UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Formal Methods for Constraint-Based Testing and Reversible Debugging in Erlang

Adrián Palacios Corella

A thesis submitted for the degree of
Doctor of Philosophy

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Advisor: Germán Vidal Oriola

January 2020

Abstract

Erlang is a message-passing concurrent, functional programming language based on the actor model. These and other features make it especially appropriate for distributed, soft real-time applications. In the recent years, Erlang's popularity has increased due to the demand for concurrent services.

However, developing error-free systems in Erlang is quite a challenge. Although Erlang avoids many problems by design (e.g., deadlocks), some other problems may appear. Here, testing and debugging techniques based on formal methods may be helpful to detect, locate and fix programming errors in Erlang.

In this thesis we propose several methods for testing and debugging in Erlang. In particular, these methods are based on semantics models for concolic testing, property-based testing, causal-consistent reversible debugging and causal-consistent replay debugging of Erlang programs. We formally prove the main properties of our proposals and design open-source tools that implement these methods.

Resumen

Erlang es un lenguaje de programación funcional con concurrencia mediante paso de mensajes basado en el modelo de actores. Éstas y otras características lo hacen especialmente adecuado para aplicaciones distribuidas en tiempo real acrítico. En los últimos años, la popularidad de Erlang ha aumentado debido a la demanda de servicios concurrentes.

No obstante, desarrollar sistemas Erlang libres de errores es un reto considerable. A pesar de que Erlang evita muchos problemas por diseño (por ejemplo, puntos muertos), algunos otros problemas pueden aparecer. En este contexto, las técnicas de testing y depuración basadas en métodos formales pueden ser útiles para detectar, localizar y arreglar errores de programación en Erlang.

En esta tesis proponemos varios métodos para testing y depuración en Erlang. En particular, estos métodos están basados en modelos semánticos para *concolic testing*, pruebas basadas en propiedades, depuración reversible con consistencia causal y repetición reversible con consistencia causal de programas Erlang. Además, probamos formalmente las principales propiedades de nuestras propuestas y diseñamos herramientas de código abierto que implementan estos métodos.

Resum

Erlang és un llenguatge de programació funcional amb concurrència mitjançant pas de missatges basat en el model d'actors. Estes i altres característiques el fan especialment adequat per a aplicacions distribuïdes en temps real acrític. En els últims anys, la popularitat d'Erlang ha augmentat degut a la demanda de servicis concurrents.

No obstant, desenvolupar sistemes Erlang lliures d'errors és un repte considerable. Encara que Erlang evita molts problemes per disseny (per exemple, punts morts), alguns altres problemes poden aparéixer. En este context, les tècniques de testing y depuració basades en mètodes formals poden ser útils per a detectar, localitzar y arreglar errors de programació en Erlang.

En esta tesis proposem diversos mètodes per a testing i depuració en Erlang. En particular, estos mètodes estan basats en models semàntics per a *concolic testing*, testing basat en propietats, depuració reversible amb consistència causal i repetició reversible amb consistència causal de programes Erlang. A més, provem formalment les principals propietats de les nostres propostes i dissenyem ferramentes de codi obert que implementen estos mètodes.

Acknowledgments

Five years ago, I got the opportunity to start my PhD studies out of nowhere. Until then, I had only dreamed about becoming a doctor. But now I could not be happier that I took the chance. It has been a long journey, but I have had the pleasure to share it with a lot of smart and caring people.

First, I will be forever grateful to my advisor, Germán Vidal, for teaching me everything he knows about the many aspects of research. For always offering his guidance both in academic and personal matters, and for all the encouragement I received from day one. I cannot stress enough how lucky I feel to have had him as my advisor during this time.

Next, I would like to express my gratitude to the members of the MiST and ELP groups. Special thanks to my labmates Sergio, Damián, Tama, Insa and Carlos. I hope you guys keep our traditions alive and continue to support each other, as we have always done. Many thanks to the professors, particularly to Josep, Marisa and Alicia, for introducing me to the world of teaching.

My formation would not have been the same without the several research stays and visits that I carried out during these years. Therefore, I would like to acknowledge the hospitality of the people who helped or hosted me abroad. In particular, I thank Naoki Nishida for his invaluable help to understand the Japanese culture, and Ivan Lanese for all those interesting discussions. Without both of them, much of the work in this thesis would not have been possible. Thanks to Maurizio Proietti, Emanuele De Angelis, Alberto Pettorossi and Fabio Fioravanti, for inviting me to their *merendas* and making me feel at home. Thanks to Kareem Khazem, Mark Tuttle and the rest of the group, for getting me my first taste of industry. And special thanks to my fellow interns Malte, Felipe, Debasmita, Lucas and Konstantinos, for making the internship such an amazing experience.

I would also like to thank the members of my thesis committee: Claudio Antares Mezzina, María Alpuente, Clara Benac, Lars-Ake Fredlund and Michael Kirkedal Thomsen, for their useful comments to improve this thesis.

Appreciation is due to my friends as well, especially to Vicent, Vidal, Christian, Fabra, Álvaro and José Andrés, for our treasured friendship and all the encouragement I received from them since the beginning.

Last but not least, I would like to thank my family for their unconditional love and support. To my brother, whose intellectual growth has surpassed all expectations. I have no doubt in saying that he will be a renowned scientist one day. To my wife Sagrario, whose clever advice helps me to look at things from a different angle. She did not hesitate to join me in our upcoming adventures, and for that I am very grateful. And finally, to my parents, who have worked really hard during their lives so that I could chase my dreams. Those dreams are becoming real now, but let us keep on dreaming.

Contents

Part I

Introduction and Objectives

Chapter 1

Introduction

1.1 The Erlang Language

Over the last years, concurrent programming has become a common practice, mainly due to the constantly growing number of smart devices around the world. Concurrent services such as telecommunications, broadcasting and cloud computing require the design, implementation and maintenance of scalable, distributed and highly-available systems [129]. The demand for such concurrent services has forced many companies to adopt new technologies (frameworks or languages, such as Erlang) in order to build this kind of systems.

Erlang [5] is a concurrent, functional programming language based on the actor model [62]. This language has many distinguishing features (dynamic typing, concurrency via asynchronous message passing or hot code loading) that make it especially appropriate for distributed, fault-tolerant, soft-real time applications. For instance, Erlang has been extensively used for WhatsApp [40], the most popular messaging application in the world. To put it in perspective, WhatsApp has around 450 million active users that send 65 billion messages and make 100 million voice calls on a daily basis—a true example of a large-scale system.

However, developing considerable, error-free systems in Erlang is generally regarded to be a challenge. Basically, programming errors in Erlang stem from common (functional) programming errors and concurrency. Even in a purely sequential setting, Erlang's typing discipline—strong/dynamic typing— does not help to detect simple programming errors. In other programming languages, these errors would be easily caught by the type checker at compile time, but in Erlang they will go

unnoticed and cause program crashes during runtime execution. Clearly, the case for concurrent programs is even worse [66]. Although Erlang avoids some concurrency bugs by design (e.g., deadlocks), other concurrency bugs such as message order violations and livelocks can still show up in programs. Here, software tools based on formal methods may be helpful to detect, locate and fix errors in Erlang programs, overall improving their code quality and achieving a higher confidence in the developed systems.

Formal methods [64] refers to mathematically based techniques used for the specification, design and verification of software and hardware systems. In the context of software development, formal methods are based on a mathematical foundation (e.g., formal semantics) for describing and reasoning about complex systems. For instance, formal semantics allow us to precisely define the meaning of a programming language and correctly reason about programs written in that language and their properties. In the case of Erlang, there is not a commonly accepted formal semantics, though there have been some attempts to define exhaustive semantics for the language [18, 130]. Only an outdated specification of Core Erlang [25]—an intermediate language used by the Erlang compiler—has been available since 2004.

In this thesis, we focus on defining semantics as the basis for testing and debugging techniques for Erlang programs. Testing and debugging are two major processes in software development which can be used together for the purpose of detection (testing) and localization (debugging) of programming errors. Typically, programmers will first use a testing tool to find potential errors in their programs. If this process reveals an unexpected result, a debugging tool will then be used for examining the program execution and track down the error source within the program. In the following, we introduce these techniques separately and describe the main problems that cannot be handled by existing approaches.

1.2 Constraint-Based Testing

Software testing is one of the most widely used techniques for software validation. As mentioned earlier, the goal of testing is to detect programming errors. In general, testing is an automatic process that may require some effort from programmers in order to work properly. In this thesis, we explore new approaches to perform constraint-based testing of Erlang programs. Constraint-based methods make use of constraint solving techniques for different purposes (testing in our case). In particular, we present two proposals based on symbolic execution to perform concolic testing and property-based testing in Erlang.

Symbolic execution [74] was an innovative technique introduced in the mid '70s as an alternative to random testing. Following this technique, input data is replaced by symbolic values and then, at each branching point of the execution, all feasible paths are explored and its associated constraints on symbolic values are stored. Therefore, symbolic states include a so-called path condition with the constraints stored so far. As a result, test cases are produced by solving the constraints in the leaves of the symbolic execution tree. Unfortunately, the path explosion problem makes test case generation based on symbolic execution difficult to scale. For instance, as soon as a path condition cannot be proved satisfiable, the execution of its corresponding branch is terminated in order to ensure soundness, which translates into poor coverage in many cases.

Concolic execution [56, 125] is a proposal that combines concrete and symbolic execution to overcome some drawbacks of previous approaches. Concolic execution takes some (initially random) concrete input data and performs simultaneously a concrete execution and a symbolic execution driven by the concrete one. Then, if the path condition becomes too complex for the constraint solver to prove its satisfiability, concrete data can be pushed from the concrete execution, thereby allowing symbolic computation to continue. Concolic execution forms the basis of some model checking and test-case generation tools (e.g., SAGE [57] and Java Pathfinder [115]). Test cases produced with this technique usually achieve better code coverage than other approaches solely based on symbolic execution, and it scales up better to complex or large programs.

Despite its popularity in the imperative programming paradigm, we can only find a few preliminary approaches to concolic execution in the context of declarative programming [54, 100]. Mostly, these approaches are based on augmented interpreters capable of dealing with symbolic values. In contrast, we consider whether concolic execution can be performed by program instrumentation.

Property-based testing [29] is another popular method for testing in functional programming languages. Here, instead of supplying specific inputs, the developer defines some properties to be satisfied in terms of input-output pairs. Then, random inputs are generated and the program is run with those input values. Finally, the outputs are then used to check whether the desired properties hold or not. QuickCheck [29] is the first tool that implemented property-based testing for the functional language Haskell, and a similar approach has been followed for other programming languages—Erlang [7, 113], Java [72, 144] and Prolog [4] to name a few.

The difficulty in property-based testing is that, when working with user-defined types and filters, users must provide a generator in addition to properties. Basically, a generator is a program that randomly constructs proper input data for a given type

(e.g., binary search trees). Unfortunately, writing generators is a time-consuming, error-prone activity which may result in the generation of of non-valid or overly restricted inputs. In our work, we explore an alternative approach based on symbolic execution to relieve users from writing generators.

1.3 Reversible Debugging

Debugging is the process of locating and removing programming errors. Generally, debugging is not an automatic process but rather performed by developers with the assistance of debugging tools. Therefore, debugging tools mostly focus on representing as much information as possible regarding a program computation.

According to recent studies [17, 136], the annual cost of debugging software is $312 billions, and it is estimated that time spent in debugging accounts for 49.9% of the total programming time. This situation is not likely to improve in the near future, given the increasing demand for concurrent software [129]. In the context of message-passing concurrent languages, most of the approaches to software validation and debugging are based on some form of static analysis (e.g., Dialyzer [91], McErlang [48], Soter [39]) or testing (e.g., QuickCheck [28], PropEr [113], CutEr [54]). Nevertheless, these techniques are helpful only to find specific categories of problems. In this setting, reversible debugging—inspired by principles of reversible computation—may be useful to complement previous approaches.

Reversible computation [12, 46, 145] is an unconventional computing model where computation is reversible. In other words, a reversible computation can go forwards (i.e., the usual direction) or backwards (i.e., undoing a regular computation). Nowadays, reversible computation is a relevant concept in many different fields like cellular automata [103] or quantum computing [143]. In fact, reversible debugging [51] is a successful application of reversible computation which can be trivially implemented on top of any reversible language or formalism. In this thesis, we first study reversible computation in the context of term rewriting [9, 132], a computational model that underlies most rule-based programming languages. We consider that term rewriting provides an excellent framework to formally define reversible computation in a functional context. Then, we extend this notion to message-passing concurrent languages, where causal consistency is additionally required to account for causality.

Hence, causal-consistent reversible debuggers allow users to run concurrent programs in a controllably reversible manner. If something (potentially) incorrect shows up, the user can stop the forward computation and go backwards—in a causal-

consistent way—to look for the root cause of the problem. In this context, we say that a backward step is causal consistent [35, 83] if an action cannot be undone until all the actions that depend on it have already been undone. Therefore, causal-consistent reversibility is particularly relevant for concurrent debugging because it allows us to undo the actions of a given process in a stepwise manner while ignoring the actions of the remaining processes, unless they are causally related. In a traditional reversible debugger, one can only go backwards in exactly the reverse order of the forward execution. Essentially, this corresponds to linearizing a concurrent computation, and it makes focusing on undoing the actions of a given process much harder, since they can be interleaved with unrelated actions from other processes.

Furthermore, traditional debuggers (like the one included in Erlang/OTP) are sometimes not particularly useful for debugging concurrent programs. The reason is that, when an unusual interleaving brings up an error, recompiling the program for debugging may lead to a completely different computation. This problem is usually tackled by so-called replay debugging, which allows the user to record a program execution and replay it inside the debugger. However, in concurrent programs, part of the program execution may not be relevant for the debugging session. For instance, some processes may not have interacted with the one showing a misbehavior, or they may have interacted only at the very beginning of their execution. Replaying these actions is pointless as well as distracting for the user. In this thesis, we introduce (controlled) causal-consistent replay and extend our reversible debugger to enable causal-consistent replay debugging of previously logged computations.

1.4 Objectives and Contributions

The main objective of this thesis is to improve software quality in Erlang programs through the usage of testing and debugging tools based on formal methods. To this end, we design semantics-based techniques for concolic testing, property-based testing, causal-consistent reversible debugging and causal-consistent replay debugging. We provide mathematical proofs for the most interesting properties of the proposed methods, in addition to software tools that experimentally show these approaches to be feasible and efficient in practice.

