\phi], \mathcal{H}))$$

if $\mathcal{R} \cup \mathcal{H} \cup \{s \to t\ [\phi]\}$ is terminating, where $s|_p$ is basic for some position p of s.

We denote by $\vdash_{\mathsf{RIm}}$ the step of RI where EXPAND is replaced by EXPAND$_{\mathit{merge}}$.

Example 4.8 The proof steps in Example 1.2 are the relation over RI states w.r.t. $\vdash_{\mathsf{RIm}}$.

Regarding *merge*, we have the following properties.

Proposition 4.9 *Let $\mathcal{R}$ be an LCTRS, $\mathcal{H}$ a minimal LCTRS, and $s \to t\ [\phi]$ a constrained rewrite rule. Then, $\mathcal{R} \cup \mathcal{H} \cup \{s \to t\ [\phi]\}$ is terminating if and only if so is $\mathcal{R} \cup \mathit{merge}(s \to t\ [\phi], \mathcal{H})$.*

Proof. It follows from Proposition 4.7 that $\mathit{merge}(s \to t\ [\phi], \mathcal{H})$ is a minimal system of $\mathcal{H} \cup \{s \to t\ [\phi]\}$, and thus $\to_{\mathcal{H} \cup \{s \to t\ [\phi]\}} = \to_{\mathit{merge}(s \to t\ [\phi], \mathcal{H})}$. Therefore, the claim holds. □

Proposition 4.10 *Let $(\mathcal{E}, \mathcal{H}), (\mathcal{E}', \mathcal{H}')$ be RI states such that $(\mathcal{E}, \mathcal{H}) \vdash_{\mathsf{RIm}} (\mathcal{E}', \mathcal{H}')$. If $\mathcal{H}$ is minimal, then so is $\mathcal{H}'$.*

Proof. Trivial by Proposition 4.7. □

The modification of EXPAND does not lose the proof power of RI.

Theorem 4.11 *Let $\mathcal{R}, \mathcal{H}$ be LCTRSs, $\mathcal{E}_0, \mathcal{E}$ finite sets of constrained equations, and n a natural number. If $(\mathcal{E}_0, \emptyset) \vdash_{\mathrm{RI}}^{n} (\mathcal{E}, \mathcal{H})$, then there exists an LCTRS $\mathcal{H}'$ such that $(\mathcal{E}_0, \emptyset) \vdash_{\mathsf{RIm}}^{n} (\mathcal{E}, \mathcal{H}')$ and $\mathcal{H}'$ is a minimal system of $\mathcal{H}$.*

Proof. We prove this claim by induction on n. Since the case where $n = 0$ is trivial, we consider the remaining case where $n > 0$. Suppose that $(\mathcal{E}_0, \emptyset) \vdash_{\mathrm{RI}}^{n-1} (\mathcal{E}_1, \mathcal{H}_1) \vdash_{\mathsf{RIm}} (\mathcal{E}, \mathcal{H})$. Then, by the induction hypothesis, there exists an LCTRS $\mathcal{H}_1'$ such that $(\mathcal{E}_0, \emptyset) \vdash_{\mathsf{RIm}}^{n-1} (\mathcal{E}_1, \mathcal{H}_1')$ and $\mathcal{H}_1'$ is a minimal system of $\mathcal{H}_1$. We make a case analysis depending on which inference rule is applied at the step of $(\mathcal{E}_1, \mathcal{H}_1) \vdash_{\mathrm{RI}} (\mathcal{E}, \mathcal{H})$. Since the case where the applied rule is CASE, DELETE, or EQ-DELETE is not relevant to $\mathcal{H}$ (and thus $\mathcal{H} = \mathcal{H}_1$ in this case), it trivially holds that $(\mathcal{E}_1, \mathcal{H}_1) \vdash_{\mathsf{RIm}} (\mathcal{E}, \mathcal{H}_1) = (\mathcal{E}, \mathcal{H})$. Thus, we consider the remaining case.

- Case where the applied rule is EXPAND. Suppose that $(\mathcal{E}_1,\mathcal{H}_1) = (\mathcal{E}'_1 \uplus \{s \simeq t\ [\phi]\},\mathcal{H}_1) \vdash_{\mathrm{RI}} (\mathcal{E}'_1 \cup \mathit{Expd}_{\mathcal{R}}(p,s,t,\phi),\mathcal{H}_1 \cup \{s \to t\ [\phi]\}) = (\mathcal{E},\mathcal{H})$, where $\mathcal{R} \cup \mathcal{H}_1 \cup \{s \to t\ [\phi]\}$ is terminating and $s|_p$ is basic for some position p of s. It follows from Proposition 4.9 that $\mathcal{R} \cup \mathit{merge}(s \to t\ [\phi],\mathcal{H}_1)$ is terminating. Thus, we have that $(\mathcal{E}_1,\mathcal{H}'_1) = (\mathcal{E}'_1 \uplus \{s \simeq t\ [\phi]\},\mathcal{H}'_1) \vdash_{\mathsf{RIm}} (\mathcal{E}'_1 \cup \mathit{Expd}_{\mathcal{R}}(p,s,t,\phi),\ \mathit{merge}(s \to t\ [\phi],\mathcal{H}'_1)) = (\mathcal{E},\mathit{merge}(s \to t\ [\phi],\mathcal{H}'_1))$. It follows from Proposition 4.7 that $\mathit{merge}(s \to t\ [\phi],\mathcal{H}'_1)$ is a minimal system of $\mathcal{H}_1 \cup \{s \to t\ [\phi]\}$ $(= \mathcal{H})$. Therefore, we have that $(\mathcal{E}_0,\emptyset) \vdash^{n-1}_{\mathsf{RIm}} (\mathcal{E}_1,\mathcal{H}'_1) = (\mathcal{E}'_1 \uplus \{s \simeq t\ \ [\phi]\},\mathcal{H}'_1) \vdash_{\mathsf{RIm}} (\mathcal{E}'_1 \cup \mathit{Expd}_{\mathcal{R}}(p,s,t,\phi),\mathit{merge}(s \to t\ \ [\phi],\mathcal{H}'_1)) = (\mathcal{E},\ \mathit{merge}(s \to t\ [\phi],\mathcal{H}'_1))$.
- Case where the applied rule is SIMPLIFY. Suppose that $(\mathcal{E}_1,\mathcal{H}_1) = (\mathcal{E}'_1 \uplus \{s \approx t\ [\phi]\},\mathcal{H}_1) \vdash_{\mathrm{RI}} (\mathcal{E}'_1 \cup \{s' \approx t'\ [\phi']\},\mathcal{H}_1) = (\mathcal{E},\mathcal{H})$, where $(s \approx t\ [\phi]) \to_{\mathcal{R} \cup \mathcal{H}_1} (s' \approx t'\ [\phi'])$. Since $\mathcal{H}'_1$ is a minimal system of $\mathcal{H}_1$, it follows from Proposition 4.4 that $(s \approx t\ [\phi]) \to_{\mathcal{R} \cup \mathcal{H}'_1} (s' \approx t'\ [\phi'])$. Therefore, we have that $(\mathcal{E}_0,\emptyset) \vdash^{n-1}_{\mathsf{RIm}} (\mathcal{E}_1,\mathcal{H}'_1) = (\mathcal{E}'_1 \uplus \{s \approx t\ [\phi]\},\mathcal{H}'_1) \vdash_{\mathrm{RI}} (\mathcal{E}'_1 \cup \{s' \approx t'\ [\phi']\},\mathcal{H}'_1) = (\mathcal{E},\mathcal{H}'_1)$. □

5 Conclusion

In this paper, we modified the EXPAND rule of RI for LCTRSs and showed that this modification does not lose the proof power of RI. We have not yet known whether the modification of EXPAND (i.e., $\mathrm{EXPAND}_{\mathit{merge}}$) is an improvement of RI. That is, for the present, the following is an open problem: If $(\mathcal{E},\emptyset) \vdash^*_{\mathsf{RIm}} (\emptyset,\mathcal{H})$, then $(\mathcal{E},\emptyset) \vdash^*_{\mathrm{RI}} (\emptyset,\mathcal{H}')$ for some LCTRS $\mathcal{H}'$. As a future work, we will try to prove this open problem or to find a counterexample. Furthermore, we will implement the $\mathrm{EXPAND}_{\mathit{merge}}$ rule in an RI-based tool for LCTRSs and evaluate the usefulness of the $\mathrm{EXPAND}_{\mathit{merge}}$ rule from the viewpoint of efficiency, by means of several examples.

References

[1] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, doi:10.1145/505863.505888.

[2] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2016): *The Satisfiability Modulo Theories Library (SMT-LIB)*. `https://www.SMT-LIB.org`.

[3] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia & Cesare Tinelli (2021): *Satisfiability Modulo Theories*. In: *Handbook of Satisfiability*, Second edition, *Frontiers in Artificial Intelligence and Applications* 336, IOS Press, pp. 1267–1329, doi:10.3233/FAIA201017.

[4] Ştefan Ciobâcă & Dorel Lucanu (2018): *A Coinductive Approach to Proving Reachability Properties in Logically Constrained Term Rewriting Systems*. In: *Proc. IJCAR 2018*, *LNCS* 10900, Springer, pp. 295–311, doi:10.1007/978-3-319-94205-6_20.

[5] Maribel Fernández (2014): *Programming Languages and Operational Semantics – A Concise Overview*. Undergraduate Topics in Computer Science, Springer, doi:10.1007/978-1-4471-6368-8.

[6] Carsten Fuhs, Cynthia Kop & Naoki Nishida (2017): *Verifying Procedural Programs via Constrained Rewriting Induction*. *ACM Trans. Comput. Log.* 18(2), pp. 14:1–14:50, doi:10.1145/3060143.

[7] Yoshiaki Kanazawa & Naoki Nishida (2019): *On Transforming Functions Accessing Global Variables into Logically Constrained Term Rewriting Systems*. In: *Proc. WPTE 2018*, *EPTCS* 289, Open Publishing Association, pp. 34–52.

[8] Yoshiaki Kanazawa, Naoki Nishida & Masahiko Sakai (2019): *On Representation of Structures and Unions in Logically Constrained Rewriting*. IEICE Technical Report SS2018-38, IEICE. Vol. 118, No. 385, pp. 67–72, in Japanese.

[9] Misaki Kojima & Naoki Nishida (2023): *From Starvation Freedom to All-Path Reachability Problems in Constrained Rewriting*. In: *Proc. PADL 2023*, *LNCS* 13880, Springer Nature Switzerland, pp. 161–179, doi:10.1007/978-3-031-24841-2_11.

[10] Misaki Kojima & Naoki Nishida (2023): *Reducing non-occurrence of specified runtime errors to all-path reachability problems of constrained rewriting*. *J. Log. Algebraic Methods Program.* 135, pp. 1–19, doi:10.1016/j.jlamp.2023.100903.

[11] Misaki Kojima, Naoki Nishida & Yutaka Matsubara (2020): *Transforming Concurrent Programs with Semaphores into Logically Constrained Term Rewrite Systems*. In: *Informal Proc. WPTE 2020*, pp. 1–12.

[12] Misaki Kojima, Naoki Nishida & Yutaka Matsubara (2025): *Transforming concurrent programs with semaphores into logically constrained term rewrite systems*. *J. Log. Algebraic Methods Program.* 143, pp. 1–23, doi:10.1016/j.jlamp.2024.101033.

[13] Cynthia Kop & Naoki Nishida (2013): *Term Rewriting with Logical Constraints*. In: *Proc. FroCoS 2013*, *LNCS* 8152, Springer, pp. 343–358, doi:10.1007/978-3-642-40885-4_24.

[14] Cynthia Kop & Naoki Nishida (2015): *Constrained Term Rewriting tooL*. In: *Proc. LPAR-20*, *LNCS* 9450, Springer, pp. 549–557, doi:10.1007/978-3-662-48899-7_38.

[15] Naoki Nishida, Misaki Kojima & Ayuka Matsumi (2025): *A nesting-preserving transformation of SIMP programs into logically constrained term rewrite systems*. *J. Log. Algebraic Methods Program.* 144, pp. 1–15, doi:10.1016/j.jlamp.2025.101045. Available at `https://doi.org/10.1016/j.jlamp.2025.101045`.

[16] Naoki Nishida & Sarah Winkler (2018): *Loop Detection by Logically Constrained Term Rewriting*. In: *Proc. VSTTE 2018*, *LNCS* 11294, Springer, pp. 309–321, doi:10.1007/978-3-030-03592-1_18.

[17] Enno Ohlebusch (2002): *Advanced Topics in Term Rewriting*. Springer, doi:10.1007/978-1-4757-3661-8.

[18] Uday S. Reddy (1990): *Term Rewriting Induction*. In: *Proc. CADE 1990*, *LNCS* 449, Springer, pp. 162–177, doi:10.1007/3-540-52885-7_86.

[19] Sarah Winkler & Aart Middeldorp (2018): *Completion for Logically Constrained Rewriting*. In: *Proc. FSCD 2018*, *LIPIcs* 108, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 30:1–30:18, doi:10.4230/LIPIcs.FSCD.2018.30.

Bounded Rewriting Induction for LCSTRSs

Kasper Hagens
Radboud University Nijmegen, Netherlands
kasper.hagens@ru.nl

Cynthia Kop
Radboud University Nijmegen, Netherlands
c.kop@cs.ru.nl

Rewriting Induction (RI) is a method for inductive theorem proving in equational reasoning, introduced in 1990 by Reddy. Using Logically Constrained Simply-typed Term Rewriting Systems (LCSTRSs) makes it into an interesting tool for program verification (in particular program equivalence), as LCSTRSs closely describe real-life programming. Correctness of RI depends on well-founded induction, and one of the core obstacles for obtaining a practically useful proof system is to find suitable well-founded orderings automatically. Using naive approaches, induction hypotheses must be oriented within the well-founded ordering, leading to very strong ordering requirements, which in turn, severely limits the proof capacity of RI. Here, we introduce bounded RI: an adaption of RI for LCSTRSs where such requirements are being minimized.

1 Introduction

Rewriting Induction (RI) is a proof system for showing equations $s \approx t$ to be inductive theorems, meaning that every variable-free instance of $s \approx t$ is related by $\leftrightarrow^*_{\mathcal{R}}$: the reflexive, transitive closure of $\leftrightarrow_{\mathcal{R}} = \rightarrow_{\mathcal{R}} \cup \leftarrow_{\mathcal{R}}$ (and with $\mathcal{R}$ the set of rewrite rules that completely describe the reduction behavior of s and t). RI was introduced by Reddy [10], as a method to validate inductive proof procedures based on Knuth-Bendix completion. Classically, it is used in equational reasoning to prove properties of inductively defined mathematical structures such as natural numbers or lists. For example, one could use RI to prove an equation $\mathsf{add}(x,y) \approx \mathsf{add}(y,x)$, expressing commutativity of addition on the natural numbers. It was adapted to constrained rewriting [6], and recently to higher-order constrained rewriting [9]. These formalisms closely relate to real-life programming and therefore have a natural place in the larger toolbox for program verification. Programs are represented by term rewriting systems, and inductive theorems provide an interpretation of program equivalence.

Why constrained rewriting? Using RI for program equivalence somewhat differs from the standard setting in equational reasoning where, for example, the Peano axioms are used to prove statements about the natural numbers. In our case, we are not so much interested in proving properties about the natural numbers themselves, but about programs that operate on them. Of course, we can express the Peano axioms as rewrite rules and use this to define programs on natural numbers. However, the disadvantage of this approach is that we also have to define add and mul to represent $+$ and $*$. In this way, studying program equivalence becomes a cumbersome experience, getting involved in complicated interactions between add, mul and the program definition itself. Like in real-life programming, we want to treat the natural numbers as being given for free. With standard term rewriting this is not possible.

Constrained rewriting provides a solution here, as they natively support primitive data structures, such as natural numbers and integers. This makes it possible to distinguish between the actual program definition (represented by rewrite rules), and underlying data structures with their operators (represented by distinguished terms with pre-determined semantical interpretations). This allows us to shift some of

the proof-burden from the rewriting side to the semantical side (which e.g. could be handled by an SMT solver).

In constrained rewriting, rewrite rules are of the shape $s \to t\ [\varphi]$ where the boolean constraint φ acts as a guard, managing control flow over primitive data structures (such as the natural numbers). Here, we will consider Logically Constrained Simply-typed Term Rewriting Systems (LCSTRSs), which concerns applicative higher-order rewriting (without λ abstractions) and first-order constraints [8]. In particular, we will build on our earlier work [9] where we defined RI for LCSTRSs.

Rewriting Induction and Termination The name Rewriting Induction refers to the principle that for a terminating rewrite system $\mathcal{R}$, the rewrite relation $\to^+_{\mathcal{R}}$ defines a well-founded order on the set of all terms, and therefore can be used for proofs by well-founded induction. In many cases, however, we will need a well-founded order $\succ$ which is strictly larger than $\to^+_{\mathcal{R}}$. This is because the role of induction hypothesis in RI is also taken by equations, which must be applied like a rewrite rule, in a decreasing direction w.r.t. $\succ$. That is, we are only allowed to use an induction hypothesis $s \approx t$ if $s \succ t$ or $t \succ s$ holds. Consequently, termination of $\mathcal{R}$ itself is not enough, since equations are not usually orientable by $\to^+_{\mathcal{R}}$.

One solution to this problem is to for instance let $\succ = \to^+_{\mathcal{R}\cup\{s\to t\}}$; or, in the case of multiple induction hypothesis, to collect all the corresponding rewrite rules in a set $\mathcal{H}$ and use $\succ = \to^+_{\mathcal{R}\cup\mathcal{H}}$. However, doing this leaves us with a proof obligation to show termination of $\mathcal{R}\cup\mathcal{H}$. Even if we already know that $\mathcal{R}$ is terminating, it may not be easy or even possible to prove that the same holds for $\mathcal{R}\cup\mathcal{H}$ (think for instance of an induction hypothesis $\mathsf{add}(x,y) \approx \mathsf{add}(y,x)$, which is not orientable in either direction). In such a situation a RI proof might get stuck.