The main contributions of this thesis can be split into three categories:

1. **Term Rewriting:** Term rewriting is a computation model that underlies most rule-based programming languages which, in general, is not reversible. We consider term rewriting to be an excellent framework to define reversible computation in a functional context and prove its main properties. This work

may be useful in different contexts: reversible debugging, parallel discrete event simulation and bidirectional program transformation, to name a few.

   (a) **Reversible Extension of Term Rewriting:** We present a general notion of reversible term rewriting by defining a Landauer embedding. Given a rewrite system $\mathcal{R}$ and its associated (standard) rewrite relation $\to_{\mathcal{R}}$, we define a reversible extension of rewriting composed of a forward relation $\rightharpoonup_{\mathcal{R}}$ and a backward relation $\leftharpoondown_{\mathcal{R}}$, such that $\rightharpoonup_{\mathcal{R}}$ is a conservative extension of $\to_{\mathcal{R}}$ and, moreover, $(\rightharpoonup_{\mathcal{R}})^{-1} = \leftharpoondown_{\mathcal{R}}$. Given a rewriting reduction $s \to^*_{\mathcal{R}} t$, this reversible relation aims at computing the term s from t and $\mathcal{R}$ in a decidable and deterministic way, which is not possible using $(\to_{\mathcal{R}})^{-1}$ since it is generally non-deterministic.
   (b) **Reversibilization of Rewrite Systems:** A reversibilization procedure transforms an irreversible computation device into a reversible one. We introduce a reversibilization transformation for a quite general class of rewrite systems and we present an improvement that removes labels from traces (i.e., a memory optimization) by performing an injectivity analysis on the rewrite system rules.
   (c) **Implementation of Reversibilization Transformations:** The aforementioned reversibilization transformation is implemented as a tool that reads an input TRS file and then applies sequentially the following transformations: flattening, simplification of constructor conditions, injectivization, and inversion. The tools prints out the rewrite systems obtained at each step and is publicly available through a web interface.

2. **Reversible Debugging:** Traditional debuggers are not particularly helpful for debugging message-passing concurrent programs. In contrast, causal-consistent reversible debugging is especially relevant to this end because it allows to undo the actions of a given process while ignoring unrelated processes. Similarly, causal-consistent replay debugging allows to reproduce a previously logged computation while excluding unrelated actions from other processes.

   (a) **Reversible Semantics for Erlang:** A reversible semantics can go both forward and backward. We introduce an uncontrolled reversible semantics for Erlang and prove its most interesting properties, including its causal consistency.
   (b) **Rollback Semantics for Erlang:** In contrast to an uncontrolled semantics, a controlled semantics drives backwards execution in the direction

of specific checkpoints before resuming forward computation. We introduce a controlled version of the backward semantics that essentially models a rollback operator, and prove its main properties.

(c) **Causal-Consistent Reversible Debugger for Erlang:** An improved version of the reversible semantics is implemented as a causal-consistent reversible debugger for Erlang programs. We show its application in two different scenarios—message order violation and livelock—to show the usefulness of the tool.

(d) **Causal-Consistent Replay Semantics for Erlang:** Unusual interleavings that lead to an erroneous behavior during regular execution may be hard to reproduce on a debugger. We introduce both an uncontrolled and a controlled causal-consistent replay semantics for Erlang, which allow to replay causally equivalent executions from a previously logged computation, and we prove their basic properties.

(e) **Causal-Consistent Replay Debugger for Erlang:** The controlled replay semantics is implemented in an adapted version of our reversible debugger, resulting in a causal-consistent replay debugger for Erlang. Additionally, we provide a tool for logging computations in a standard environment so that they can be (causally) reproduced in our replay debugger.

3. **Constraint-Based Testing:** Constraint-based testing techniques make use of constraint solving techniques during symbolic execution for different purposes. In this thesis, we focus on concolic testing and property-based testing for automatic test case generation.

(a) **Concolic Execution by Program Instrumentation:** We consider whether concolic execution can be performed by program instrumentation. First, we introduce an instrumented version of a semantics for Erlang. Then, we propose a program transformation that instruments Erlang programs so that its execution in a standard environment is equivalent to the execution of the original program with the instrumented semantics.

(b) **Implementation of a Concolic Testing Tool:** The previously mentioned program transformation is implemented as a tool that instruments Erlang programs and enables the execution of the instrumented programs to obtain the associated sequence of events.

(c) **Automatic Test Case Generation:** Property-based testing often requires the specification of data generators designed by developers. We

explore an alternative approach that relieves developers from this task. This approach takes the data type and filter specifications and automatically generates data of the given type that satisfies the given filter.

(d) **Implementation of Test Case Generator Tool:** The previously mentioned approach is implemented in a fully automatic tool composed of six modules that enables automatic test case generation from PropEr (a popular property-based testing tool for Erlang) specifications in a more efficient way.

1.5 Structure of this Thesis

This thesis is written as a collection of articles. This format (as opposed to a monograph thesis) is more suited for PhD theses where a significant amount of contributions have already been published in international conferences or journals.

In Part I we have already introduced the motivation and objectives of this work. The remaining sections summarize the activities of the author over the course of his PhD studies (publications, research projects and stays). Part II comprises a collection of the main articles that support this thesis. In this part, each chapter is an adapted version of an already-published work in a scientific conference or journal. In all cases, the articles have been adapted from author versions and a link to the final (published) version of the article is included, as established by the guidelines for thesis writing from the Doctorate School. Then, Part III presents a general discussion about the results obtained in this thesis. The discussion is split into a few categories where the results are discussed and compared with related work. Finally, Part IV concludes this thesis with a final review of the main contributions of our work and a discussion about future work.

1.6 Publications

Over the course of his PhD studies, the author has published several scientific articles which are listed in this section. Here, we distinguish between papers published in high-impact journals and international conferences. An underlined title means that the paper has been included in Part II.

Articles in journals listed in the Journal Citation Report (JCR):

- N. Nishida, A. Palacios and G. Vidal. Reversible computation in term rewriting. *Journal of Logical and Algebraic Methods in Programming*, 94: 128–149, 2018.

- I. Lanese, N. Nishida, A. Palacios and G. Vidal. A theory of reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming*, 100: 71–97, 2018.

Full papers in international conferences:

- A. Palacios and G. Vidal. Towards Modelling Actor-Based Concurrency in Term Rewriting. *Proceedings of the 2nd International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2015)*, OASIcs 46: 12–29, 2015.
- A. Palacios and G. Vidal. Concolic Execution in Functional Programming by Program Instrumentation. *Proceedings of the 25th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2015)*, LNCS 9527: 277–292, 2015.
- N. Nishida, A. Palacios and G. Vidal. Reversible Term Rewriting. *Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, LIPIcs 52: 28:1–28:18, 2016.
- N. Nishida, A. Palacios and G. Vidal. A Reversible Semantics for Erlang. *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*, LNCS 10184: 259–274, 2017.
- I. Lanese, N. Nishida, A. Palacios and G. Vidal. CauDEr: A Causal-Consistent Reversible Debugger for Erlang. *Proceedings of the 14th International Symposium on Functional and Logic Programming (FLOPS 2018)*, LNCS 10818: 247–263, 2018.
- E. De Angelis, F. Fioravanti, A. Palacios, A. Pettorossi, M. Proietti. Bounded Symbolic Execution for Runtime Error Detection of Erlang Programs. *Proceedings of the 5th Workshop on Horn Clauses for Verification and Synthesis (HCVS 2018)*, EPTCS 278: 19–26, 2018.
- I. Lanese, A. Palacios and G. Vidal. Causal-Consistent Replay Debugging for Message Passing Programs. *Proceedings of the 39th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2019)*, LNCS 11535: 167–184, 2019.
- E. De Angelis, F. Fioravanti, A. Palacios, A. Pettorossi, M. Proietti. Property-Based Test Case Generators for Free. *Proceedings of the 13th International Conference on Tests and Proofs (TAP 2019)*, LNCS 11823: 186–206, 2019.

1.7 Research Projects

This thesis would not have possible without funding for research projects. The main contributions and derivative works of this thesis have been made in the context of the following projects:

- **CAVI-ROSE:** National research project *CAVI-ROSE: Computer assisted validation by using sound and rigorous methods*, funded by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* (MINECO) under grant TIN2013-44742-C4-1-R, was part of a coordinated project with other Spanish universities whose main goals aimed at advancing the knowledge and technology within the area of software validation. Its lead applicants were Germán Vidal and Josep Silva, and the project concluded after a period of 4 years (2014-2017). The author of thesis was awarded a 4-year PhD scholarship (associated to the CAVI-ROSE project) *Ayudas para contratos predoctorales para la formación de doctores*, funded by the EU (FEDER) and the Spanish MINECO under FPI grant BES-2014-069749.

- **MERINET:** National research project *MERINET: Rigorous Methods for the Future Internet*, funded by the EU (FEDER) and the Spanish MINECO[1] under grant TIN2016-76843-C4-1-R, was part of a coordinated project with other Spanish universities whose main goal is producing new (more competitive) techniques, methods and tools for software analysis and development, with special emphasis in web systems and programming languages. Its lead applicants were Germán Vidal and Josep Silva, and the project has been ongoing since 2017.

- **COST Action IC1405:** COST Action is defined as a "network dedicated to scientific collaboration, complementing national research funds" and therefore cannot be considered a proper research project. But given the great influence that this network has had on our research (in terms of collaboration, discussion, etc.), we have decided to acknowledge it in this thesis anyway. *COST Action IC1405 on Reversible Computation - extending horizons of computing* had a duration of four years and held at least two meetings each year. In these meetings, the participants were encouraged to present their latest works to foster technical discussions on the related topics. The author attended these meetings in multiple occasions and applied for short-term scientific missions

[1] The scientific and innovation competences were later transferred to the Spanish *Ministerio de Ciencia, Innovación y Universidades* (MICINN).

(i.e., travel grants) a couple of times. More importantly, thanks to these meetings we were able to start our collaboration with Ivan Lanese, an expert on reversible computation who has co-authored some works included in this thesis.

1.8 Research Stays

Research stays in international countries are essential for learning about other research methodologies in foreign teams. Over the course of his PhD studies, the author has carried out two research stays and one internship abroad. Below we present a summary of these stays.

- **Nagoya University — Nagoya, Japan:** The first research stay was carried out from September 14, 2016 to December 14, 2016 (3 months) and supervised by Naoki Nishida, a long-time collaborator of our research group. During this stay, the author focused on the implementation of reversibilization transformations and other contributions in the context of reversible term rewriting. A complementary grant *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D españoles y extranjeros 2016*, funded by the EU (FEDER) and the Spanish MINECO under FPI grant EEBB-I-16-11469, was awarded to the author in order to finance this research stay.

- **Consiglio Nazionale delle Ricerche — Rome, Italy:** This research stay was conducted from September 14, 2017 to December 14, 2017 (3 months) and supervised by Maurizio Proietti. During this stay, the author worked closely with Maurizio and other members of his research group to design a CLP semantics for Erlang and develop the corresponding CLP interpreter. A complementary grant *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D españoles y extranjeros 2017*, funded by the EU (FEDER) and the Spanish MINECO under FPI grant EEBB-I-17-12101, was awarded to the author in order to finance this research stay.

- **Amazon Web Services — Boston, United States of America:** This internship was carried out from April 15, 2019 to July 5, 2019 and supervised by Kareem Khazeem and Mark R. Tuttle. Amazon Web Services hired the author as an intern of the Automated Reasoning Group, a team that focuses on applying formal methods to industrial code.

Part II

Selected Papers

Chapter 2

Reversible Computation in Term Rewriting

Naoki Nishida[1], Adrián Palacios[2], Germán Vidal[2]

[1] Graduate School of Informatics, Nagoya University
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
nishida@i.nagoya-u.ac.jp

[2] MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
{apalacios,gvidal}@dsic.upv.es

Abstract. Essentially, in a reversible programming language, for each forward computation from state S to state S', there exists a constructive method to go backwards from state S' to state S. Besides its theoretical interest, reversible computation is a fundamental concept which is relevant in many different areas like cellular automata, bidirectional program transformation, or quantum computing, to name a few.

In this work, we focus on term rewriting, a computation model that underlies most rule-based programming languages. In general, term rewriting is not

This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* (MINECO) under grants TIN2013-44742-C4-1-R and TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic), and by the COST Action IC1405 on Reversible Computation - extending horizons of computing. Adrián Palacios was partially supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores* and *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D*, MINECO (SEIDI), under FPI grants BES-2014-069749 and EEBB-I-16-11469. Part of this research was done while the second and third authors were visiting Nagoya University; they gratefully acknowledge their hospitality. This chapter is an adapted author version of the paper published in *"Naoki Nishida, Adrián Palacios, Germán Vidal: Reversible computation in term rewriting. Journal of Logical and Algebraic Methods in Programming 94: 128–149 (2018)"*. DOI: https://doi.org/10.1016/j.jlamp.2017.10.003.

reversible, even for injective functions; namely, given a rewrite step $t_1 \to t_2$, we do not always have a decidable method to get t_1 from t_2. Here, we introduce a conservative extension of term rewriting that becomes reversible. Furthermore, we also define two transformations, injectivization and inversion, to make a rewrite system reversible using standard term rewriting. We illustrate the usefulness of our transformations in the context of bidirectional program transformation.

2.1 Introduction

The notion of reversible computation can be traced back to Landauer's pioneering work [80]. Although Landauer was mainly concerned with the energy consumption of erasing data in irreversible computing, he also claimed that every computer can be made reversible by saving the *history* of the computation. However, as Landauer himself pointed out, this would only postpone the problem of erasing the tape of a reversible Turing machine before it could be reused. Bennett [11] improved the original proposal so that the computation now ends with a tape that only contains the output of a computation and the initial source, thus deleting all remaining "garbage" data, though it performs twice the usual computation steps. More recently, Bennett's result is extended in [32] to nondeterministic Turing machines, where it is also proved that transforming an irreversible Turing machine into a reversible one can be done with a quadratic loss of space. We refer the interested reader to, e.g., [12, 46, 145] for a high level account of the principles of reversible computation.