Our goal is to redefine RI for LCSTRSs in such a way that we minimize the termination requirements. As already observed by Reddy [10], we do not necessarily need every induction hypothesis being oriented, as long if we can guarantee that an induction rule $s \to t$ is only applied to terms $\succ$-smaller than s. For this, it is not required to choose the well-founded ordering $\succ = \to^+_{\mathcal{R}\cup\mathcal{H}}$. Reddy proposed to use modulo rewriting to build a well-founded $\succ$ which may not need to contain all induction rules. This approach was investigated by Aoto, who introduced several extensions of RI for first-order unconstrained rewriting [1, 2, 3]. Here we will follow a strategy along the same idea: by redefining RI we can construct a well-founded relation $\succ$ during the RI process, aiming to keep it as small as possible.

2 Preliminaries

2.1 Logically Constrained Simply Typed Rewriting Systems

Types and Terms Assume a set of sorts (base types) $\mathcal{S}$; the set $\mathcal{T}$ of types is defined by the grammar $\mathcal{T} ::= \mathcal{S} \mid \mathcal{T} \to \mathcal{T}$. Here, $\to$ is right-associative, so all types may be written as $type_1 \to \ldots \to type_m \to sort$ with $m \geq 0$. We also assume a subset $\mathcal{S}_{theory} \subseteq \mathcal{S}$ of *theory sorts* (e.g., int and bool), and define the *theory types* by $\mathcal{T}_{theory} ::= \mathcal{S}_{theory} \mid \mathcal{S}_{theory} \to \mathcal{T}_{theory}$. Each theory sort $\iota \in \mathcal{S}_{theory}$ is associated with a non-empty set $\mathcal{I}_\iota$ (e.g., $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$, the set of all integers), and we let $\mathcal{I}_{\iota\to\sigma}$ be the set of functions from $\mathcal{I}_\iota$ to $\mathcal{I}_\sigma$.

We assume a signature Σ of *function symbols* and a disjoint set $\mathcal{V}$ of variables, and a function *typeof* from $\Sigma\cup\mathcal{V}$ to $\mathcal{T}$; we require that there are infinitely many variables of all types. The set of terms $T(\Sigma,\mathcal{V})$ over Σ and $\mathcal{V}$ are the expressions in $\mathbb{T}$ – defined by the grammar $\mathbb{T} ::= \Sigma \mid \mathcal{V} \mid \mathbb{T}\ \mathbb{T}$ – that are *well-typed*: if $s :: \sigma \to \tau$ and $t :: \sigma$ then $s\ t :: \tau$, and $a :: typeof(a)$ for $a \in \Sigma\cup\mathcal{V}$.

Application is left-associative, which allows all terms to be written in a form $a\ t_1 \cdots t_n$ with $a \in \Sigma\cup\mathcal{V}$ and $n \geq 0$. Writing $t = a\ t_1 \cdots t_n$, we define $head(t) = a$. For a term t, let $Var(t)$ be the set of variables in

t. A term t is *ground* if $Var(t) = \emptyset$. We say that a type is *inhabited* if there are ground terms of that type. We assume that Σ is the disjoint union $\Sigma_{theory} \uplus \Sigma_{terms}$, where $typeof(\mathsf{f}) \in \mathcal{T}_{theory}$ for all $\mathsf{f} \in \Sigma_{theory}$. Each $\mathsf{f} \in \Sigma_{theory}$ has an interpretation $[\![\mathsf{f}]\!] \in \mathcal{I}_{typeof(\mathsf{f})}$. For example, a theory symbol $* :: \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ may be interpreted as multiplication on $\mathbb{Z}$. We use infix notation for the binary symbols, or use $[\mathsf{f}]$ for prefix or partially applied notation (e.g., $[+]\ x\ y$ and $x+y$ are the same).

Symbols in Σ_{terms} do not have an interpretation since their behavior will be defined through the rewriting system. Values are theory symbols of base type, i.e. $\mathcal{V}al = \{v \in \Sigma_{theory} \mid typeof(v) \in \mathcal{S}_{theory}\}$. We assume there is exactly one value for each element of $\mathcal{I}_\iota$ ($\iota \in \mathcal{S}_{theory}$). Elements of $T(\Sigma_{theory}, \mathcal{V})$ are called *theory terms*. For *ground* theory terms, we define $[\![s\ t]\!] = [\![s]\!]([\![t]\!])$. We fix a theory sort bool with $\mathcal{I}_{\mathsf{bool}} = \{\top, \bot\}$. A *constraint* is a theory term $s :: \mathsf{bool}$, such that $typeof(x) \in \mathcal{S}_{theory}$ for all $x \in Var(s)$.

Example 1 In this text we always use $\mathcal{S}_{theory} = \{\mathsf{int}, \mathsf{bool}\}$ and $\Sigma_{theory} = \{+, -, *, <, \leq, >, \geq, =, \wedge, \vee, \neg, \mathtt{true}, \mathtt{false}\} \cup \{\mathsf{n} \mid n \in \mathbb{Z}\}$, with $+, -, * :: \mathsf{int} \to \mathsf{int} \to \mathsf{int}$, $<, \leq, >, \geq, = :: \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$, $\wedge, \vee :: \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$, $\neg :: \mathsf{bool} \to \mathsf{bool}$, $\mathtt{true}, \mathtt{false} :: \mathsf{bool}$ and $\mathsf{n} :: \mathsf{int}$. We let $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$, $\mathcal{I}_{\mathsf{bool}} = \{\top, \bot\}$ and interpret all symbols as expected. The values are $\mathtt{true}, \mathtt{false}$ and all n. Theory terms are for instance $x + 3$, $\mathtt{true}$ and $7 * 0$. The latter two are ground. We have $[\![7 * 0]\!] = 0$. Let $x \in \mathcal{V}$ of type int. Then theory term $x > 0$ is a constraint, but the theory term $(f\ x) > 0$ with $f \in \mathcal{V}$ of type $\mathsf{int} \to \mathsf{int}$ is not (since $typeof(f) \notin \mathcal{S}_{theory}$), nor is $[>]\ 0 :: \mathsf{int} \to \mathsf{bool}$ (since constraints must have type bool).

Substitutions, contexts and subterms A substitution is a type-preserving mapping $\gamma : \mathcal{V} \to T(\Sigma, \mathcal{V})$. The domain of a substitution is defined as $dom(\gamma) = \{x \in \mathcal{V} \mid \gamma(x) \neq x\}$, and the image of a substitution as $im(\gamma) = \{\gamma(x) \mid x \in dom(\gamma)\}$. A substitution on finite domain $\{x_1, \ldots, x_n\}$ is often denoted $[x_1 := s_1, \ldots, x_n := s_n]$. A substitution γ is extended to a function $s \mapsto s\gamma$ on terms by placewise substituting variables in the term by their image: (**i**) $t\gamma = t$ if $t \in \Sigma$, (**ii**) $t\gamma = \gamma(t)$ if $t \in \mathcal{V}$, and (**iii**) $(t_0\ t_1)\gamma = (t_0\gamma)\ (t_1\gamma)$. If $M \subseteq T(\Sigma, \mathcal{V})$ then $\gamma(M)$ denotes the set $\{t\gamma \mid t \in M\}$. A *ground substitution* is a substitution γ such that for all $x \in dom(\gamma)$ of an inhabited type, $\gamma(x)$ is a ground term. A substitution γ *respects* a constraint φ if $\gamma(Var(\varphi)) \subseteq \mathcal{V}al$ and $[\![\varphi\gamma]\!] = \top$. We say that a constraint φ is *satisfiable* if there exists a substitution γ that respects φ, and is *valid* if $[\![\varphi\gamma]\!] = \top$ for all ground substitutions γ that map each $x \in Var(\varphi)$ to values. Let $\Box_1, \ldots, \Box_n$ be fresh, typed constants ($n \geq 1$). A context $C[\Box_1, \ldots, \Box_n]$ (or just: C) is a term in $T(\Sigma \cup \{\Box_1, \ldots, \Box_n\}, \mathcal{V})$ in which each $\Box_i$ occurs exactly once. The term obtained from C by replacing each $\Box_i$ by a term t_i of the same type is denoted by $C[t_1, \ldots, t_n]$. We say that t is a *subterm* of s, notation $s \trianglerighteq t$, if either $s = t$ or $s = a\ s_1 \cdots s_n$ and $s_i \trianglerighteq t$ for some i. We say that t is a *strict* subterm of s, notation $s \triangleright t$, if $s \trianglerighteq t$ and $s \neq t$. (Here we deviate from the typical norm in higher-order rewriting, since we do *not* for instance include s as a subterm of a term $s\ t$. This is deliberate, because we are only interested in "maximally applied" subterms.)

Rewrite rules and reduction relation A rewrite rule is an expression $\ell \to r\ [\varphi]$. Here ℓ and r are terms of the same type, ℓ has a form $\mathsf{f}\ \ell_1 \cdots \ell_k$ with $\mathsf{f} \in \Sigma$ and $k \geq 0$, φ is a constraint and $Var(r) \subseteq Var(\ell) \cup Var(\varphi)$. If $\varphi = \mathtt{true}$, we just write $\ell \to r$. In what follows we fix a signature Σ. We define the set of *calculation rules* as: $\mathcal{R}_{calc} = \{\mathsf{f}\ x_1 \cdots x_m \to y\ [y = \mathsf{f}\ x_1 \cdots x_m] \mid \mathsf{f} \in \Sigma_{theory} \setminus \mathcal{V}al$ with $typeof(\mathsf{f}) = \iota_1 \to \ldots \to \iota_m \to \kappa\}$. We furthermore assume a set of rewrite rules $\mathcal{R}$ satisfying the following properties

- for all $\ell \to r\ [\varphi] \in \mathcal{R}$: ℓ is not a theory term (such rules are contained in $\mathcal{R}_{calc}$)
- for all $\mathsf{f}\ \ell_1 \cdots \ell_k \to r\ [\varphi]$, $\mathsf{g}\ \ell'_1 \cdots \ell'_n \to r'\ [\psi] \in \mathcal{R} \cup \mathcal{R}_{calc}$: if $\mathsf{f} = \mathsf{g}$ then $k = n$

The latter restriction blocks us for instance from having both a rule $\mathsf{append}\ \mathsf{nil} \to \mathsf{id}$ and a rule $\mathsf{append}\ (\mathsf{cons}\ x\ y)\ z \to \mathsf{cons}\ x\ (\mathsf{append}\ y\ z)$. While such rules would normally be allowed in higher-order

rewriting, we need to impose this limitation for the notion of *quasi-reductivity* to make sense, as discussed in [9]. This does not really limit expressivity, since we can use a strategy similar to η-expansion, padding both sides of a rule with variables, e.g., replacing the first rule above by $\mathsf{append}\ \mathsf{nil}\ x \to \mathsf{id}\ x$.

Elements of $\mathcal{D} = \{\mathsf{f} \in \Sigma \mid \exists \mathsf{f}\ \ell_1 \cdots \ell_k \to r\ [\varphi] \in \mathcal{R}\}$ are called *defined symbols*. Elements of $\mathcal{C} = \mathcal{V}al \cup (\Sigma_{terms} \setminus \mathcal{D})$ are called *constructors*. Elements of $\Sigma_{calc} = \Sigma_{theory} \setminus \mathcal{V}al$ are called *calculation symbols*.

For every defined or calculation symbol $\mathsf{f} :: \sigma_1 \to \ldots \to \sigma_m \to \iota$ with $\iota \in \mathcal{S}$, we let $ar(\mathsf{f}) \leq m$ be the number such that for every rule of the form $\mathsf{f}\ \ell_1 \cdots \ell_k \to r\ [\varphi]$ in $\mathcal{R} \cup \mathcal{R}_{calc}$ we have $ar(\mathsf{f}) = k$. (By the restrictions above, this number exists.) For all constructors $\mathsf{f} \in \mathcal{C}$, we define $ar(\mathsf{f}) = \infty$.

The reduction relation $\to_{\mathcal{R}}$ is defined by:

$$C[l\gamma] \to_{\mathcal{R}} C[r\gamma] \text{ if } \ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc} \text{ and } \gamma \text{ respects } \varphi$$

Note that by definition of context, reductions may occur at the head of an application. For example, if $\mathsf{append}\ \mathsf{nil} \to \mathsf{id} \in \mathcal{R}$, then we could reduce $\mathsf{append}\ \mathsf{nil}\ s \to_{\mathcal{R}} \mathsf{id}\ s$.

LCSTRS An LCSTRS is a pair $(T(\Sigma,\mathcal{V}), \to_{\mathcal{R}})$ generated by $(\mathcal{S}, \mathcal{S}_{theory}, \Sigma_{terms}, \Sigma_{theory}, \mathcal{V}, typeof, \mathcal{I}, [\![\cdot]\!], \mathcal{R})$. We often refer to an LCSTRS by $\mathcal{L} = (\Sigma, \mathcal{R})$, or just $\mathcal{R}$, leaving the rest implicit.

We say $\mathcal{L} = (\Sigma, \mathcal{R})$ is *terminating* if there is no infinite reduction sequence $s_0 \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \ldots$ for any $s_0 \in T(\Sigma,\mathcal{V})$. A term s *has normal form* t if $s \to_{\mathcal{R}}^* t$ and t cannot be reduced. We say $\mathcal{L}$ is *weakly normalising* if every term has at least one normal form. Note that termination implies weak normalisation, but not the other way around.

Example 2 Let $\mathcal{R}$ consist of the following rules

$$\textbf{(R1)}\ \mathsf{recdown}\ f\ n\ i\ a \to a \quad [i < n] \qquad \textbf{(R2)}\ \mathsf{recdown}\ f\ n\ i\ a \to f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a) \quad [i \geq n]$$
$$\textbf{(R3)}\ \mathsf{tailup}\ f\ i\ m\ a \to a \quad [i > m] \qquad \textbf{(R4)}\ \mathsf{tailup}\ f\ i\ m\ a \to \mathsf{tailup}\ f\ (i+1)\ m\ (f\ i\ a) \quad [i \leq m]$$

The intuition is that recdown and tailup define recursors that can be used to describe a class of simple real-life programs which compute a return-value using a recursive or tail-recursive procedure. More specifically, we consider programs that use a loop index i, being decreased/increased by 1 during each recursive call, until i reaches a value below lower bound n or above upper bound m. For example, we can represent the following two programs (both computing the factorial function $x \mapsto \prod_{i=1}^{x}$ when restricting to non-negative integers)

```
int factRec(int x){
    if (x >= 1)
        return(x*factRec(x-1));
    else
        return 1; }
```

```
int factTail(int x){
    int a = 1; int i = 1;
    while (i<=x){
        a = i*a; i = i+1;}
    return a; }
```

with recdown and tailup by introducing rewrite rules $\mathsf{factRec}\ x \to \mathsf{recdown}\ [*]\ 1\ x\ 1$ (loop index x and lower bound 1) and $\mathsf{factTail}\ x \to \mathsf{tailup}\ [*]\ 1\ x\ 1$ (loop index 1 and upper bound x).

In general, we can think about $\mathsf{recdown}\ [*]\ n\ i\ a$ to compute $(\prod_{k=n}^{i} k) \cdot a$ and we can think about $\mathsf{tailup}\ [*]\ j\ m\ b$ to compute $(\prod_{k=j}^{m} k) \cdot b$. Hence, all ground instances of $\mathsf{recdown}\ [*]\ n\ i\ a$ and $\mathsf{tailup}\ [*]\ n\ i\ a$ produce the same result. We will prove this with bounded rewriting induction in subsection 3.2; not just for $f = *$, but for arbitrary function $f :: \mathsf{int} \to \mathsf{int} \to \mathsf{int}$.

Considering $\mathcal{R} = \{\textbf{(R1)}, \textbf{(R2)}, \textbf{(R3)}, \textbf{(R4)}\}$ we have $\mathcal{S} = \mathcal{S}_{theory} = \{\mathsf{int}, \mathsf{bool}\}$, $\Sigma_{terms} = \{\mathsf{recdown}, \mathsf{tailup} :: (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}\}$ and Σ_{theory} as in Example 1. Furthermore, $\Sigma_{calc} = \{+, -, *, <, \leq, >, \geq, =, \wedge, \vee\}$, $\mathcal{D} = \Sigma_{terms}$ and $\mathcal{C} = \mathcal{V}al = \{\mathtt{true}, \mathtt{false}\} \cup \{\mathsf{n} \mid n \in \mathbb{Z}\}$. Substitution $\gamma = [i := 1,\ n := 0]$ induces a reduction $\mathsf{recdown}\ f\ 0\ 1\ a \to_{\mathcal{R}} f\ 1\ (\mathsf{recdown}\ f\ 0\ (1-1)\ a) \to_{\mathcal{R}_{calc}}$

$f\ 1\ (\mathsf{recdown}\ f\ 0\ 0\ a) \to_{\mathcal{R}} f\ 1\ (f\ 0\ (\mathsf{recdown}\ f\ 0\ (0-1)\ a)) \to_{\mathcal{R}_{calc}} f\ 1\ (f\ 0\ (\mathsf{recdown}\ f\ 0\ (-1)\ a)) \to_{\mathcal{R}} f\ 1\ (f\ 0\ a)$. It is easy to check that $(\mathsf{tailup}\ f\ n\ i\ a)\gamma = \mathsf{tailup}\ f\ 0\ 1\ a$ also reduces to $f\ 1\ (f\ 0\ a)$.

We will limit our interest to *quasi-reductive* LCSTRSs (defined below), which is needed to guarantee correctness of RI. Intuitively, this property expresses that pattern matching on ground terms is exhaustive (i.e. there are no missing reduction cases). For example, the rewrite system $\mathcal{R} = \{\mathbf{(R1)}, \mathbf{(R2)}\}$ is quasi-reductive because $i < n$ and $i \geq n$ together cover all ground instances of $\mathsf{recdown}\ f\ n\ i\ a$. But if we, for example, replace **(R2)** by $\mathsf{recdown}\ f\ n\ i\ a \to f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a)\ [i > n]$ then it is not, as we are missing all ground reduction cases for $i = n$ (for example $\mathsf{recdown}\ [*]\ 0\ 0\ 0$ does not reduce anymore).

For first-order the LCTRSs, quasi-reductivity is achieved by demanding that there are no other ground normal forms than the ground constructor terms $T(\mathcal{C}, \emptyset)$. For higher-order LCSTRSs, however, this approach does not work as we can have ground normal forms with partially applied defined symbols (for example, $\mathsf{recdown}\ [+]$). Hence, the notion of constructor terms is generalized to the higher-order setting.