In the last decades, reversible computing and *reversibilization* (transforming an irreversible computation device into a reversible one) have been the subject of intense research, giving rise to successful applications in many different fields, e.g., cellular automata [103], where reversibility is an essential property, bidirectional program transformation [97], where reversibility helps to automate the generation of inverse functions (see Section 2.6), reversible debugging [51], where one can go both forward and backward when seeking the cause of an error, parallel discrete event simulation [124], where reversible computation is used to undo the effects of speculative computations made on a wrong assumption, quantum computing [143], where all computations should be reversible, and so forth. The interested reader can find detailed surveys in the *state of the art* reports of the different working groups of COST Action IC1405 on Reversible Computation [68].

In this work, we introduce reversibility in the context of *term rewriting* [9, 132], a computation model that underlies most rule-based programming languages. In contrast to other, more *ad-hoc* approaches, we consider that term rewriting is an excel-

lent framework to rigorously define reversible computation in a functional context and formally prove its main properties. We expect our work to be useful in different (sequential) contexts, like reversible debugging, parallel discrete event simulation or bidirectional program transformation, to name a few. In particular, Section 2.6 presents a first approach to formalize bidirectional program transformation in our setting.

To be more precise, we present a general and intuitive notion of *reversible* term rewriting by defining a Landauer embedding. Given a rewrite system $\mathcal{R}$ and its associated (standard) rewrite relation $\to_{\mathcal{R}}$, we define a reversible extension of rewriting with two components: a forward relation $\rightharpoonup_{\mathcal{R}}$ and a backward relation $\leftharpoondown_{\mathcal{R}}$, such that $\rightharpoonup_{\mathcal{R}}$ is a conservative extension of $\to_{\mathcal{R}}$ and, moreover, $(\rightharpoonup_{\mathcal{R}})^{-1} = \leftharpoondown_{\mathcal{R}}$. We note that the inverse rewrite relation, $(\to_{\mathcal{R}})^{-1}$, is not an appropriate basis for "reversible" rewriting since we aim at defining a technique to *undo* a particular reduction. In other words, given a rewriting reduction $s \to^*_{\mathcal{R}} t$, our reversible relation aims at computing the term s from t and $\mathcal{R}$ in a decidable and deterministic way, which is not possible using $(\to_{\mathcal{R}})^{-1}$ since it is generally non-deterministic, non-confluent, and non-terminating, even for systems defining injective functions (see Example 2.6). In contrast, our backward relation $\leftharpoondown_{\mathcal{R}}$ is deterministic (thus confluent) and terminating. Moreover, our relation proceeds backwards step by step, i.e., the number of reduction steps in $s \rightharpoonup^*_{\mathcal{R}} t$ and $t \leftharpoondown^*_{\mathcal{R}} s$ are the same.

In order to introduce a reversibilization transformation for rewrite systems, we use a *flattening* transformation so that the reduction at top positions of terms suffices to get a normal form in the transformed systems. For instance, given the following rewrite system:

$$\begin{array}{rcl} \mathsf{add}(\mathsf{0}, y) & \to & y, \\ \mathsf{add}(\mathsf{s}(x), y) & \to & \mathsf{s}(\mathsf{add}(x, y)) \end{array}$$

defining the addition on natural numbers built from constructors $\mathsf{0}$ and $\mathsf{s}(\ )$, we produce the following *flattened* (conditional) system:

$$\begin{array}{rcl} \mathcal{R} = \{ \quad \mathsf{add}(\mathsf{0}, y) & \to & y, \\ \mathsf{add}(\mathsf{s}(x), y) & \to & \mathsf{s}(z) \Leftarrow \mathsf{add}(x, y) \twoheadrightarrow z \quad \} \end{array}$$

(see Example 2.29 for more details). This allows us to provide an improved notion of reversible rewriting in which some information (namely, the positions where reduction takes place) is not required anymore. This opens the door to *compile* the reversible extension of rewriting into the system rules. Loosely speaking, given a system $\mathcal{R}$, we produce new systems $\mathcal{R}_f$ and $\mathcal{R}_b$ such that *standard* rewriting in $\mathcal{R}_f$, i.e., $\to_{\mathcal{R}_f}$, coincides with the forward reversible extension $\rightharpoonup_{\mathcal{R}}$ in the original system, and analogously $\to_{\mathcal{R}_b}$ is equivalent to $\leftharpoondown_{\mathcal{R}}$. E.g., for the system $\mathcal{R}$ above, we

would produce

$$\begin{aligned}\mathcal{R}_f = \{ \quad & \mathsf{add}^{\mathtt{i}}(0, y) \to \langle y, \beta_1 \rangle, \\ & \mathsf{add}^{\mathtt{i}}(\mathsf{s}(x), y) \to \langle \mathsf{s}(z), \beta_2(w) \rangle \Leftarrow \mathsf{add}^{\mathtt{i}}(x, y) \twoheadrightarrow \langle z, w \rangle \ \}\end{aligned}$$

$$\begin{aligned}\mathcal{R}_b = \{ \quad & \mathsf{add}^{-1}(y, \beta_1) \to \langle 0, y \rangle, \\ & \mathsf{add}^{-1}(\mathsf{s}(z), \beta_2(w)) \to \langle \mathsf{s}(x), y \rangle \Leftarrow \mathsf{add}^{-1}(z, w) \to \langle x, y \rangle \quad \ \}\end{aligned}$$

where $\mathsf{add}^{\mathtt{i}}$ is an injective version of function add, add^{-1} is the inverse of $\mathsf{add}^{\mathtt{i}}$, and β_1, β_2 are fresh symbols introduced to label the rules of $\mathcal{R}$.

In this work, we will mostly consider *conditional* rewrite systems, not only to have a more general notion of reversible rewriting, but also to define a reversibilization technique for unconditional rewrite systems, since the application of *flattening* (cf. Section 2.4) may introduce conditions in a system that is originally unconditional, as illustrated above.

This paper is an extended version of [108]. In contrast to [108], our current paper includes the proofs of technical results, the reversible extension of term rewriting is introduced first in the unconditional case (which is simpler and more intuitive), and presents an improved injectivization transformation when the system includes injective functions. Furthermore, a prototype implementation of the reversibilization technique is publicly available from `http://kaz.dsic.upv.es/rev-rewriting.html`.

The paper is organized as follows. After introducing some preliminaries in Section 2.2, we present our approach to reversible term rewriting in Section 2.3. Section 2.4 introduces the class of *pure constructor* systems where all reductions take place at topmost positions, so that storing this information in reversible rewrite steps becomes unnecessary. Then, Section 2.5 presents injectivization and inversion transformations in order to make a rewrite system reversible with standard rewriting. Here, we also present an improvement of the transformation for injective functions. The usefulness of these transformations is illustrated in Section 2.6. Finally, Section 2.7 discusses some related work and Section 2.8 concludes and points out some ideas for future research.

2.2 Preliminaries

We assume familiarity with basic concepts of term rewriting. We refer the reader to, e.g., [9] and [132] for further details.

2.2.1 Terms and Substitutions

A *signature* $\mathcal{F}$ is a set of ranked function symbols. Given a set of variables $\mathcal{V}$ with $\mathcal{F} \cap \mathcal{V} = \varnothing$, we denote the domain of *terms* by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We use $\mathsf{f}, \mathsf{g}, \ldots$ to denote functions and $x, y, \ldots$ to denote variables. Positions are used to address the nodes of a term viewed as a tree. A *position* p in a term t, in symbols $p \in \mathcal{P}os(t)$, is represented by a finite sequence of natural numbers, where ϵ denotes the root position. We let $t|_p$ denote the *subterm* of t at position p and $t[s]_p$ the result of *replacing the subterm* $t|_p$ by the term s. $\mathcal{V}ar(t)$ denotes the set of variables appearing in t. We also let $\mathcal{V}ar(t_1, \ldots, t_n)$ denote $\mathcal{V}ar(t_1) \cup \cdots \cup \mathcal{V}ar(t_n)$. A term t is *ground* if $\mathcal{V}ar(t) = \varnothing$.

A *substitution* $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a mapping from variables to terms such that $\mathcal{D}\mathsf{om}(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is its domain. A substitution σ is *ground* if $x\sigma$ is ground for all $x \in \mathcal{D}\mathsf{om}(\sigma)$. Substitutions are extended to morphisms from $\mathcal{T}(\mathcal{F}, \mathcal{V})$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ in the natural way. We denote the application of a substitution σ to a term t by $t\sigma$ rather than $\sigma(t)$. The identity substitution is denoted by id. We let "$\circ$" denote the composition of substitutions, i.e., $\sigma \circ \theta(x) = (x\theta)\sigma = x\theta\sigma$. The *restriction* $\theta\restriction_V$ of a substitution θ to a set of variables V is defined as follows: $x\theta\restriction_V = x\theta$ if $x \in V$ and $x\theta\restriction_V = x$ otherwise.

2.2.2 Term Rewriting Systems

A set of rewrite rules $l \to r$ such that l is a nonvariable term and r is a term whose variables appear in l is called a *term rewriting system* (TRS for short); terms l and r are called the left-hand side and the right-hand side of the rule, respectively. We restrict ourselves to finite signatures and TRSs. Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *defined* symbols $\mathcal{D}_\mathcal{R}$ are the root symbols of the left-hand sides of the rules and the *constructors* are $\mathcal{C}_\mathcal{R} = \mathcal{F} \setminus \mathcal{D}_\mathcal{R}$. *Constructor terms* of $\mathcal{R}$ are terms over $\mathcal{C}_\mathcal{R}$ and $\mathcal{V}$, denoted by $\mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$. We sometimes omit $\mathcal{R}$ from $\mathcal{D}_\mathcal{R}$ and $\mathcal{C}_\mathcal{R}$ if it is clear from the context. A substitution σ is a *constructor substitution* (of $\mathcal{R}$) if $x\sigma \in \mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$ for all variables x.

For a TRS $\mathcal{R}$, we define the associated rewrite relation $\to_\mathcal{R}$ as the smallest binary relation on terms satisfying the following: given terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $s \to_\mathcal{R} t$ iff there exist a position p in s, a rewrite rule $l \to r \in \mathcal{R}$, and a substitution σ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$; the rewrite step is sometimes denoted by $s \to_{p, l\to r} t$ to make explicit the position and rule used in this step. The instantiated left-hand side $l\sigma$ is called a *redex*. A term s is called *irreducible* or in *normal form* with respect to a TRS $\mathcal{R}$ if there is no term t with $s \to_\mathcal{R} t$. A substitution is called *normalized* with respect to $\mathcal{R}$ if every variable in the domain is replaced by a normal

form with respect to $\mathcal{R}$. We sometimes omit "with respect to $\mathcal{R}$" if it is clear from the context. A *derivation* is a (possibly empty) sequence of rewrite steps. Given a binary relation $\rightarrow$, we denote by $\rightarrow^*$ its reflexive and transitive closure, i.e., $s \rightarrow^*_{\mathcal{R}} t$ means that s can be reduced to t in $\mathcal{R}$ in zero or more steps; we also use $s \rightarrow^n_{\mathcal{R}} t$ to denote that s can be reduced to t in exactly n steps.

We further assume that rewrite rules are labeled, i.e., given a TRS $\mathcal{R}$, we denote by $\beta : l \rightarrow r$ a rewrite rule with label β. Labels are unique in a TRS. Also, to relate label β to fixed variables, we consider that the variables of the rewrite rules are not renamed[1] and that the reduced terms are always ground. Equivalently, one could require terms to be variable disjoint with the variables of the rewrite system, but we require groundness for simplicity. We often write $s \rightarrow_{p,\beta} t$ instead of $s \rightarrow_{p,l\rightarrow r} t$ if rule $l \rightarrow r$ is labeled with β.

2.2.3 Conditional Term Rewrite Systems

In this paper, we also consider *conditional* term rewrite systems (CTRSs); namely oriented 3-CTRSs, i.e., CTRSs where extra variables are allowed as long as $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(C)$ for any rule $l \rightarrow r \Leftarrow C$ [101]. In *oriented* CTRSs, a conditional rule $l \rightarrow r \Leftarrow C$ has the form $l \rightarrow r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$, where each oriented equation $s_i \twoheadrightarrow t_i$ is interpreted as reachability ($\rightarrow^*_{\mathcal{R}}$). In the following, we denote by $\overline{o_n}$ a sequence of elements $o_1, \ldots, o_n$ for some n. We also write $\overline{o_{i,j}}$ for the sequence $o_i, \ldots, o_j$ when $i \leq j$ (and the empty sequence otherwise). We write $\overline{o}$ when the number of elements is not relevant. In addition, we denote a condition $o_1 \twoheadrightarrow o'_1, \ldots, o_n \twoheadrightarrow o'_n$ by $\overline{o_n \twoheadrightarrow o'_n}$.

As in the unconditional case, we consider that rules are labeled and that labels are unique in a CTRS. And, again, to relate label β to fixed variables, we consider that the variables of the conditional rewrite rules are not renamed and that the reduced terms are always ground.

For a CTRS $\mathcal{R}$, the associated rewrite relation $\rightarrow_{\mathcal{R}}$ is defined as the smallest binary relation satisfying the following: given ground terms $s, t \in \mathcal{T}(\mathcal{F})$, we have $s \rightarrow_{\mathcal{R}} t$ iff there exist a position p in s, a rewrite rule $l \rightarrow r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$, and a ground substitution σ such that $s|_p = l\sigma$, $s_i\sigma \rightarrow^*_{\mathcal{R}} t_i\sigma$ for all $i = 1, \ldots, n$, and $t = s[r\sigma]_p$.

In order to simplify the presentation, we only consider *deterministic* CTRSs (DCTRSs), i.e., oriented 3-CTRSs where, for each rule $l \rightarrow r \Leftarrow \overline{s_n \twoheadrightarrow t_n}$, we have

[1] This will become useful in the next section where the reversible extension of rewriting keeps a "history" of a computation in the form of a list of terms $\beta(p, \sigma)$, and we want the domain of σ to be a subset of the left-hand side of the rule labeled with β.

$\mathcal{V}\mathsf{ar}(s_i) \subseteq \mathcal{V}\mathsf{ar}(l, \overline{t_{i-1}})$ for all $i = 1, \ldots, n$ (see Section 2.3.2 for a justification of this requirement and how it could be relaxed to arbitrary 3-CTRSs). Intuitively speaking, the use of DCTRs allows us to compute the bindings for the variables in the condition of a rule in a deterministic way. E.g., given a ground term s and a rule $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n}$ with $s|_p = l\theta$, we have that $s_1\theta$ is ground. Therefore, one can reduce $s_1\theta$ to some term s_1' such that s_1' is an instance of $t_1\theta$ with some ground substitution θ_1. Now, we have that $s_2\theta\theta_1$ is ground and we can reduce $s_2\theta\theta_1$ to some term s_2' such that s_2' is an instance of $t_2\theta\theta_1$ with some ground substitution θ_2, and so forth. If all equations in the condition hold using $\theta_1, \ldots, \theta_n$, we have that $s \to_{p,\beta} s[r\sigma]_p$ with $\sigma = \theta\theta_1 \ldots \theta_n$.

Example 2.1. Consider the following DCTRS $\mathcal{R}$ that defines the function double that doubles the value of its argument when it is an even natural number:

$$\begin{array}{ll} \beta_1 : \quad \mathsf{add}(0, y) \to y & \beta_4 : \quad \mathsf{even}(0) \to \mathsf{true} \\ \beta_2 : \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) & \beta_5 : \mathsf{even}(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{even}(x) \\ \beta_3 : \quad \mathsf{double}(x) \to \mathsf{add}(x, x) \Leftarrow \mathsf{even}(x) \twoheadrightarrow \mathsf{true} & \end{array}$$

Given the term $\mathsf{double}(\mathsf{s}(\mathsf{s}(0)))$ we have, for instance, the following derivation:

$$\begin{array}{lll} \mathsf{double}(\mathsf{s}(\mathsf{s}(0))) \to_{\epsilon,\beta_3} & \mathsf{add}(\mathsf{s}(\mathsf{s}(0)), \mathsf{s}(\mathsf{s}(0))) & \text{since } \mathsf{even}(\mathsf{s}(\mathsf{s}(0))) \to_{\mathcal{R}}^* \mathsf{true} \\ & & \text{with } \sigma = \{x \mapsto \mathsf{s}(\mathsf{s}(0))\} \\ \to_{\epsilon,\beta_2} & \mathsf{s}(\mathsf{add}(\mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)))) & \text{with } \sigma = \{x \mapsto \mathsf{s}(0),\ y \mapsto \mathsf{s}(\mathsf{s}(0))\} \\ \to_{1,\beta_2} & \mathsf{s}(\mathsf{s}(\mathsf{add}(0, \mathsf{s}(\mathsf{s}(0))))) & \text{with } \sigma = \{x \mapsto 0, y \mapsto \mathsf{s}(\mathsf{s}(0))\} \\ \to_{1.1,\beta_1} & \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))) & \text{with } \sigma = \{y \mapsto \mathsf{s}(\mathsf{s}(0))\} \end{array}$$

2.3 Reversible Term Rewriting

In this section, we present a conservative extension of the rewrite relation which becomes reversible. In the following, we use $\rightharpoonup_{\mathcal{R}}$ to denote our *reversible* (forward) term rewrite relation, and $\leftharpoondown_{\mathcal{R}}$ to denote its application in the reverse (backward) direction. Note that, in principle, we do not require $\leftharpoondown_{\mathcal{R}} = \rightharpoonup_{\mathcal{R}}^{-1}$, i.e., we provide independent (constructive) definitions for each relation. Nonetheless, we will prove that $\leftharpoondown_{\mathcal{R}} = \rightharpoonup_{\mathcal{R}}^{-1}$ indeed holds (cf. Theorems 2.9 and 2.20). In some approaches to reversible computing, both forward and backward relations should be deterministic. Here, we will only require deterministic *backward* steps, while forward steps might be non-deterministic, as it is often the case in term rewriting.

2.3.1 Unconditional Term Rewrite Systems

We start with unconditional TRSs since it is conceptually simpler and thus will help the reader to better understand the key ingredients of our approach. In the next section, we will consider the more general case of DCTRSs.

Given a TRS $\mathcal{R}$, reversible rewriting is defined on pairs $\langle t, \pi \rangle$, where t is a ground term and π is a trace (the "history" of the computation so far). Here, a *trace* in $\mathcal{R}$ is a list of *trace terms* of the form $\beta(p, \sigma)$ such that β is a label for some rule $l \to r \in \mathcal{R}$, p is a position, and σ is a substitution with $\mathcal{D}\mathsf{om}(\sigma) = \mathcal{V}\mathsf{ar}(l) \backslash \mathcal{V}\mathsf{ar}(r)$ which will record the bindings of erased variables when $\mathcal{V}\mathsf{ar}(l) \backslash \mathcal{V}\mathsf{ar}(r) \neq \varnothing$ (and $\sigma = id$ if $\mathcal{V}\mathsf{ar}(l) \backslash \mathcal{V}\mathsf{ar}(r) = \varnothing$).[2] Our trace terms have some similarities with *proof terms* [132]. However, proof terms do not store the bindings of erased variables (and, to the best of our knowledge, they are only defined for unconditional TRSs, while we use trace terms both for unconditional and conditional TRSs).

Our reversible term rewriting relation is only defined on *safe* pairs:

Definition 2.2. Let $\mathcal{R}$ be a TRS. The pair $\langle s, \pi \rangle$ is *safe* in $\mathcal{R}$ iff, for all $\beta(p, \sigma)$ in π, σ is a ground substitution with $\mathcal{D}\mathsf{om}(\sigma) = \mathcal{V}\mathsf{ar}(l) \backslash \mathcal{V}\mathsf{ar}(r)$ and $\beta : l \to r \in \mathcal{R}$.

In the following, we often omit $\mathcal{R}$ when referring to traces and safe pairs if the underlying TRS is clear from the context.

Safety is not necessary when applying a forward reduction step, but will become essential for the backward relation $\leftharpoondown_{\mathcal{R}}$ to be correct. E.g., all traces that come from the forward reduction of some initial pair with an empty trace will be safe (see below). Reversible rewriting is then introduced as follows:

Definition 2.3. Let $\mathcal{R}$ be a TRS. A reversible rewrite relation $\rightharpoonup_{\mathcal{R}}$ is defined on safe pairs $\langle t, \pi \rangle$, where t is a ground term and π is a trace in $\mathcal{R}$. The reversible rewrite relation extends standard rewriting as follows:[3]

$$\langle s, \pi \rangle \rightharpoonup_{\mathcal{R}} \langle t, \beta(p, \sigma') : \pi \rangle$$

iff there exist a position $p \in \mathcal{P}os(s)$, a rewrite rule $\beta : l \to r \in \mathcal{R}$, and a ground substitution σ such that $s|_p = l\sigma$, $t = s[r\sigma]_p$, and $\sigma' = \sigma{\restriction}_{\mathcal{V}\mathsf{ar}(l) \backslash \mathcal{V}\mathsf{ar}(r)}$. The reverse relation, $\leftharpoondown_{\mathcal{R}}$, is then defined as follows:

$$\langle t, \beta(p, \sigma') : \pi \rangle \leftharpoondown_{\mathcal{R}} \langle s, \pi \rangle$$

[2] Note that if a rule $l \to r$ is non-erasing, i.e., $\mathcal{V}\mathsf{ar}(l) = \mathcal{V}\mathsf{ar}(r)$, then $\sigma = id$.

[3] In the following, we consider the usual infix notation for lists where $[\,]$ is the empty list and $x : xs$ is a list with head x and tail xs.

iff $\langle t, \beta(p, \sigma') : \pi \rangle$ is a safe pair in $\mathcal{R}$ and there exist a ground substitution θ and a rule $\beta : l \to r \in \mathcal{R}$ such that $\mathcal{D}\mathsf{om}(\theta) = \mathcal{V}\mathsf{ar}(r)$, $t|_p = r\theta$ and $s = t[l\theta\sigma']_p$. Note that $\theta\sigma' = \sigma'\theta = \theta \cup \sigma'$, where $\cup$ is the union of substitutions, since $\mathcal{D}\mathsf{om}(\theta) = \mathcal{V}\mathsf{ar}(r)$, $\mathcal{D}\mathsf{om}(\sigma') = (\mathcal{V}\mathsf{ar}(l) \backslash \mathcal{V}\mathsf{ar}(r))$ and both substitutions are ground, so $\mathcal{D}\mathsf{om}(\theta) \cap \mathcal{D}\mathsf{om}(\sigma') = \varnothing$.

We denote the union of both relations $\rightharpoonup_{\mathcal{R}} \cup \leftharpoondown_{\mathcal{R}}$ by $\rightleftharpoons_{\mathcal{R}}$.

Example 2.4. Let us consider the following TRS $\mathcal{R}$ defining the addition on natural numbers built from $\mathsf{0}$ and $\mathsf{s}(\ )$, and the function fst that returns its first argument:

$$\begin{array}{lrcl} \beta_1: & \mathsf{add}(\mathsf{0}, y) & \to & y \\ \beta_2: & \mathsf{add}(\mathsf{s}(x), y) & \to & \mathsf{s}(\mathsf{add}(x, y)) \end{array} \qquad \beta_3: \ \mathsf{fst}(x, y) \to x$$

Given the term $\mathsf{fst}(\mathsf{add}(\mathsf{s}(\mathsf{0}), \mathsf{0}), \mathsf{0})$, we have, for instance, the following reversible (forward) derivation:

$$\begin{array}{rcl} \langle \mathsf{fst}(\mathsf{add}(\mathsf{s}(\mathsf{0}), \mathsf{0}), \mathsf{0}), [\,] \rangle & \rightharpoonup_{\mathcal{R}} & \langle \mathsf{fst}(\mathsf{s}(\mathsf{add}(\mathsf{0}, \mathsf{0})), \mathsf{0}), [\beta_2(1, id)] \rangle \\ & \rightharpoonup_{\mathcal{R}} & \langle \mathsf{s}(\mathsf{add}(\mathsf{0}, \mathsf{0})), [\beta_3(\epsilon, \{y \mapsto \mathsf{0}\}), \beta_2(1, id)] \rangle \\ & \rightharpoonup_{\mathcal{R}} & \langle \mathsf{s}(\mathsf{0}), [\beta_1(1, id), \beta_3(\epsilon, \{y \mapsto \mathsf{0}\}), \beta_2(1, id)] \rangle \end{array}$$

The reader can easily check that $\langle \mathsf{s}(\mathsf{0}), [\beta_1(1, id), \beta_3(\epsilon, \{y \mapsto \mathsf{0}\}), \beta_2(1, id)] \rangle$ is reducible to $\langle \mathsf{fst}(\mathsf{add}(\mathsf{s}(\mathsf{0}), \mathsf{0}), \mathsf{0}), [\,] \rangle$ using the backward relation $\leftharpoondown_{\mathcal{R}}$ by performing exactly the same steps but in the backward direction.

An easy but essential property of $\rightharpoonup_{\mathcal{R}}$ is that it is a conservative extension of standard rewriting in the following sense (we omit its proof since it is straightforward):

Theorem 2.5. *Let $\mathcal{R}$ be a TRS. Given terms s, t, if $s \to_{\mathcal{R}}^* t$, then for any trace π there exists a trace π' such that $\langle s, \pi \rangle \rightharpoonup_{\mathcal{R}}^* \langle t, \pi' \rangle$.*

Here, and in the following, we assume that $\leftarrow_{\mathcal{R}} = (\to_{\mathcal{R}})^{-1}$, i.e., $s \to_{\mathcal{R}}^{-1} t$ is denoted by $s \leftarrow_{\mathcal{R}} t$. Observe that the backward relation is not a conservative extension of $\leftarrow_{\mathcal{R}}$: in general, $t \leftarrow_{\mathcal{R}} s$ does not imply $\langle t, \pi' \rangle \leftharpoondown_{\mathcal{R}} \langle s, \pi \rangle$ for any arbitrary trace π'. This is actually the purpose of our notion of reversible rewriting: $\leftharpoondown_{\mathcal{R}}$ should not extend $\leftarrow_{\mathcal{R}}$ but is only aimed at performing *exactly the same steps* of the forward computation whose trace was stored, but in the reverse order. Nevertheless, one can still ensure that for all steps $t \leftarrow_{\mathcal{R}} s$, there exists some trace π' such that $\langle t, \pi' \rangle \leftharpoondown_{\mathcal{R}} \langle s, \pi \rangle$ (which is an easy consequence of the above result and Theorem 2.9 below).

Example 2.6. Consider again the following TRS $\mathcal{R} = \{\beta : \mathsf{snd}(x, y) \to y\}$. Given the reduction $\mathsf{snd}(1, 2) \to_{\mathcal{R}} 2$, there are infinitely many reductions for 2 using $\leftarrow_{\mathcal{R}}$, e.g., $2 \leftarrow_{\mathcal{R}} \mathsf{snd}(1, 2)$, $2 \leftarrow_{\mathcal{R}} \mathsf{snd}(2, 2)$, $2 \leftarrow_{\mathcal{R}} \mathsf{snd}(3, 2)$, etc. The relation is also non-terminating: $2 \leftarrow_{\mathcal{R}} \mathsf{snd}(1, 2) \leftarrow_{\mathcal{R}} \mathsf{snd}(1, \mathsf{snd}(1, 2)) \leftarrow_{\mathcal{R}} \cdots$. In contrast, given a pair $\langle 2, \pi \rangle$, we can only perform a single deterministic and finite reduction (as proved below). For instance, if $\pi = [\beta(\epsilon, \{x \mapsto 1\}), \beta(2, \{x \mapsto 1\})]$, then the only possible reduction is $\langle 2, \pi \rangle \leftharpoondown_{\mathcal{R}} \langle \mathsf{snd}(1, 2), [\beta(2, \{x \mapsto 1\})] \rangle \leftharpoondown_{\mathcal{R}} \langle \mathsf{snd}(1, \mathsf{snd}(1, 2)), [\,] \rangle \not\leftharpoondown_{\mathcal{R}}$.