Quasi-reductivity Let $\mathcal{L} = (\Sigma, \mathcal{R})$ be some LCSTRS. The *semi-constructor terms* over $\mathcal{L}$, notation $\mathcal{SCT}_{\mathcal{L}}$, are defined by **(i)**. $\mathcal{V} \subseteq \mathcal{SCT}_{\mathcal{L}}$, **(ii)**. if $\mathsf{f} \in \Sigma$ with $\mathsf{f} :: \sigma_1 \to \ldots \to \sigma_m \to \iota$, $\iota \in \mathcal{S}$ and $s_1 :: \sigma_1, \ldots, s_n :: \sigma_n \in \mathcal{SCT}_{\mathcal{L}}$ with $n \leq m$, then $\mathsf{f}\ s_1 \cdots s_n \in \mathcal{SCT}_{\mathcal{L}}$ if $n < ar(\mathsf{f})$.

Semi-constructor terms are always normal forms. Furthermore, as $ar(\mathsf{f}) = \infty$ for $\mathsf{f} \in \mathcal{C}$, the constructor terms $T(\mathcal{C}, \mathcal{V})$ are contained in $\mathcal{SCT}_{\mathcal{L}}$. *Ground* semi-constructor terms $\mathcal{SCT}^{\emptyset}_{\mathcal{L}}$ are the terms built without (i). A *ground semi-constructor substitution* (*gsc substitution*) is a substitution such that $im(\gamma) \subseteq \mathcal{SCT}^{\emptyset}_{\mathcal{L}}$.

$\mathcal{L}$ is quasi-reductive if for every $t \in T(\Sigma, \emptyset)$ we have $t \in \mathcal{SCT}^{\emptyset}_{\mathcal{L}}$ or t reduces with $\to_{\mathcal{R}}$. Put differently, the only ground normal forms are semi-constructor terms. Weak normalization and quasi-reductivity together ensure that every ground term reduces to a semi-constructor term. Note that, if $s_1, \ldots, s_n$ are ground normal forms and $\mathsf{f} \in \Sigma$, then $\mathsf{f}\ s_1 \cdots s_n$ is a ground normal form if and only if $n < ar(\mathsf{f})$.

Equations and inductive theorems An *equation* is a triple $s \approx t\ [\varphi]$ with $typeof(s) = typeof(t)$ and φ a constraint, such that all variables in $Var(s) \cup Var(t) \cup Var(\varphi)$ have an inhabited type. If φ equals `true`, we will simply write the equation as $s \approx t$. A substitution γ respects $s \approx t\ [\varphi]$ if γ respects φ. An equation $s \approx t\ [\varphi]$ is an *inductive theorem* (aka *ground convertible*) if $s\gamma \leftrightarrow^*_{\mathcal{R}} t\gamma$ for every ground substitution γ that respects φ. Here $\leftrightarrow_{\mathcal{R}} = \to_{\mathcal{R}} \cup \leftarrow_{\mathcal{R}}$, and $\leftrightarrow^*_{\mathcal{R}}$ is its transitive, reflexive closure.

Example 3 The LCSTRS from Example 2 admits an equation $\mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a$. Since it has constraint `true`, any substitution respects it. In subsection 3.2 we will prove that this equation is an inductive theorem, meaning that $(\mathsf{recdown}\ f\ n\ i\ a)\gamma \leftrightarrow^*_{\mathcal{R}} (\mathsf{tailup}\ f\ n\ i\ a)\gamma$ for any ground substitution γ.

3 Rewriting Induction

RI was introduced [10] as a deduction system for proving inductive theorems, using unconstrained first-order term rewriting systems. Since then, many variations on the system have appeared (e.g., [1, 2, 3, 5, 6, 9]), including a version for LCSTRSs. All are based on well-founded induction, using some well-founded relation $\succ$. Some [10, 5, 6, 9] use a fixed strategy to construct a terminating rewrite system $A \supseteq \mathcal{R}$ and then choose $\succ = \to^+_A$. However, as explained in the introduction, this approach leads to heavy termination requirements, because these strategies include all induction hypothesis into A.

To improve on this, some work has been done [1, 2, 3] employing a well-founded relation $\succ$ that satisfies certain requirements (like monotonicity and stability, but also ground totality). This relation may either be fixed beforehand (e.g., the lexicographic path ordering), or constructed during or after the

proof, as the proof process essentially accumulates termination requirements. The RI system is designed to keep termination requirements as mild as possible, for example by allowing reduction steps with an induction hypothesis to be oriented using a second relation $\succeq$ rather than the default $\succ$. However, this approach also imposes more bureaucracy, since derivation rules rely on several steps being done at once – for example, by reasoning *modulo* the set of induction hypotheses. This makes it quite hard to use especially when the relation $\succ$ is not fixed beforehand but rather constructed on the fly.

Here, we aim to combine the best of both worlds. We try to reduce termination requirements using a pair $(\succ,\succeq)$, which may either be fixed in advance, or constructed as part of the proof process. Importantly, we do not impose the ground totality requirement (which would be extremely restrictive in higher-order rewriting!), and thus allow for $\succ$ to for instance be a relation $(\to_A \cup \rhd)^+$, or a construction based on dependency pairs. We avoid the bureaucracy of combining steps by introducing the notion of an equation *context*, which keeps track of an extra pair of terms to be used for ordering requirements.

3.1 Equation contexts and proof states

RI is a deduction system on proof states, which are pairs of the shape $(\mathcal{E},\mathcal{H})$. Intuitively (and following the existing literature), $\mathcal{E}$ is a set of equations, describing all proof goals, and $\mathcal{H}$ is the set of induction hypotheses that have been assumed. At the start $\mathcal{E}$ consists of all equations that we want to prove to be inductive theorems, and $\mathcal{H}=\emptyset$. With a deduction rule we may transform a proof state $(\mathcal{E},\mathcal{H})$ into another proof state $(\mathcal{E}',\mathcal{H}')$. This is denoted as $(\mathcal{E},\mathcal{H}) \vdash (\mathcal{E}',\mathcal{H}')$. We write $\vdash^*$ for the reflexive, transitive closure of $\vdash$. Correctness of RI is guaranteed by the following principle: "If $(\mathcal{E},\mathcal{H}) \vdash^* (\emptyset,\mathcal{H})$ for some set $\mathcal{H}$, then every equation in $\mathcal{E}$ is an inductive theorem" [6, 9]. Intuitively, this reads as: if we can remove every proof obligation (making $\mathcal{E}$ empty) then every equation in $\mathcal{E}$ is an inductive theorem.

In Bounded RI, we will deviate from this setting in one respect: instead of letting $\mathcal{E}$ be a set of equations, we will use a set of *equation contexts*.

3.2 Bounded Rewriting Induction

We will now introduce *bounded rewriting induction*, considering proof states containing only *bounded equation contexts*. For this, we assume a bounding pair:

Definition 1 (Bounding Pair) A *bounding pair* for an LCSTRS $\mathcal{L}=(\Sigma,\mathcal{R})$ is a pair $(\succ,\succeq)$ with $\succ$ a well-founded partial ordering on $T(\Sigma,\emptyset)$ (that is, $\succ$ is a transitive, anti-symmetric, irreflexive and well-founded relation) and $\succeq$ a quasi-order on $T(\Sigma,\emptyset)$ (that is, $\succeq$ is a transitive and reflexive relation) such that $\succ \subseteq \succeq$, $\succ \cdot \succeq \subseteq \succ$, $\succeq \cdot \succ \subseteq \succ$ and such that $s \succeq t$ whenever $s \to_{\mathcal{R}} t$ or $s \rhd t$.

A bounding pair is extended to non-ground terms with constraint: we define $s \succ t \ [\psi]$ iff $s\gamma \succ t\gamma$ for all ground substitutions γ that respect ψ. ($s \succeq t \ [\psi]$ is defined similarly)

If $\mathcal{L}$ is terminating we can choose $\succ = (\to_{\mathcal{R}} \cup \rhd)^+$ and $\succeq$ the reflexive closure of $\succ$. But there are other ways to choose a bounding pair, for example monotonic algebras or recursive path orderings.

Definition 2 (Equation context) Let $\bullet$ be a fresh symbol. We define $\bullet \succ s$ and $\bullet \succeq s$ for all $s \in T(\Sigma,\mathcal{V})$, and also $\bullet \succeq \bullet$. An *equation context* $(\varsigma \ ; \ s \approx t \ ; \ \tau) \ [\psi]$ is a tuple of two elements $\varsigma,\tau \in T(\Sigma,\mathcal{V}) \cup \{\bullet\}$, two terms s,t and a constraint ψ. We write $(\varsigma \ ; \ s \simeq t \ ; \ \tau) \ [\psi]$ (so with $\simeq$ instead of $\approx$) to denote either an equation context $(\varsigma \ ; \ s \approx t \ ; \ \tau) \ [\psi]$ or an equation context $(\tau \ ; \ t \approx s \ ; \ \varsigma) \ [\psi]$.

A *bounded equation context* is an equation context such that both $\varsigma \succeq s \ [\psi]$ and $\tau \succeq t \ [\psi]$. A substitution γ *respects* an equation context $(\varsigma \ ; \ s \approx t \ ; \ \tau) \ [\psi]$ if γ respects ψ.