Now, we state a lemma which shows that safe pairs are preserved through reversible term rewriting (both in the forward and backward directions):

Lemma 2.7. *Let $\mathcal{R}$ be a TRS. Let $\langle s, \pi \rangle$ be a safe pair. If $\langle s, \pi \rangle \rightleftharpoons^*_{\mathcal{R}} \langle t, \pi' \rangle$, then $\langle t, \pi' \rangle$ is also safe.*

Proof. We prove the claim by induction on the length k of the derivation. Since the base case $k = 0$ is trivial, consider the inductive case $k > 0$. Assume a derivation of the form $\langle s, \pi \rangle \rightleftharpoons^*_{\mathcal{R}} \langle s_0, \pi_0 \rangle \rightleftharpoons_{\mathcal{R}} \langle t, \pi' \rangle$. By the induction hypothesis, we have that $\langle s_0, \pi_0 \rangle$ is a safe pair. Now, we distinguish two cases depending on the last step. If we have $\langle s_0, \pi_0 \rangle \rightharpoonup_{\mathcal{R}} \langle t, \pi' \rangle$, then there exist a position $p \in \mathcal{P}os(s_0)$, a rewrite rule $\beta : l \to r \in \mathcal{R}$, and a ground substitution σ such that $s_0|_p = l\sigma$, $t = s_0[r\sigma]_p$, $\sigma' = \sigma\restriction_{\mathcal{V}\mathsf{ar}(l) \setminus \mathcal{V}\mathsf{ar}(r)}$, and $\pi' = \beta(p, \sigma') : \pi_0$. Then, since σ' is ground and $\mathcal{D}\mathsf{om}(\sigma') = \mathcal{V}\mathsf{ar}(l) \setminus \mathcal{V}\mathsf{ar}(r)$ by construction, the claim follows straightforwardly. If the last step has the form $\langle s_0, \pi_0 \rangle \leftharpoondown_{\mathcal{R}} \langle t, \pi' \rangle$, then the claim follows trivially since each step with $\leftharpoondown_{\mathcal{R}}$ only removes trace terms from π_0. □

Hence, since any pair with an empty trace is safe the following result, which states that every pair that is *reachable* from an initial pair with an empty trace is safe, straightforwardly follows from Lemma 2.7:

Proposition 2.8. Let $\mathcal{R}$ be a TRS. If $\langle s, [\,] \rangle \rightleftharpoons^*_{\mathcal{R}} \langle t, \pi \rangle$, then $\langle t, \pi \rangle$ is safe.

Now, we state the reversibility of $\rightharpoonup_{\mathcal{R}}$, i.e., the fact that $(\rightharpoonup_{\mathcal{R}})^{-1} = \leftharpoondown_{\mathcal{R}}$ (and thus the reversibility of $\leftharpoondown_{\mathcal{R}}$ and $\rightleftharpoons_{\mathcal{R}}$, too).

Theorem 2.9. *Let $\mathcal{R}$ be a TRS. Given the safe pairs $\langle s, \pi \rangle$ and $\langle t, \pi' \rangle$, for all $n \geq 0$, $\langle s, \pi \rangle \rightharpoonup^n_{\mathcal{R}} \langle t, \pi' \rangle$ iff $\langle t, \pi' \rangle \leftharpoondown^n_{\mathcal{R}} \langle s, \pi \rangle$.*

Proof. ($\Rightarrow$) We prove the claim by induction on the length n of the derivation $\langle s, \pi \rangle \rightharpoonup^n_{\mathcal{R}} \langle t, \pi' \rangle$. Since the base case $n = 0$ is trivial, let us consider the inductive case $n > 0$. Consider a derivation $\langle s, \pi \rangle \rightharpoonup^{n-1}_{\mathcal{R}} \langle s_0, \pi_0 \rangle \rightharpoonup_{\mathcal{R}} \langle t, \pi' \rangle$. By Lemma 2.7, both $\langle s_0, \pi_0 \rangle$

and $\langle t, \pi' \rangle$ are safe. By the induction hypothesis, we have $\langle s_0, \pi_0 \rangle \leftharpoondown_{\mathcal{R}}^{n-1} \langle s, \pi \rangle$. Consider now the step $\langle s_0, \pi_0 \rangle \rightharpoonup_{\mathcal{R}} \langle t, \pi' \rangle$. Then, there is a position $p \in \mathcal{P}os(s_0)$, a rule $\beta : l \to r \in \mathcal{R}$ and a ground substitution σ such that $s_0|_p = l\sigma$, $t = s_0[r\sigma]_p$, $\sigma' = \sigma\restriction_{\mathcal{V}\mathsf{ar}(l)\setminus\mathcal{V}\mathsf{ar}(r)}$, and $\pi' = \beta(p, \sigma') : \pi_0$. Let $\theta = \sigma\restriction_{\mathcal{V}\mathsf{ar}(r)}$. Then, we have $\langle t, \pi' \rangle \leftharpoondown_{\mathcal{R}} \langle s'_0, \pi_0 \rangle$ with $t|_p = r\theta$, $\beta : l \to r \in \mathcal{R}$ and $s'_0 = t[l\theta\sigma']_p$. Moreover, since $\sigma = \theta\sigma'$, we have $s'_0 = t[l\theta\sigma']_p = t[l\sigma]_p = s_0$, and the claim follows.

($\Leftarrow$) This direction proceeds in a similar way. We prove the claim by induction on the length n of the derivation $\langle t, \pi' \rangle \leftharpoondown_{\mathcal{R}}^{n} \langle s, \pi \rangle$. As before, we only consider the inductive case $n > 0$. Let us consider a derivation $\langle t, \pi' \rangle \leftharpoondown_{\mathcal{R}}^{n-1} \langle s_0, \pi_0 \rangle \leftharpoondown_{\mathcal{R}} \langle s, \pi \rangle$. By Lemma 2.7, both $\langle s_0, \pi_0 \rangle$ and $\langle s, \pi \rangle$ are safe. By the induction hypothesis, we have $\langle s_0, \pi_0 \rangle \rightharpoonup_{\mathcal{R}}^{n-1} \langle t, \pi' \rangle$. Consider now the reduction step $\langle s_0, \pi_0 \rangle \leftharpoondown_{\mathcal{R}} \langle s, \pi \rangle$. Then, we have $\pi_0 = \beta(p, \sigma') : \pi$, $\beta : l \to r \in \mathcal{R}$, and there exists a ground substitution θ with $\mathcal{D}\mathsf{om}(\theta) = \mathcal{V}\mathsf{ar}(r)$ such that $s_0|_p = r\theta$ and $s = s_0[l\theta\sigma']_p$. Moreover, since $\langle s_0, \pi_0 \rangle$ is safe, we have that $\mathcal{D}\mathsf{om}(\sigma') = \mathcal{V}\mathsf{ar}(l)\setminus\mathcal{V}\mathsf{ar}(r)$ and, thus, $\mathcal{D}\mathsf{om}(\theta) \cap \mathcal{D}\mathsf{om}(\sigma') = \varnothing$. Let $\sigma = \theta\sigma'$. Then, since $s|_p = l\sigma$ and $\mathcal{D}\mathsf{om}(\sigma') = \mathcal{V}\mathsf{ar}(l)\setminus\mathcal{V}\mathsf{ar}(r)$, we can perform the step $\langle s, \pi \rangle \rightharpoonup_{\mathcal{R}} \langle s'_0, \beta(p, \sigma') : \pi \rangle$ with $s'_0 = s[r\sigma]_p = s[r\theta\sigma']_p = s[r\theta]_p = s_0[r\theta]_p = s_0$, and the claim follows. □

The next corollary is then immediate:

Corollary 2.10. *Let $\mathcal{R}$ be a TRS. Given the safe pairs $\langle s, \pi \rangle$ and $\langle t, \pi' \rangle$, for all $n \geq 0$, $\langle s, \pi \rangle \rightleftharpoons_{\mathcal{R}}^{n} \langle t, \pi' \rangle$ iff $\langle t, \pi' \rangle \rightleftharpoons_{\mathcal{R}}^{n} \langle s, \pi \rangle$.*

A key issue of our notion of reversible rewriting is that the backward rewrite relation $\leftharpoondown_{\mathcal{R}}$ is deterministic (thus confluent), terminating, and has a constructive definition:

Theorem 2.11. *Let $\mathcal{R}$ be a TRS. Given a safe pair $\langle t, \pi' \rangle$, there exists at most one pair $\langle s, \pi \rangle$ such that $\langle t, \pi' \rangle \leftharpoondown_{\mathcal{R}} \langle s, \pi \rangle$.*

Proof. First, if there is no step using $\leftharpoondown_{\mathcal{R}}$ from $\langle t, \pi' \rangle$, the claim follows trivially. Now, assume there is at least one step $\langle t, \pi' \rangle \leftharpoondown_{\mathcal{R}} \langle s, \pi \rangle$. We prove that this is the only possible step. By definition, we have $\pi' = \beta(p, \sigma') : \pi$, $p \in \mathcal{P}os(t)$, $\beta : l \to r \in \mathcal{R}$, and there exists a ground substitution θ with $\mathcal{D}\mathsf{om}(\theta) = \mathcal{V}\mathsf{ar}(r)$ such that $t|_p = r\theta$ and $s = t[l\theta\sigma']_p$. The only source of nondeterminism may come from choosing a rule labeled with β and from the computation of the substitution θ. The claim follows trivially from the fact that labels are unique in $\mathcal{R}$ and that, if there is some ground substitution θ' with $\theta' = \mathcal{V}\mathsf{ar}(r)$ and $t|_p = r\theta'$, then $\theta = \theta'$. □

Therefore, $\leftharpoondown_{\mathcal{R}}$ is clearly deterministic and confluent. Termination holds straightforwardly for pairs with finite traces since its length strictly decreases with every

backward step. Note however that even when $\rightharpoonup_{\mathcal{R}}$ and $\leftharpoonup_{\mathcal{R}}$ are terminating, the relation $\rightleftharpoons_{\mathcal{R}}$ is always non-terminating since one can keep going back and forth.

2.3.2 Conditional Term Rewrite Systems

In this section, we extend the previous notions and results to DCTRSs. We note that considering DCTRSs is not enough to make conditional rewriting deterministic. In general, given a rewrite step $s \to_{p,\beta} t$ with p a position of s, $\beta : l \to r \Leftarrow \overline{s_n \to t_n}$ a rule, and σ a substitution such that $s|_p = l\sigma$ and $s_i\sigma \to^*_{\mathcal{R}} t_i\sigma$ for all $i = 1, \ldots, n$, there are three potential sources of non-determinism: the selected position p, the selected rule β, and the substitution σ. The use of DCTRSs can only make deterministic the last one, but the choice of a position and the selection of a rule may still be non-deterministic.

For DCTRSs, the notion of a trace term used for TRSs is not sufficient since we also need to store the traces of the subderivations associated to the condition of the applied rule (if any). Therefore, we generalize the notion of a trace as follows:

Definition 2.12 (trace)**.** Given a CTRS $\mathcal{R}$, a *trace* in $\mathcal{R}$ is recursively defined as follows:

- the empty list is a trace;
- if $\pi, \pi_1, \ldots, \pi_n$ are traces in $\mathcal{R}$, $n \geq 0$, $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$ is a rule, p is a position, and σ is a ground substitution, then $\beta(p, \sigma, \pi_1, \ldots, \pi_n) : \pi$ is a trace in $\mathcal{R}$.

We refer to each component $\beta(p, \sigma, \pi_1, \ldots, \pi_n)$ in a trace as a *trace term*.

Intuitively speaking, a trace term $\beta(p, \sigma, \pi_1, \ldots, \pi_n)$ stores the position of a reduction step, a substitution with the bindings that are required for the step to be reversible (e.g., the bindings for the erased variables, but not only; see below) and the traces associated to the subcomputations in the condition.

The notion of a safe pair is now more involved in order to deal with conditional rules. The motivation for this definition will be explained below, after introducing reversible rewriting for DCTRSs.

Definition 2.13 (safe pair)**.** Let $\mathcal{R}$ be a DCTRS. A trace π is *safe* in $\mathcal{R}$ iff, for all trace terms $\beta(p, \sigma, \overline{\pi_n})$ in π, σ is a ground substitution with $\mathcal{D}\mathsf{om}(\sigma) = (\mathcal{V}\mathsf{ar}(l) \backslash \mathcal{V}\mathsf{ar}(r, \overline{s_n}, \overline{t_n})) \cup \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(t_i) \backslash \mathcal{V}\mathsf{ar}(r, \overline{s_{i+1,n}})$, $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$, and $\overline{\pi_n}$ are safe too. The pair $\langle s, \pi \rangle$ is *safe* in $\mathcal{R}$ iff π is safe.

Reversible (conditional) rewriting can now be introduced as follows:

Definition 2.14 (reversible rewriting)**.** Let $\mathcal{R}$ be a DCTRS. The reversible rewrite relation $\rightharpoonup_{\mathcal{R}}$ is defined on safe pairs $\langle t, \pi\rangle$, where t is a ground term and π is a trace in $\mathcal{R}$. The reversible rewrite relation extends standard conditional rewriting as follows:

$$\langle s, \pi\rangle \rightharpoonup_{\mathcal{R}} \langle t, \beta(p, \sigma', \pi_1, \ldots, \pi_n) : \pi\rangle$$

iff there exist a position $p \in \mathcal{P}os(s)$, a rewrite rule $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$, and a ground substitution σ such that $s|_p = l\sigma$, $\langle s_i\sigma, [\,]\rangle \rightharpoonup^*_{\mathcal{R}} \langle t_i\sigma, \pi_i\rangle$ for all $i = 1, \ldots, n$, $t = s[r\sigma]_p$, and $\sigma' = \sigma\restriction_{(\mathcal{V}\mathsf{ar}(l)\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_n},\overline{t_n}))\cup\bigcup_{i=1}^{n}\mathcal{V}\mathsf{ar}(t_i)\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_{i+1,n}})}$. The reverse relation, $\leftharpoondown_{\mathcal{R}}$, is then defined as follows:

$$\langle t, \beta(p, \sigma', \pi_1, \ldots, \pi_n) : \pi\rangle \leftharpoondown_{\mathcal{R}} \langle s, \pi\rangle$$

iff $\langle t, \beta(p, \sigma', \overline{\pi_n}) : \pi\rangle$ is a safe pair in $\mathcal{R}$, $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$ and there is a ground substitution θ such that $\mathcal{D}\mathsf{om}(\theta) = \mathcal{V}\mathsf{ar}(r, \overline{s_n})\setminus\mathcal{D}\mathsf{om}(\sigma')$, $t|_p = r\theta$, $\langle t_i\theta\sigma', \pi_i\rangle \leftharpoondown^*_{\mathcal{R}} \langle s_i\theta\sigma', [\,]\rangle$ for all $i = 1, \ldots, n$, and $s = t[l\theta\sigma']_p$. Note that $\theta\sigma' = \sigma'\theta = \theta \cup \sigma'$ since $\mathcal{D}\mathsf{om}(\theta) \cap \mathcal{D}\mathsf{om}(\sigma') = \varnothing$ and both substitutions are ground.

As in the unconditional case, we denote the union of both relations $\rightharpoonup_{\mathcal{R}} \cup \leftharpoondown_{\mathcal{R}}$ by $\rightleftharpoons_{\mathcal{R}}$.

Example 2.15. Consider again the DCTRS $\mathcal{R}$ from Example 2.1:

$$\begin{array}{ll}
\beta_1 : \quad \mathsf{add}(0, y) \to y & \beta_4 : \quad \mathsf{even}(0) \to \mathsf{true} \\
\beta_2 : \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) & \beta_5 : \mathsf{even}(\mathsf{s}(\mathsf{s}(x))) \to \mathsf{even}(x) \\
\beta_3 : \quad \mathsf{double}(x) \to \mathsf{add}(x, x) \Leftarrow \mathsf{even}(x) \twoheadrightarrow \mathsf{true} &
\end{array}$$

Given the term $\mathsf{double}(\mathsf{s}(\mathsf{s}(0)))$, we have, for instance, the following forward derivation:

$$\begin{aligned}
&\langle \mathsf{double}(\mathsf{s}(\mathsf{s}(0))), [\,]\rangle \\
&\qquad \rightharpoonup_{\mathcal{R}} \ \langle \mathsf{add}(\mathsf{s}(\mathsf{s}(0)), \mathsf{s}(\mathsf{s}(0))), [\beta_3(\epsilon, id, \pi)]\rangle \\
&\qquad \rightharpoonup_{\mathcal{R}} \ \cdots \\
&\qquad \rightharpoonup_{\mathcal{R}} \ \langle \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))), [\beta_1(1.1, id), \beta_2(1, id), \beta_2(\epsilon, id), \beta_3(\epsilon, id, \pi)]\rangle
\end{aligned}$$

where $\pi = [\beta_4(\epsilon, id), \beta_5(\epsilon, id)]$ since we have the following reduction:

$$\langle \mathsf{even}(\mathsf{s}(\mathsf{s}(0))), [\,]\rangle \rightharpoonup_{\mathcal{R}} \langle \mathsf{even}(0), [\beta_5(\epsilon, id)]\rangle \rightharpoonup_{\mathcal{R}} \langle \mathsf{true}, [\beta_4(\epsilon, id), \beta_5(\epsilon, id)]\rangle$$

The reader can easily construct the associated backward derivation:

$$\langle \mathsf{add}(\mathsf{s}(\mathsf{s}(0)), \mathsf{s}(\mathsf{s}(0))), [\beta_1(1.1, id), \beta_2(1, id), \ldots]\rangle \leftharpoondown^*_{\mathcal{R}} \langle \mathsf{double}(\mathsf{s}(\mathsf{s}(0))), [\,]\rangle$$

Let us now explain why we need to store σ' in a step of the form $\langle s, \pi\rangle \rightharpoonup_{\mathcal{R}} \langle t, \beta(p, \sigma', \overline{\pi_n}) : \pi\rangle$. Given a DCTRS, for each rule $l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n}$, the following conditions hold:

- 3-CTRS: $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l, \overline{s_n}, \overline{t_n})$.
- Determinism: for all $i = 1, \ldots, n$, we have $\mathcal{V}\mathsf{ar}(s_i) \subseteq \mathcal{V}\mathsf{ar}(l, \overline{t_{i-1}})$.

Intuitively, the backward relation $\leftharpoondown_{\mathcal{R}}$ can be seen as equivalent to the forward relation $\rightharpoonup_{\mathcal{R}}$ but using a reverse rule of the form $r \to l \Leftarrow t_n \twoheadrightarrow s_n, \ldots, t_1 \twoheadrightarrow s_1$. Therefore, in order to ensure that backward reduction is deterministic, we need the same conditions as above but on the reverse rule:[4]

- 3-CTRS: $\mathcal{V}\mathsf{ar}(l) \subseteq \mathcal{V}\mathsf{ar}(r, \overline{s_n}, \overline{t_n})$.
- Determinism: for all $i = 1, \ldots, n$, $\mathcal{V}\mathsf{ar}(t_i) \subseteq \mathcal{V}\mathsf{ar}(r, \overline{s_{i+1,n}})$.

Since these conditions cannot be guaranteed in general, we store

$$\sigma' = \sigma\restriction_{(\mathcal{V}\mathsf{ar}(l)\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_n},\overline{t_n}))\cup\bigcup_{i=1}^{n}\mathcal{V}\mathsf{ar}(t_i)\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_{i+1,n}})}$$

in the trace term so that $(r \to l \Leftarrow t_n \twoheadrightarrow s_n, \ldots, t_1 \twoheadrightarrow s_1)\sigma'$ is deterministic and fulfills the conditions of a 3-CTRS by construction, i.e., $\mathcal{V}\mathsf{ar}(l\sigma') \subseteq \mathcal{V}\mathsf{ar}(r\sigma', \overline{s_n\sigma'}, \overline{t_n\sigma'})$ and for all $i = 1, \ldots, n$, $\mathcal{V}\mathsf{ar}(t_i\sigma') \subseteq \mathcal{V}\mathsf{ar}(r\sigma', \overline{s_{i+1,n}\sigma'})$; see the proof of Theorem 2.21 for more details.

Example 2.16. Consider the following DCTRS:

$$\beta_1 : \mathsf{f}(x, y, m) \to \mathsf{s}(w) \Leftarrow \mathsf{h}(x) \twoheadrightarrow x, \mathsf{g}(y, 4) \twoheadrightarrow w$$
$$\beta_2 : \mathsf{h}(0) \to 0 \qquad \beta_3 : \mathsf{h}(1) \to 1 \qquad \beta_4 : \mathsf{g}(x, y) \to x$$

and the step $\langle \mathsf{f}(0, 2, 4), [\,]\rangle \rightharpoonup_{\mathcal{R}} \langle \mathsf{s}(2), [\beta_1(\epsilon, \sigma', \pi_1, \pi_2)]\rangle$ with $\sigma' = \{m \mapsto 4, x \mapsto 0\}$, $\pi_1 = [\beta_2(\epsilon, id)]$ and $\pi_2 = [\beta_4(\epsilon, \{y \mapsto 4\})]$. The binding of variable m is required to recover the value of the *erased* variable m, but the binding of variable x is also needed to perform the subderivation $\langle x, \pi_1\rangle \leftharpoondown_{\mathcal{R}} \langle \mathsf{h}(x), [\,]\rangle$ when applying a backward step from $\langle \mathsf{s}(2), [\beta_1(\epsilon, \sigma', \pi_1, \pi_2)]\rangle$. If the binding for x were unknown, this step would not be deterministic. As mentioned above, an instantiated reverse rule $(\mathsf{s}(w) \to \mathsf{f}(x, y, m) \Leftarrow w \twoheadrightarrow \mathsf{g}(y, 4), x \twoheadrightarrow \mathsf{h}(x))\sigma' = \mathsf{s}(w) \to \mathsf{f}(0, y, 4) \Leftarrow w \twoheadrightarrow \mathsf{g}(y, 4), 0 \twoheadrightarrow \mathsf{h}(0)$ would be a legal DCTRS rule thanks to σ'.

[4] We note that the notion of a non-erasing rule is extended to the DCTRSs in [110], which results in a similar condition.

We note that similar conditions could be defined for arbitrary 3-CTRSs. However, the conditions would be much more involved; e.g., one had to compute first the *variable dependencies* between the equations in the conditions. Therefore, we prefer to keep the simpler conditions for DCTRSs (where these dependencies are fixed), which is still quite a general class of CTRSs.

Reversible rewriting is also a conservative extension of rewriting for DCTRSs (we omit the proof since it is straightforward):

Theorem 2.17. *Let $\mathcal{R}$ be a DCTRS. Given ground terms s, t, if $s \to^*_{\mathcal{R}} t$, then for any trace π there exists a trace π' such that $\langle s, \pi\rangle \rightharpoonup^*_{\mathcal{R}} \langle t, \pi'\rangle$.*

For the following result, we need some preliminary notions (see, e.g., [132]). For every oriented CTRS $\mathcal{R}$, we inductively define the TRSs $\mathcal{R}_k$, $k \geq 0$, as follows:

$$\begin{aligned}\mathcal{R}_0 &= \varnothing\\ \mathcal{R}_{k+1} &= \{l\sigma \to r\sigma \mid l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R},\ s_i\sigma \to^*_{\mathcal{R}_k} t_i\sigma \text{ for all } i = 1, \ldots, n\}\end{aligned}$$

Observe that $\mathcal{R}_k \subseteq \mathcal{R}_{k+1}$ for all $k \geq 0$. We have $\to_{\mathcal{R}} = \bigcup_{i\geq 0} \to_{\mathcal{R}_i}$. We also have $s \to_{\mathcal{R}} t$ iff $s \to_{\mathcal{R}_k} t$ for some $k \geq 0$. The minimum such k is called the *depth* of $s \to_{\mathcal{R}} t$, and the maximum depth k of $s = s_0 \to_{\mathcal{R}_{k_1}} \cdots \to_{\mathcal{R}_{k_m}} s_m = t$ (i.e., k is the maximum of depths $k_1, \ldots, k_m$) is called the *depth* of the derivation. If a derivation has depth k and length m, we write $s \to^m_{\mathcal{R}_k} t$. Analogous notions can naturally be defined for $\rightharpoonup_{\mathcal{R}}$, $\leftharpoondown_{\mathcal{R}}$, and $\rightleftharpoons_{\mathcal{R}}$.

The next result shows that safe pairs are also preserved through reversible rewriting with DCTRSs:

Lemma 2.18. *Let $\mathcal{R}$ be a DCTRS and $\langle s, \pi\rangle$ a safe pair. If $\langle s, \pi\rangle \rightleftharpoons^*_{\mathcal{R}} \langle t, \pi'\rangle$, then $\langle t, \pi'\rangle$ is also safe.*

Proof. We prove the claim by induction on the lexicographic product (k, m) of the depth k and the length m of the derivation $\langle s, \pi\rangle \rightleftharpoons^m_{\mathcal{R}_k} \langle t, \pi'\rangle$. Since the base case is trivial, we consider the inductive case $(k, m) > (0, 0)$. Consider a derivation $\langle s, \pi\rangle \rightleftharpoons^{m-1}_{\mathcal{R}_k} \langle s_0, \pi_0\rangle \rightleftharpoons_{\mathcal{R}_k} \langle t, \pi'\rangle$. By the induction hypothesis, we have that $\langle s_0, \pi_0\rangle$ is safe. Now, we distinguish two cases depending on the last step. If the last step is $\langle s_0, \pi_0\rangle \rightharpoonup_{\mathcal{R}_k} \langle t, \pi'\rangle$, then there exist a position $p \in \mathcal{P}os(s_0)$, a rewrite rule $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$, and a ground substitution σ such that $s_0|_p = l\sigma$, $\langle s_i\sigma, [\,]\rangle \rightharpoonup^*_{\mathcal{R}_{k_i}} \langle t_i\sigma, \pi_i\rangle$ for all $i = 1, \ldots, n$, $t = s_0[r\sigma]_p$, $\sigma' = \sigma\restriction_{(\mathcal{V}\mathsf{ar}(l)\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_n},\overline{t_n}))\cup\bigcup_{i=1}^n \mathcal{V}\mathsf{ar}(t_i)\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_{i+1,n}})}$, and $\pi' = \beta(p, \sigma', \pi_1, \ldots, \pi_n)$. Then, since $k_i < k$, $i = 1, \ldots, n$, σ' is ground and $\mathcal{D}\mathsf{om}(\sigma') = (\mathcal{V}\mathsf{ar}(l)\setminus\mathcal{V}\mathsf{ar}(r, \overline{s_n}, \overline{t_n})) \cup \bigcup_{i=1}^n \mathcal{V}\mathsf{ar}(t_i)\setminus\mathcal{V}\mathsf{ar}(r, \overline{s_{i+1,n}})$ by construction, the claim follows by induction. Finally,

if the last step has the form $\langle s_0, \pi_0\rangle \leftharpoondown_{\mathcal{R}_k} \langle t, \pi'\rangle$, then the claim follows trivially since a step with $\leftharpoondown_{\mathcal{R}}$ only removes trace terms from π_0. □

As in the unconditional case, the following proposition follows straightforwardly from the previous lemma since any pair with an empty trace is safe.

Proposition 2.19. Let $\mathcal{R}$ be a DCTRS. If $\langle s, [\,]\rangle \rightleftharpoons^*_{\mathcal{R}} \langle t, \pi\rangle$, then $\langle t, \pi\rangle$ is safe in $\mathcal{R}$.