An equation context couples an equation with a bound on the induction: we implicitly work with the induction hypothesis "all ground instances of an equation in $\mathcal{H}$ that are strictly smaller than the current instance of $\varsigma \approx \tau\ [\psi]$ are convertible". For example, in an equivalence proof of two implementations of the factorial function, we may encounter induction hypothesis $\mathsf{fact}_1\ n \approx \mathsf{fact}_2\ n\ [n \geq 0]$, and equation context $(\mathsf{fact}_1\ n\ ;\ \mathsf{fact}_1\ k \approx \mathsf{fact}_2\ k\ ;\ \mathsf{fact}_2\ n)\ [n > 0 \wedge n = k+1]$. We can apply the instance $\mathsf{fact}_1\ k \approx \mathsf{fact}_2\ k\ [k \geq 0]$ of the induction hypothesis to $\mathsf{fact}_1\ k \approx \mathsf{fact}_2\ k\ [n > 0 \wedge n = k+1]$ because

(**1**). $n > 0 \wedge n = k+1$ implies $k \geq 0$, and (**2**). both $\mathsf{fact}_1\ n \succ \mathsf{fact}_1\ k\ [n > 0 \wedge n = k+1]$ and $\mathsf{fact}_2\ n \succ \mathsf{fact}_2\ k\ [n > 0 \wedge n = k+1]$ hold for an appropriately chosen $\succ$.

Definition 3 (Proof state) $(\mathcal{E}, \mathcal{H})$ is a proof state if $\mathcal{E}$ a set of equation contexts and $\mathcal{H}$ a set of equations.

From Definition 2 we can see that $\bullet$ behaves as an infinity term with respect to $\succ$ and $\succeq$. As expressed by Theorem 1: when using bounded RI to prove a set of equations, we pour them into a set $\mathcal{E}$ of equation contexts using infinite bounds $\varsigma = \tau = \bullet$. This is not a problem, because we always start with the proof state $(\mathcal{E}, \emptyset)$, so there are no induction hypothesis available yet. As soon we add an induction hypothesis to the proof state, the bounds are correctly getting lowered, as dictated by Figure 1.(Induct).

Theorem 1 (Correctness of Bounded RI) *Let $\mathcal{L}$ be a weakly normalizing, quasi-reductive LCSTRS; let $\mathcal{A}$ be a set of equations; and let $\mathcal{E}$ be the set of equation contexts $\{(\bullet\ ;\ s \approx t\ ;\ \bullet)\ [\psi] \mid s \approx t\ [\psi] \in \mathcal{A}\}$. Let $(\succ, \succeq)$ be some bounding pair, such that $(\mathcal{E}, \emptyset) \vdash^* (\emptyset, \mathcal{H})$, for some $\mathcal{H}$ using the derivation rules in Figure 1. Then every equation in $\mathcal{A}$ is an inductive theorem.*

The deduction rules for bounded rewriting induction are provided in Figure 1, and explained in detail below via a running example. This figure uses one particular new notation $\psi \models^{\delta} \varphi$, defined as follows:

Definition 4 ($\models^{\delta}$) Let δ be a substitution and φ, ψ be constraints. We write $\psi \models^{\delta} \varphi$ if $\delta(Var(\varphi)) \subseteq \mathcal{V}al \cup Var(\psi)$, and $\psi \Longrightarrow \varphi\delta$ is a valid constraint.

We will now elaborate on the rules of Figure 1, and illustrate their use through examples. To start, we will use the LCSTRS from Example 2 applied on the equation $\mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a$. Following Theorem 1, we will show that there is a set $\mathcal{H}$ such that

$$(\mathcal{E}_1, \emptyset) \vdash^* (\emptyset, \mathcal{H}) \text{ with } \mathcal{E}_1 := \{(\bullet\ ;\ \mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a\ ;\ \bullet)\ [\mathtt{true}]\}$$

We use the proof process to accumulate requirements on $\succ$ to be used, but precommit to a bounding pair such that $\succeq$ is the reflexive closure of $\succ$. We also assume that $s \succ t$ whenever $s \to_{\mathcal{R}} t$ or $s \rhd t$. To guarantee a well-defined proof system on bounded equation contexts we should demonstrate (which we will not do here) that all deduction rules preserve the following property

$$\begin{aligned}(\star\star):\quad & \text{For every equation context } (\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi] \\ & \text{either } \varsigma = s \text{ or } \varsigma \succ s\ [\psi], \text{ and also either } \tau = t \text{ or } \tau \succ t\ [\psi].\end{aligned}$$

(Induct) This deduction rule starts an induction proof. From a proof-technical point of view two things happen. First, and most importantly for the proof progress, the current equation is added to $\mathcal{H}$, making it available for later application of (Hypothesis) or ($\mathcal{H}$-Delete). Second, the bounding terms ς, τ are replaced by s, t. This ensures that, when an induction hypothesis $s \approx t\ [\psi]$ is applied, it is only on equations that are strictly smaller than $s \approx t\ [\psi]$.

In our running example, we use (Induct) to obtain $(\mathcal{E}_1, \emptyset) \vdash (\mathcal{E}_2, \mathcal{H}_2)$ where

$$\mathcal{E}_2 = \{(\varsigma_2\ ;\ \mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a\ ;\ \tau_2)\ [\mathtt{true}]\} \quad \mathcal{H}_2 = \{\mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a\}$$

We recall $\varsigma_2 = \mathsf{recdown}\ f\ n\ i\ a$ and $\tau_2 = \mathsf{tailup}\ f\ n\ i\ a$ for later usage in the RI process.

Figure 1: Derivation rules for bounded rewriting induction, given a bounding pair $(\succ, \succeq)$.

(Simplify)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \;;\; C[\ell\delta] \simeq t \;;\; \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(\varsigma \;;\; C[r\delta] \approx t \;;\; \tau)\ [\psi]\}, \mathcal{H})} \quad \ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc} \text{ and } \psi \models^{\delta} \varphi$$

(Case)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \;;\; s \approx t \;;\; \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(\varsigma\delta \;;\; s\delta \approx t\delta \;;\; \tau\delta)\ [\psi\delta \wedge \varphi] \mid (\delta, \varphi) \in \mathcal{C}\}, \mathcal{H})} \quad \mathcal{C} \text{ a cover set of } s \approx t\ [\psi] \text{ (see Definition 5)}$$

(Delete)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \;;\; s \approx t \;;\; \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E}, \mathcal{H})} \quad \psi \text{ unsatisfiable, or } s = t$$

(Semi-constructor)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \;;\; f\ s_1 \cdots s_n \approx f\ t_1 \cdots t_n \;;\; \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(\varsigma \;;\; s_i \approx t_i \;;\; \tau)\ [\psi] \mid 1 \leq i \leq n\}, \mathcal{H})} \quad n > 0 \text{ and } (f \in \mathcal{V} \text{ or } n < ar(f))$$

(Induct)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \;;\; s \approx t \;;\; \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(s \;;\; s \approx t \;;\; t)\ [\psi]\}, \mathcal{H} \cup \{s \approx t\ [\psi]\})}$$

(Hypothesis)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \;;\; C[\ell\delta] \simeq t \;;\; \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(\varsigma \;;\; C[r\delta] \approx t \;;\; \tau)\ [\psi]\}, \mathcal{H})} \quad \begin{array}{l}\ell \simeq r\ [\varphi] \in \mathcal{H} \text{ and } \psi \models^{\delta} \varphi \text{ and} \\ \varsigma \succ \ell\delta\ [\psi] \text{ and } \varsigma \succ r\delta\ [\psi] \text{ and } \varsigma \succeq C[r\delta]\ [\psi]\end{array}$$

($\mathcal{H}$-Delete)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \;;\; C[\ell\delta] \simeq C[r\delta] \;;\; \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E}, \mathcal{H})} \quad \begin{array}{l}\ell \simeq r\ [\varphi] \in \mathcal{H} \text{ and } \psi \models^{\delta} \varphi \text{ and} \\ \varsigma \succ \ell\delta\ [\psi] \text{ or } \tau \succ r\delta\ [\psi]\end{array}$$

(Generalize)/(Alter)

$$\frac{(\mathcal{E} \uplus \{(\varsigma \;;\; s \approx t \;;\; \tau)\ [\psi]\}, \mathcal{H})}{(\mathcal{E} \cup \{(\varsigma' \;;\; s' \approx t' \;;\; \tau')\ [\varphi]\}, \mathcal{H})} \quad \begin{array}{l}(\varsigma' \;;\; s' \approx t' \;;\; \tau')\ [\varphi] \text{ generalizes/alters } (\varsigma \;;\; s \approx t \;;\; \tau)\ [\psi] \\ \text{(see Definition 6), and } \varsigma' \succeq s'\ [\varphi] \text{ and } \tau' \succeq t'\ [\varphi]\end{array}$$

(Postulate)

$$\frac{(\mathcal{E}, \mathcal{H})}{(\mathcal{E} \cup \{(\bullet \;;\; s \approx t \;;\; \bullet)\ [\psi]\}, \mathcal{H})}$$

(Case) Comparing $\mathcal{E}_2$ to $\mathcal{R}$, which of the rules should we apply? As we will see in (Simplify), this requires information about how the variables i,n in the equation are instantiated, since we have to distinguish between the cases $i < n$ and $i \geq n$. This is where (Case) can help us, splitting an equation into multiple cases. Of course, we have to make sure that the cases together cover the original equation.

Definition 5 (Cover set) A cover set of $s \approx t\ [\psi]$ is a set $\mathcal{C}$ of pairs (δ, φ), with δ a substitution and φ a constraint, such that for every gsc substitution γ respecting $s \approx t\ [\psi]$, there exists $(\delta, \varphi) \in \mathcal{C}$ such that $s\gamma \approx t\gamma\ [\psi\gamma]$ is an instance of $s\delta \approx t\delta\ [\psi\delta \wedge \varphi]$. (That is, there is a substitution σ that respects $\psi\delta \wedge \varphi$ such that $s\delta\sigma = s\gamma$ and $t\delta\sigma = t\gamma$.)

Continuing our example: the only gsc terms of type int are values. Hence, $\mathcal{C} = \{([], i < n),\ ([], i \geq n)\}$ is a cover set of $\mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a$. Using (Case), we obtain $(\mathcal{E}_2, \mathcal{H}_2) \vdash (\mathcal{E}_3, \mathcal{H}_2)$ with $\mathcal{E}_3 =$

$$\big\{(\varsigma_2\ ;\ \mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a\ ;\ \tau_2)\ [i < n],\ (\varsigma_2\ ;\ \mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a\ ;\ \tau_2)\ [i \geq n]\big\}$$

The bounding terms ς_2, τ_2 are unchanged because the substitutions in the cover set are both empty.

(Simplify) With (Simplify) we use a rule $\ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc}$ to rewrite an equation $C[\ell\delta] \simeq t\ [\psi]$. The requirement $\psi \models^{\delta} \varphi$ makes sure that the δ-instance of $\ell \to r\ [\varphi]$ is actually applicable. The bounding terms are not affected by the reduction.

Continuing our example, the first equation in $\mathcal{E}_3$ has constraint $i < n$, so we apply (Simplify) on both sides of this equation, using **(R1)** and **(R3)**. For the second equation, we also apply (Simplify) to both sides, using **(R2)** and **(R4)**. We obtain $(\mathcal{E}_3, \mathcal{H}_2) \vdash^* (\mathcal{E}_4, \mathcal{H}_2)$ with

$$\mathcal{E}_4 = \left\{ \begin{array}{ll} (\varsigma_2\ ;\ a \approx a\ ;\ \tau_2) & [i < n] \\ (\varsigma_2\ ;\ f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a) \approx \mathsf{tailup}\ f\ (n+1)\ i\ (f\ n\ a)\ ;\ \tau_2) & [i \geq n] \end{array} \right\}$$

(Delete) This deduction rule allows us to remove an equation that has an unsatisfiable constraint, or whose two sides are syntactically equal. In our example, we use (Delete) and obtain $(\mathcal{E}_4, \mathcal{H}_2) \vdash (\mathcal{E}_5, \mathcal{H}_2)$

$$\mathcal{E}_5 = \{(\varsigma_2\ ;\ f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a) \approx \mathsf{tailup}\ f\ (n+1)\ i\ (f\ n\ a)\ ;\ \tau_2)\ [i \geq n]\}$$

(Alter) If is often useful to rewrite an equation (context) to another that might be syntactically different, but has the same ground instances (or at least: the same ground semi-constructor instances). Indeed, this may even be necessary, for instance to support the application of a rewrite rule through (Simplify).

Definition 6 We say that an equation context $(\varsigma'\ ;\ s' \approx t'\ ;\ \tau')\ [\varphi]$ *generalizes* $(\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi]$ if for every gsc substitution γ that respects $(\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi]$ there is a substitution δ that respects $(\varsigma'\ ;\ s' \approx t'\ ;\ \tau')\ [\varphi]$ such that $s\gamma = s'\delta$ and $t\gamma = t'\delta$, and $\varsigma\gamma \succeq \varsigma'\delta$ and $\tau\gamma \succeq \tau'\delta$. It *alters* $(\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi]$ if both $(\varsigma'\ ;\ s' \approx t'\ ;\ \tau')\ [\varphi]$ generalizes $(\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi]$, and $(\varsigma\ ;\ s \approx t\ ;\ \tau)\ [\psi]$ generalizes $(\varsigma'\ ;\ s' \approx t'\ ;\ \tau')\ [\varphi]$.

There are many ways to use the (Alter) rule, but following the discussion in [9], we will particularly apply it in two ways: **(i)**. Replacing a constraint by an equi-satisfiable one **(ii)**. Replacing variables by equivalent variables or values.

Continuing our example, we apply (Alter) with case (i) to obtain $(\mathcal{E}_5, \mathcal{H}_2) \vdash (\mathcal{E}_6, \mathcal{H}_2)$, with

$$\mathcal{E}_6 = \big\{(\varsigma_2\ ;\ f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a) \approx \mathsf{tailup}\ f\ (n+1)\ i\ (f\ n\ a)\ ;\ \tau_2)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n]\big\}$$

To allow this rule to be applied, we must have $\varsigma_2 \succeq f\ i\ (\mathsf{recdown}\ f\ n\ (i-1)\ a)\ [\varphi]$ and $\tau_2 \succeq \mathsf{tailup}\ f\ (n+1)\ i\ (f\ n\ a)\ [\varphi]$ where φ is the constraint $i' = i-1 \wedge n' = n+1 \wedge i \geq n$. But this follows immediately from $(\star\star)$: if $\varsigma \succ s\ [i \geq n]$ then also $\varsigma \succ s\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n]$, and similar for $\tau \succ t\ [i \geq n]$.

Our previous (Alter) step allows us to continue on the example by two successive (Simplify) steps, using calculation rules $i-1 \to i'\ [i' = i-1]$ and $n+1 \to n'\ [n' = n+1]$, to obtain $(\mathcal{E}_7, \mathcal{H}_2)$, with

$$\mathcal{E}_7 = \left\{ (\varsigma_2\ ;\ f\ i\ (\mathsf{recdown}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ ;\ \tau_2)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \right\}$$

(Hypothesis) Similar to (Simplify), we can use an induction hypothesis to reduce either side of an equation. Here, finally, the bounding terms ς, τ come into play, as we need to make sure that we have a decrease of some kind, to apply induction.

We apply (Hypothesis) on the lhs of $\mathcal{E}_7$ with the induction hypothesis from $\mathcal{H}_2$ in the direction $\mathsf{recdown}\ f\ n\ i\ a \ \to\ \mathsf{tailup}\ f\ n\ i\ a$, with substitution $[i := i']$. We obtain $(\mathcal{E}_7, \mathcal{H}_2) \vdash (\mathcal{E}_8, \mathcal{H}_2)$ with

$$\mathcal{E}_8 = \left\{ (\varsigma_2\ ;\ f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ ;\ \tau_2) \quad [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \right\}$$

To be allowed to apply this deduction rule, we must show that the following $\succ$ requirements are satisfied:

$$\begin{array}{rlll} & \mathsf{recdown}\ f\ n\ i\ a & \succ & \mathsf{recdown}\ f\ n\ i'\ a & [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \\ & \mathsf{recdown}\ f\ n\ i\ a & \succ & \mathsf{tailup}\ f\ n\ i'\ a & [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \\ \textbf{(REQ1)} & \mathsf{recdown}\ f\ n\ i\ a & \succeq & f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) & [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \end{array}$$

The first of these is satisfied by property $(\star\star)$. The second is an immediate consequence of the third, since $\mathsf{f}\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \rhd \mathsf{tailup}\ f\ n\ i'\ a$ and we have committed to let $\rhd$ be included in $\succ$. For the third, we remember that **(REQ1)** still needs to be satisfied. Since we have set $\succeq$ as the reflexive closure of $\succ$, this property is only satisfied if $\mathsf{recdown}\ f\ n\ i\ a \succ f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n]$.