Now, we can already state the reversibility of $\rightharpoonup_{\mathcal{R}}$ for DCTRSs:

Theorem 2.20. *Let $\mathcal{R}$ be a DCTRS. Given the safe pairs $\langle s, \pi\rangle$ and $\langle t, \pi'\rangle$, for all $k, m \geq 0$, $\langle s, \pi\rangle \rightharpoonup^m_{\mathcal{R}_k} \langle t, \pi'\rangle$ iff $\langle t, \pi'\rangle \leftharpoondown^m_{\mathcal{R}_k} \langle s, \pi\rangle$.*

Proof. ($\Rightarrow$) We prove the claim by induction on the lexicographic product (k, m) of the depth k and the length m of the derivation $\langle s, \pi\rangle \rightharpoonup^m_{\mathcal{R}_k} \langle t, \pi'\rangle$. Since the base case is trivial, we consider the inductive case $(k, m) > (0, 0)$. Consider a derivation $\langle s, \pi\rangle \rightharpoonup^{m-1}_{\mathcal{R}_k} \langle s_0, \pi_0\rangle \rightharpoonup_{\mathcal{R}_k} \langle t, \pi'\rangle$ whose associated product is (k, m). By Proposition 2.19, both $\langle s_0, \pi_0\rangle$ and $\langle t, \pi'\rangle$ are safe. By the induction hypothesis, since $(k, m-1) < (k, m)$, we have $\langle s_0, \pi_0\rangle \leftharpoondown^{m-1}_{\mathcal{R}_k} \langle s, \pi\rangle$. Consider now the step $\langle s_0, \pi_0\rangle \rightharpoonup_{\mathcal{R}_k} \langle t, \pi'\rangle$. Thus, there exist a position $p \in \mathcal{P}os(s_0)$, a rule $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$, and a ground substitution σ such that $s_0|_p = l\sigma$, $\langle s_i\sigma, [\,]\rangle \rightharpoonup^*_{\mathcal{R}_{k_i}} \langle t_i\sigma, \pi_i\rangle$ for all $i = 1, \ldots, n$, $t = s_0[r\sigma]_p$, $\sigma' = \sigma\restriction_{(\mathcal{V}\mathsf{ar}(l)\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_n},\overline{t_n}))\cup\bigcup_{i=1}^n \mathcal{V}\mathsf{ar}(t_i)\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_{i+1,n}})}$, and $\pi' = \beta(p, \sigma', \pi_1, \ldots, \pi_n) : \pi_0$. By definition of $\rightharpoonup_{\mathcal{R}_k}$, we have that $k_i < k$ and, thus, $(k_i, m_1) < (k, m_2)$ for all $i = 1, \ldots, n$ and for all m_1, m_2. Hence, by the induction hypothesis, we have $\langle t_i\sigma, \pi_i\rangle \leftharpoondown^*_{\mathcal{R}_{k_i}} \langle s_i\sigma, [\,]\rangle$ for all $i = 1, \ldots, n$. Let $\theta = \sigma\restriction_{\mathcal{V}\mathsf{ar}(r,\overline{s_n})\setminus\mathcal{D}\mathsf{om}(\sigma')}$, so that $\sigma = \theta\sigma'$ and $\mathcal{D}\mathsf{om}(\theta) \cap \mathcal{D}\mathsf{om}(\sigma') = \varnothing$. Therefore, we have $\langle t, \pi'\rangle \leftharpoondown_{\mathcal{R}_k} \langle s'_0, \pi_0\rangle$ with $t|_p = r\theta$, $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$ and $s'_0 = t[l\theta\sigma']_p = t[l\sigma]_p = s_0$, and the claim follows.

($\Leftarrow$) This direction proceeds in a similar way. We prove the claim by induction on the lexicographic product (k, m) of the depth k and the length m of the considered derivation. Since the base case is trivial, let us consider the inductive case $(k, m) > (0, 0)$. Consider a derivation $\langle t, \pi'\rangle \leftharpoondown^{m-1}_{\mathcal{R}_k} \langle s_0, \pi_0\rangle \leftharpoondown_{\mathcal{R}_k} \langle s, \pi\rangle$ whose associated product is (k, m). By Proposition 2.19, both $\langle s_0, \pi_0\rangle$ and $\langle s, \pi\rangle$ are safe. By the induction hypothesis, since $(k, m-1) < (k, m)$, we have $\langle s_0, \pi_0\rangle \rightharpoonup^{m-1}_{\mathcal{R}_k} \langle t, \pi'\rangle$. Consider now the step $\langle s_0, \pi_0\rangle \leftharpoondown_{\mathcal{R}_k} \langle s, \pi\rangle$. Then, we have $\pi_0 = \beta(p, \sigma', \pi_1, \ldots, \pi_n) : \pi$, $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$, and there exists a ground substitution θ with $\mathcal{D}\mathsf{om}(\theta) = \mathcal{V}\mathsf{ar}(r, \overline{s_n})\setminus\mathcal{D}\mathsf{om}(\sigma')$ such that $s_0|_p = r\theta$, $\langle t_i\theta\sigma', \pi_i\rangle \leftharpoondown^*_{\mathcal{R}_{k_i}} \langle s_i\theta\sigma', [\,]\rangle$ for all $i = 1, \ldots, n$, and $s = s_0[l\theta\sigma']_p$. Moreover, since $\langle s_0, \pi_0\rangle$ is safe, we have that $\mathcal{D}\mathsf{om}(\sigma') = (\mathcal{V}\mathsf{ar}(l)\setminus\mathcal{V}\mathsf{ar}(r, \overline{s_n}, \overline{t_n})) \cup \bigcup_{i=1}^n \mathcal{V}\mathsf{ar}(t_i)\setminus\mathcal{V}\mathsf{ar}(r, \overline{s_{i+1,n}})$. By definition of

$\leftharpoondown_{\mathcal{R}_k}$, we have that $k_i < k$ and, thus, $(k_i, m_1) < (k, m_2)$ for all $i = 1, \ldots, n$ and for all m_1, m_2. By the induction hypothesis, we have $\langle s_i\theta\sigma', [\,]\rangle \rightharpoonup^*_{\mathcal{R}_{k_i}} \langle t_i\theta\sigma', \pi_i\rangle$ for all $i = 1, \ldots, n$. Let $\sigma = \theta\sigma'$, with $\mathcal{D}\mathsf{om}(\theta) \cap \mathcal{D}\mathsf{om}(\sigma') = \varnothing$. Then, since $s|_p = l\sigma$, we can perform the step $\langle s, \pi\rangle \rightharpoonup_{\mathcal{R}_k} \langle s'_0, \beta(p, \sigma', \pi_1, \ldots, \pi_n) : \pi\rangle$ with $s'_0 = s[r\sigma]_p = s[r\theta\sigma']_p$; moreover, $s[r\theta\sigma']_p = s[r\theta]_p = s_0[r\theta]_p = s_0$ since $\mathcal{D}\mathsf{om}(\sigma') \cap \mathcal{V}\mathsf{ar}(r) = \varnothing$, which concludes the proof. □

In the following, we say that $\langle t, \pi'\rangle \leftharpoondown_{\mathcal{R}} \langle s, \pi\rangle$ is a *deterministic* step if there is no other, different pair $\langle s'', \pi''\rangle$ with $\langle t, \pi'\rangle \leftharpoondown_{\mathcal{R}} \langle s'', \pi''\rangle$ and, moreover, the subderivations for the equations in the condition of the applied rule (if any) are deterministic, too. We say that a derivation $\langle t, \pi'\rangle \leftharpoondown^*_{\mathcal{R}} \langle s, \pi\rangle$ is deterministic if each reduction step in the derivation is deterministic.

Now, we can already prove that backward reversible rewriting is also deterministic, as in the unconditional case:

Theorem 2.21. *Let $\mathcal{R}$ be a DCTRS. Let $\langle t, \pi'\rangle$ be a safe pair with $\langle t, \pi'\rangle \leftharpoondown^*_{\mathcal{R}} \langle s, \pi\rangle$ for some term s and trace π. Then $\langle t, \pi'\rangle \leftharpoondown^*_{\mathcal{R}} \langle s, \pi\rangle$ is deterministic.*

Proof. We prove the claim by induction on the lexicographic product (k, m) of the depth k and the length m of the steps. The case $m = 0$ is trivial, and thus we let $m > 0$. Assume $\langle t, \pi'\rangle \leftharpoondown^{m-1}_{\mathcal{R}_k} \langle u, \pi''\rangle \leftharpoondown_{\mathcal{R}_k} \langle s, \pi\rangle$. For the base case $k = 1$, the applied rule is unconditional and the proof is analogous to that of Theorem 2.11.

Let us now consider $k > 1$. By definition, if $\langle u, \pi''\rangle \leftharpoondown_{\mathcal{R}_k} \langle s, \pi\rangle$, we have $\pi'' = \beta(p, \sigma', \pi_1, \ldots, \pi_n) : \pi$, $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$ and there exists a ground substitution θ with $\mathcal{D}\mathsf{om}(\theta) = \mathcal{V}\mathsf{ar}(r)$ such that $u|_p = r\theta$, $\langle t_i\theta\sigma', \pi_i\rangle \leftharpoondown^*_{\mathcal{R}_j} \langle s_i\theta\sigma', [\,]\rangle$, $j < k$, for all $i = 1, \ldots, n$, and $s = t[l\theta\sigma']_p$. By the induction hypothesis, the subderivations $\langle t_i\theta\sigma', \pi_i\rangle \leftharpoondown^*_{\mathcal{R}_j} \langle s_i\theta\sigma', [\,]\rangle$ are deterministic, i.e., $\langle s_i\theta\sigma', [\,]\rangle$ is a unique resulting term obtained by reducing $\langle t_i\theta\sigma', \pi_i\rangle$. Therefore, the only remaining source of nondeterminism can come from choosing a rule labeled with β and from the computed substitution θ. On the one hand, the labels are unique in $\mathcal{R}$. As for θ, we prove that this is indeed the only possible substitution for the reduction step. Consider the instance of rule $l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n}$ with σ': $l\sigma' \to r\sigma' \Leftarrow \overline{s_n\sigma' \twoheadrightarrow t_n\sigma'}$. Since $\langle u, \pi''\rangle$ is safe, we have that σ' is a ground substitution and $\mathcal{D}\mathsf{om}(\sigma') = (\mathcal{V}\mathsf{ar}(l) \backslash \mathcal{V}\mathsf{ar}(r, \overline{s_n}, \overline{t_n})) \cup \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(t_i) \backslash \mathcal{V}\mathsf{ar}(r, \overline{s_{i+1,n}})$. Then, the following properties hold:

- $\mathcal{V}\mathsf{ar}(l\sigma') \subseteq \mathcal{V}\mathsf{ar}(r\sigma', \overline{s_n\sigma'}, \overline{t_n\sigma'})$, since σ' is ground and it covers all the variables in $\mathcal{V}\mathsf{ar}(l) \backslash \mathcal{V}\mathsf{ar}(r, \overline{s_n}, \overline{t_n})$.

- $\mathcal{V}\mathsf{ar}(t_i\sigma') \subseteq \mathcal{V}\mathsf{ar}(r\sigma', \overline{s_{i+1,n}\sigma'})$ for all $i = 1, \ldots, n$, since σ' is ground and it covers all variables in $\bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(t_i) \backslash \mathcal{V}\mathsf{ar}(r, \overline{s_{i+1,n}})$.

The above properties guarantee that a rule of the form $r\sigma' \to l\sigma' \Leftarrow t_n\sigma' \twoheadrightarrow s_n\sigma', \ldots, t_1\sigma' \twoheadrightarrow s_1\sigma'$ can be seen as a rule of a DCTRS and, thus, there exists a deterministic procedure to compute θ, which completes the proof. □

Therefore, $\leftharpoondown_{\mathcal{R}}$ is deterministic and confluent. Termination is trivially guaranteed for pairs with a finite trace since the trace's length strictly decreases with every backward step.

2.4 Removing Positions from Traces

Once we have a feasible definition of reversible rewriting, there are two refinements that can be considered: i) reducing the size of the traces and ii) defining a *reversibilization* transformation so that standard rewriting becomes reversible in the transformed system. In this section, we consider the first problem, leaving the second one for the next section.

In principle, one could remove information from the traces by requiring certain conditions on the considered systems. For instance, requiring injective functions may help to remove rule labels from trace terms. Also, requiring *non-erasing* rules may help to remove the second component of trace terms (i.e., the substitutions). In this section, however, we deal with a more challenging topic: removing positions from traces. This is useful not only to reduce the size of the traces but it is also essential to define a reversibilization technique for DCTRSs in the next section.[5] In particular, we aim at transforming a given DCTRS into one that fulfills some conditions that make storing positions unnecessary.

In the following, given a CTRS $\mathcal{R}$, we say that a term t is *basic* [63] if it has the form $f(\overline{t_n})$ with $f \in \mathcal{D}_{\mathcal{R}}$ a defined function symbol and $\overline{t_n} \in \mathcal{T}(\mathcal{C}_{\mathcal{R}}, \mathcal{V})$ constructor terms. Furthermore, in the remainder of this paper, we assume that the right-hand sides of the equations in the conditions of the rules of a DCTRS are constructor terms. This is not a significant restriction since these terms cannot be reduced anyway (since we consider oriented equations in this paper), and still covers most practical examples.

Now, we introduce the following subclass of DCTRSs:

[5] We note that defining a transformation with traces that include positions would be a rather difficult task because positions are *dynamic* (i.e., they depend on the term being reduced) and thus would require a complex (and inefficient) system instrumentation.

Definition 2.22 (pcDCTRS [105])**.** We say that a DCTRS $\mathcal{R}$ is a pcDCTRS ("pc" stands for *pure constructor*) if, for each rule $l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$, we have that l and $\overline{s_n}$ are basic terms and r and $\overline{t_n}$ are constructor terms.

Pure constructor systems are called *normalized* systems in [3]. Also, they are mostly equivalent to the class III_n of conditional systems in [13], where $t_1, \ldots, t_n$ are required to be ground unconditional normal forms instead.[6]

In principle, any DCTRS with basic terms in the left-hand sides (i.e., a *constructor* DCTRS) and constructor terms in the right-hand sides of the equations of the rules can be transformed into a pcDCTRS by applying a few simple transformations: flattening and simplification of constructor conditions. Let us now consider each of these transformations separately. Roughly speaking, flattening involves transforming a term (occurring, e.g., in the right-hand side of a DCTRS or in the condition) with nested defined functions like $\mathsf{f}(\mathsf{g}(x))$ into a term $\mathsf{f}(y)$ and an (oriented) equation $\mathsf{g}(x) \twoheadrightarrow y$, where y is a fresh variable. Formally,

Definition 2.23 (flattening)**.** Let $\mathcal{R}$ be a CTRS, $R = (l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n}) \in \mathcal{R}$ be a rule and R' be a new rule either of the form $l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_i|_p \twoheadrightarrow w, s_i[w]_p \twoheadrightarrow t_i, \ldots, s_n \twoheadrightarrow t_n$, for some $p \in \mathcal{P}os(s_i)$, $1 \leqslant i \leqslant n$, or $l \to r[w]_q \Leftarrow \overline{s_n \twoheadrightarrow t_n}, r|_q \twoheadrightarrow w$, for some $q \in \mathcal{P}os(r)$, where w is a fresh variable.[7] Then, a CTRS $\mathcal{R}'$ is obtained from $\mathcal{R}$ by a *flattening* step if $\mathcal{R}' = (\mathcal{R} \backslash \{R\}) \cup \{R'\}$.