Let $\varsigma_9 = f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a)$ and $\tau_9 = \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)$. We apply (Induct) to $(\mathcal{E}_8, \mathcal{H}_2)$ and obtain

$$\begin{aligned} \mathcal{E}_9 &= \left\{ (\varsigma_9\ ;\ f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ ;\ \tau_9)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \right\} \\ \mathcal{H}_9 &= \mathcal{H}_2 \cup \left\{ f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \right\} \end{aligned}$$

Next, we use (Case) once more, splitting up the constraint $\mathcal{E}_9$ into $i = n$ and $i > n$, giving $(\mathcal{E}_{10}, \mathcal{H}_9)$:

$$\mathcal{E}_{10} = \left\{ \begin{array}{l} (\varsigma_9\ ;\ f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ ;\ \tau_9)\ [i' = i-1 \wedge n' = n+1 \wedge i = n] \\ (\varsigma_9\ ;\ f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \approx \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a)\ ;\ \tau_9)\ [i' = i-1 \wedge n' = n+1 \wedge i > n] \end{array} \right\}$$

Observing that $i' = i-1 \wedge n' = n+1 \wedge i = n$ implies both $n > i'$ and $n' > i$, and that $i' = i-1 \wedge n' = n+1 \wedge i > n$ implies both $n \leq i'$ and $n' \leq i$, we use (Simplify) on both sides of the first equation with **(R3)** and on both sides of the second equation with **(R4)** respectively, to deduce $(\mathcal{E}_{10}, \mathcal{H}_9) \vdash^* (\mathcal{E}_{11}, \mathcal{H}_9)$:

$$\mathcal{E}_{11} = \left\{ \begin{array}{l} (\varsigma_9\ ;\ f\ i\ a \approx f\ n\ a\ ;\ \tau_9) \qquad\qquad [i' = i-1 \wedge n' = n+1 \wedge i = n] \\ (\varsigma_9\ ;\ f\ i\ (\mathsf{tailup}\ f\ (n+1)\ i'\ (f\ n\ a)) \approx \mathsf{tailup}\ f\ (n'+1)\ i\ (f\ n'\ (f\ n\ a))\ ;\ \tau_9) \\ \qquad\qquad [i' = i-1 \wedge n' = n+1 \wedge i > n] \end{array} \right\}$$

The first equation above does not satisfy the requirements for (Delete), even though the $i = n$ part of the constraint makes it look very delete-worthy. With (Alter) (case (ii)), we replace the first equation context by $(\varsigma_9\ ;\ f\ n\ a \approx f\ n\ a\ ;\ \tau_9)\ [i' = i-1 \wedge n' = n+1 \wedge i = n]$, which may immediately be deleted. We also use (Alter) (now case (i)) on the second equation, succeeded by (Simplify). This yields $(\mathcal{E}_{12}, \mathcal{H}_9)$ with

$$\mathcal{E}_{12} = \left\{ \begin{array}{l} (\varsigma_9\ ;\ f\ i\ (\mathsf{tailup}\ f\ n'\ i'\ (f\ n\ a)) \approx \mathsf{tailup}\ f\ n''\ i\ (f\ n'\ (f\ n\ a))\ ;\ \tau_9) \\ \qquad\qquad [i' = i-1 \wedge n' = n+1 \wedge n'' = n'+1 \wedge i > n] \end{array} \right\}$$

($\mathcal{H}$-Delete). With this deduction rule we may rewrite an equation with an instance of equation in $\mathcal{H}$.

In our example, consider the second equation in $\mathcal{H}_9$. Let $\delta = [n := n', n' := n'', a := f\ n\ a]$. Using ($\mathcal{H}$-Delete), we can deduce $(\mathcal{E}_{12}, \mathcal{H}_9) \vdash (\emptyset, \mathcal{H}_9)$ if one of the following ordering requirements are satisfied:

$$\begin{array}{rll} \varsigma_9 = & f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a) \succ f\ i\ (\mathsf{tailup}\ f\ n'\ i'\ (f\ n\ a)) & [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \\ \tau_9 = & \mathsf{tailup}\ f\ n'\ i\ (f\ n\ a) \succ \mathsf{tailup}\ f\ n''\ i\ (f\ n'\ (f\ n\ a)) & [i' = i-1 \wedge n' = n+1 \wedge i \geq n] \end{array}$$

We obtained $(\mathcal{E}_1, \emptyset) \vdash^* (\emptyset, \mathcal{H}_9)$. By Theorem 1 $\mathsf{recdown}\ f\ n\ i\ a \approx \mathsf{tailup}\ f\ n\ i\ a$ is an inductive theorem – provided we have a suitable bounding pair that satisfies **(REQ1)**. But this is easily achieved: let $\succ$ equal $(\to_{\mathcal{R}\cup\mathcal{Q}} \cup \rhd)^+$ where $\mathcal{Q} = \{\mathsf{recdown}\ f\ n\ i\ a \to f\ i\ (\mathsf{tailup}\ f\ n\ i'\ a)\ [i' = i-1 \wedge n' = n+1 \wedge i \geq n]\}$.

It is easy to see that this is indeed a bounding pair if $\to_{\mathcal{R}\cup\mathcal{Q}}$ is terminating. Termination can for instance be proved using static dependency pairs [7].

Remark 1 The choice to let $\succ$ be a relation $(\to_{\mathcal{R}\cup\mathcal{Q}} \cup \rhd)^+$ is quite natural: in traditional definitions of rewriting induction [10, 6, 9] this is the only choice for $(\succ, \succeq)$, with $\mathcal{Q}$ always being a directed version of the last $\mathcal{H}$ (so in the case of this example, $\mathcal{H}_9$). However, while such a choice is natural in strategies for rewriting induction, we leave it open in the definition to allow for alternative orderings.

4 Closing remarks

Two deduction rules we did not demonstrate are (Generalize) and (Postulate). Although (Generalize) appears to be very similar to (Alter) (in fact, every step that can be done by (Alter) can also be done by (Generalize)), they are used quite differently: (Alter) is designed to set up an equation for the use of simplification or deletion, while (Generalize) and (Postulate) are a way to perform of lemma generation.

Lemma generation is often needed in practice to obtain a successful RI proof. This was not visible in the running example in subsection 3.2, where we could for example continue on the equation in $\mathcal{E}_7$ by applying the hypothesis in $\mathcal{H}_2$, which was automatically generated by (Induct), in an (Hypothesis)-step. However, in many practical situations the hypothesis generated by (Induct) is not applicable, and we first have to use (Generalize) to introduce a more general equation, suitable to save as hypothesis for later usage in (Hypothesis) or ($\mathcal{H}$-Delete). How to find such generalizations automatically is a separate topic, and beyond the scope of this paper. The idea of (Postulate) is similar to (Generalize), but rather than replacing the original equation by a generalized equation, we add the generalized equation as a new equation and apply (Induct) on this new equation to obtain the required induction hypothesis.

We have not demonstrated (Semi-constructor) either. With this deduction rule we can split up an equation that contains a constructor or partially applied function symbols. For example, we can split up $\mathsf{foldl}\ \mathsf{g}\ (\mathsf{h}\ \mathsf{0}\ x) \approx \mathsf{foldl}\ \mathsf{h}\ (\mathsf{g}\ \mathsf{0}\ x)$ into two equations: $\mathsf{g} \approx \mathsf{h}$ and $\mathsf{h}\ \mathsf{0}\ x \approx \mathsf{g}\ \mathsf{0}\ x$.

Implementation and work in progress A basic version of Bounded RI for LCSTRSs has been implemented in Cora (available on `https://github.com/hezzel/cora`).

Considering work in progress, we are currently working on RI as a method for proving ground confluence in LCSTRSs. This builds on the work of [4], where the authors showed that this is possible for first-order unconstrained rewriting. Ground confluence is of relevance for completeness because for ground confluent LCSTRSs we can extend RI with a new deduction rule, in order to disprove equations to be inductive theorems. For first-order constrained rewriting this has been shown in [6], and we want to generalize this result to RI for LCSTRSs.

References

[1] T. Aoto (2006): *Dealing with Non-orientable Equations in Rewriting Induction*. In: *Proc. RTA 06*, *Lecture Notes in Computer Science* 4098, pp. 242–256, doi:10.1007/11805618_18. Available at `https://doi.org/10.1007/11805618_18`.

[2] T. Aoto (2008): *Designing a rewriting induction prover with an increased capability of non-orientable theorems*. In: *Proc. SCSS 08*.

[3] T. Aoto (2008): *Soundness of Rewriting Induction Based on an Abstract Principle*. *Inf. Media Technol.* 3(2), pp. 225–235, doi:10.11185/IMT.3.225. Available at `https://doi.org/10.11185/imt.3.225`.

[4] T. Aoto & Y. Toyama (2016): *Ground Confluence Prover based on Rewriting Induction*. In: *Proc. FSCD 16*, *LIPIcs* 52, pp. 33:1–33:12, doi:10.4230/LIPICS.FSCD.2016.33. Available at `https://doi.org/10.4230/LIPIcs.FSCD.2016.33`.

[5] S. Falke & D. Kapur (2012): *Rewriting Induction + Linear Arithmetic = Decision Procedure*. In: *Proc. IJCAR 12*, *LNAI* 7364, pp. 241–255, doi:10.1007/978-3-642-31365-3_20.

[6] C. Fuhs, C. Kop & N. Nishida (2017): *Verifying Procedural Programs via Constrained Rewriting Induction*. *ACM Transactions On Computational Logic (TOCL)* 18(2), pp. 14:1–14:50, doi:10.1145/3060143.

[7] L. Guo, K. Hagens, C Kop & D. Vale (2024): *Higher-Order Constrained Dependency Pairs for (Universal) Computability*. In: *Proc. MFCS 24*, doi:10.48550/arXiv.2406.19379.

[8] L. Guo & C. Kop (2024): *Higher-Order LCTRSs and Their Termination*. In: *Proc. ESOP 24*, *LNCS* 14577, pp. 331–357, doi:10.1007/978-3-031-57267-8_13.

[9] K. Hagens & C. Kop (2024): *Rewriting Induction for Higher-Order Constrained Term Rewriting Systems*. In: *Proc. LOPSTR 24*, 14919, pp. 202–219, doi:10.1007/978-3-031-71294-4_12. Available at `https://doi.org/10.1007/978-3-031-71294-4_12`.

[10] U.S. Reddy (1990): *Term Rewriting Induction*. In: *Proc. CADE '90*, *LNCS* 449, pp. 162–177, doi:10.1007/3-540-52885-7_86.