Note that, if an unconditional rule is non-erasing (i.e., $\mathcal{V}\mathsf{ar}(l) \subseteq \mathcal{V}\mathsf{ar}(r)$ for a rule $l \to r$), any conditional rule obtained by flattening is trivially non-erasing too, according to the notion of non-erasingness for DCTRSs in [110].[8]

Flattening is trivially *complete* since any flattening step can be undone by binding the fresh variable again to the selected subterm and, then, proceeding as in the original system. Soundness is more subtle though. In this work, we prove the correctness of flattening for arbitrary DCTRSs with respect to *innermost* rewriting. As usual, the innermost rewrite relation, in symbols, $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$, is defined as the smallest binary relation satisfying the following: given ground terms $s, t \in \mathcal{T}(\mathcal{F})$, we have $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$ iff there exist a position p in s such that no proper subterms of $s|_p$ are reducible, a rewrite rule $l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$, and a normalized ground substitution σ such that $s|_p = l\sigma$, $s_i\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t_i\sigma$, for all $i = 1, \ldots, n$, and $t = s[r\sigma]_p$.

[6] Given a CTRS $\mathcal{R}$, we define $\mathcal{R}_u = \{l \to r \mid l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}\}$. A term is an *unconditional* normal form in $\mathcal{R}$, if it is a normal form in $\mathcal{R}_u$.

[7] The positions p, q can be required to be different from ϵ, but this is not strictly necessary.

[8] Roughly, a DCTRS is considered non-erasing in [110] if its transformation into an unconditional TRS by an unraveling transformation gives rise to a non-erasing TRS.

In order to prove the correctness of flattening, we state the following auxiliary lemma:

Lemma 2.24. *Let $\mathcal{R}$ be a DCTRS. Given terms s and t, with t a normal form, and a position $p \in \mathcal{P}os(s)$, we have $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$ iff $s|_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} w\sigma$ and $s[w\sigma]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$, for some fresh variable w and normalized substitution σ.*

Proof. ($\Rightarrow$) Let us consider an arbitrary position $p \in \mathcal{P}os(s)$. If $s|_p$ is normalized, the proof is straightforward. Otherwise, since we use innermost reduction (*leftmost* innermost, for simplicity), we can represent the derivation $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$ as follows:

$$s[s|_p]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'[s|_p]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'[s'']_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$$

where s'' is a normal form and the subderivation $s[s|_p]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'[s|_p]_p$ reduces the leftmost innermost subterms that are to the left of $s|_p$ (if any). Then, by choosing $\sigma = \{w \mapsto s''\}$ we have $s|_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} w\sigma$ (by mimicking the steps of $s'[s|_p]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'[s'']_p$), $s[w\sigma]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'[w\sigma]_p$ (by mimicking the steps of $s[s|_p]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'[s|_p]_p$), and $s'[w\sigma]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$ (by mimicking the steps of $s'[s'']_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$), which concludes the proof.

($\Leftarrow$) This direction is perfectly analogous to the previous case. We consider an arbitrary position $p \in \mathcal{P}os(s)$ such that $s|_p$ is not normalized (otherwise, the proof is trivial). Now, since derivations are innermost, we can consider that $s[w\sigma]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$ is as follows: $s[w\sigma]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'[w\sigma]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$, where $s[w\sigma]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'[w\sigma]_p$ reduces the innermost subterms to the left of $s|_p$. Therefore, we have $s[s|_p]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'[s|_p]_p$ (by mimicking the steps of $s[w\sigma]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'[w\sigma]_p$), $s'[s|_p]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'[s'']_p$ (by mimicking the steps of $s|_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} w\sigma$, with $\sigma = \{w \mapsto s''\}$), and $s'[s'']_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$ (by mimicking the steps of $s'[w\sigma]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$). □

The following theorem is an easy consequence of the previous lemma:

Theorem 2.25. *Let $\mathcal{R}$ be a DCTRS. If $\mathcal{R}'$ is obtained from $\mathcal{R}$ by a flattening step, then $\mathcal{R}'$ is a DCTRS and, for all ground terms s, t, with t a normal form, we have $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$ iff $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t$.*

Proof. ($\Rightarrow$) We prove the claim by induction on the lexicographic product (k, m) of the depth k and the length m of the derivation $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}_k} t$. Since the base case is trivial, we consider the inductive case $(k, m) > (0, 0)$. Assume that $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}_k} t$ has the

form $s[l\sigma]_u \xrightarrow{\mathsf{i}}_{\mathcal{R}_k} s[r\sigma]_u \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}_k} t$ with $l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$ and $s_i\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}_{k_i}} t_i\sigma$, $k_i < k$, $i = 1, \ldots, n$. If $l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}'$, the claim follows directly by induction. Otherwise, we have that either $l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_i|_p \twoheadrightarrow w, s_i[w]_p \twoheadrightarrow t_i, \ldots, s_n \twoheadrightarrow t_n \in \mathcal{R}'$, for some $p \in \mathcal{P}os(s_i)$, $1 \leqslant i \leqslant n$, or $l \to r[w]_q \Leftarrow \overline{s_n \twoheadrightarrow t_n}, r|_q \twoheadrightarrow w \in \mathcal{R}'$, for some $q \in \mathcal{P}os(r)$, where w is a fresh variable. Consider first the case $l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_i|_p \twoheadrightarrow w, s_i[w]_p \twoheadrightarrow t_i, \ldots, s_n \twoheadrightarrow t_n \in \mathcal{R}'$, for some $p \in \mathcal{P}os(s_i)$, $1 \leqslant i \leqslant n$. Since $s_i\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}_{k_i}} t_i\sigma$, $k_i < k$, $i = 1, \ldots, n$, by the induction hypothesis, we have $s_i\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t_i\sigma$, $i = 1, \ldots, n$. By Lemma 2.24, there exists $\sigma' = \{w \mapsto s'\}$ for some normal form s' such that $s_i|_p\sigma = s_i|_p\sigma\sigma' \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}_{k_i}} w\sigma\sigma' = w\sigma'$ and $s_i[w]_p\sigma\sigma' = s_i\sigma[w\sigma']_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}_{k_i}} t_i$. Moreover, since w is an extra variable, we also have $s_j\sigma\sigma' = s_j\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t_j\sigma = t_j\sigma\sigma'$ for $j = 1, \ldots, i-1, i+1, \ldots, n$. Therefore, since $l\sigma\sigma' = l\sigma$ and $r\sigma\sigma' = r\sigma$, we have $s[l\sigma]_u \xrightarrow{\mathsf{i}}_{\mathcal{R}} s[r\sigma]_u$, and the claim follows by induction. Consider the second case. By the induction hypothesis, we have $s[r\sigma]_u \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t$ and $s_i\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t_i\sigma$ for all $i = 1, \ldots, n$. By Lemma 2.24, there exists a substitution $\sigma' = \{w \mapsto s'\}$ such that s' is the normal form of $r|_q\sigma$ and we have $r|_q\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} w\sigma'$ and $s[r\sigma[w\sigma']_q]_u \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t$. Moreover, since w is a fresh variable, we have $s_i\sigma\sigma' \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t_i\sigma\sigma'$ for all $i = 1, \ldots, n$. Therefore, we have $s[l\sigma\sigma']_u = s[l\sigma]_u \xrightarrow{\mathsf{i}}_{\mathcal{R}'} s[r\sigma[w\sigma']_q]_u$, which concludes the proof.

($\Leftarrow$) This direction is perfectly analogous to the previous one, and follows easily by Lemma 2.24 too. □

Let us now consider the second kind of transformations: the simplification of constructor conditions. Basically, we can drop an equation $s \twoheadrightarrow t$ when the terms s and t are constructor, called a *constructor condition*, by either applying the *most general unifier* (mgu) of s and t (if it exists) to the remaining part of the rule, or by deleting entirely the rule if they do not unify because (under innermost rewriting) the equation will never be satisfied by any normalized substitution. Similar transformations can be found in [111].

In order to justify these transformations, we state and prove the following results. In the following, we let $mgu(s, t)$ denote the most general unifier of terms s and t if it exists, and *fail* otherwise.

Theorem 2.26 (removal of unifiable constructor conditions). *Let $\mathcal{R}$ be a DCTRS and let $R = (l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n}) \in \mathcal{R}$ be a rule with $mgu(s_i, t_i) = \theta$, for some $i \in \{1, \ldots, n\}$, where s_i and t_i are constructor terms. Let R' be a new rule of the form*

$l\theta \to r\theta \Leftarrow s_1\theta \twoheadrightarrow t_1\theta, \ldots, s_{i-1}\theta \twoheadrightarrow t_{i-1}\theta, s_{i+1}\theta \twoheadrightarrow t_{i+1}\theta, \ldots, s_n\theta \twoheadrightarrow t_n\theta$.[9] *Then* $\mathcal{R}' = (\mathcal{R}\backslash\{R\}) \cup \{R'\}$ *is a DCTRS and, for all ground terms* s *and* t*, we have* $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$ *iff* $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t$.

Proof. ($\Rightarrow$) First, we prove the following claim by induction on the lexicographic product (k, m) of the depth k and the length m of the steps: if $s \xrightarrow{\mathsf{i}}{}^m_{\mathcal{R}_k} t$, then $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t$. It suffices to consider the case where R is applied, i.e., $s = s[l\sigma]_p \xrightarrow{\mathsf{i}}_{\{R\}} s[r\sigma]_p$ with $s_j\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}_{k_j}} t_j\sigma$ for all $j \in \{1, \ldots, n\}$. By definition, σ is normalized. Hence, since s_i and t_i are constructor terms, we have that $s_i\sigma$ and $t_i\sigma$ are trivially normal forms since the normalized subterms introduced by σ cannot become reducible in a constructor context. Therefore, we have $s_i\sigma = t_i\sigma$. Thus, σ is a unifier of s_i and t_i and, hence, θ is more general than σ. Let δ be a substitution such that $\sigma = \theta\delta$. Since σ is normalized, so is δ. Since $k_j < k$ for all $j = 1, \ldots, n$, by the induction hypothesis, we have that $s_j\sigma = s_j\theta\delta \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t_j\theta\delta = t_j\sigma$ for $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$. Therefore, we have that $s[l\sigma]_p = s[l\theta\delta]_p \xrightarrow{\mathsf{i}}_{\{R'\}} s[r\theta\delta]_p = s[r\sigma]_p$.

($\Leftarrow$) Now, we prove the following claim by induction on the lexicographic product (k, m) of the depth k and the length m of the steps: if $s \xrightarrow{\mathsf{i}}{}^m_{\mathcal{R}'_k} t$, then $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$. It suffices to consider the case where R' is applied, i.e., $s = s[l\theta\delta]_p \xrightarrow{\mathsf{i}}_{\{R\}} s[r\theta\delta]_p$ with $s_j\theta\delta \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'_{k_j}} t_j\theta\delta$ for all $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$. By the assumption and the definition, θ and δ are normalized, and thus, $s_i\theta\delta$ and $t_i\theta\delta$ are normal forms (as in the previous case, because the normalized subterms introduced by $\theta\delta$ cannot become reducible in a constructor context), i.e., $s_i\theta\delta = t_i\theta\delta$. Since $k_j < k$ for all $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$, by the induction hypothesis, we have that $s_j\theta\delta \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t_j\theta\delta$ for $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$. Therefore, we have that $s[l\sigma]_p = s[l\theta\delta]_p \xrightarrow{\mathsf{i}}_{\{R\}} s[r\theta\delta]_p = s[r\sigma]$ with $\sigma = \theta\delta$. □

Now we consider the case when the terms in the constructor condition do not unify:

Theorem 2.27 (removal of infeasible rules)**.** *Let* $\mathcal{R}$ *be a DCTRS and let* $R = (l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n}) \in \mathcal{R}$ *be a rule with* $mgu(s_i, t_i) = \mathit{fail}$*, for some* $i \in \{1, \ldots, n\}$*. Then* $\mathcal{R}' = \mathcal{R}\backslash\{R\}$ *is a DCTRS and, for all ground terms* s *and* t*, we have* $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$ *iff* $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t$.

[9] In [111], the condition $\mathcal{D}\mathsf{om}(\theta) \cap \mathcal{V}ar(l, r, s_1, t_1, \ldots, s_n, t_n) = \varnothing$ is required, but this condition is not really necessary.

Proof. Since $\mathcal{R} \supseteq \mathcal{R}'$, the *if* part is trivial, and thus, we consider the *only-if* part. To apply R to a term, there must exist a normalized substitution σ such that $s_i\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t_i\sigma$. Since s_i, t_i are constructor terms and σ is normalized, $s_i\sigma$ is a normal form (because the normalized subterms introduced by σ cannot become reducible in a constructor context). If $s_i\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t_i\sigma$ is satisfied (i.e., $s_i\sigma = t_i\sigma$), then s_i and t_i are unifiable, and thus, this contradicts the assumption. Therefore, R is never applied to any term, and hence, $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t$ iff $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} t$. □

Using flattening and the simplification of constructor conditions, any constructor DCTRS with constructor terms in the right-hand sides of the equations of the rules can be transformed into a pcDCTRS. One can use, for instance, the following simple algorithm. Let $\mathcal{R}$ be such a constructor DCTRS. We apply the following transformations as much as possible:

(flattening-rhs) Assume that $\mathcal{R}$ contains a rule of the form $R = (l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n})$ where r is not a constructor term. Let $r|_q$, $q \in \mathcal{P}os(r)$, be a basic subterm of r. Then, we replace rule R by a new rule of the form $l \to r[w]_q \Leftarrow \overline{s_n \twoheadrightarrow t_n}, r|_q \twoheadrightarrow w$, where w is a fresh variable.

(flattening-condition) Assume that $\mathcal{R}$ contains a rule of the form $R = (l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n})$ where s_i is neither a constructor term nor a basic term, $i \in \{1, \ldots, n\}$. Let $s_i|_q$, $q \in \mathcal{P}os(s_1)$, be a basic subterm of s_i. Then, we replace rule R by a new rule of the form $l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_i|_q \twoheadrightarrow w, s_i[w]_q \twoheadrightarrow t_i, \ldots, s_n \twoheadrightarrow t_n$, where w is a fresh variable.

(removal-unify) Assume that $\mathcal{R}$ contains a rule of the form $R = (l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n})$ where s_i is a constructor term, $i \in \{1, \ldots, n\}$. If $mgu(s_i, t_i) = \theta \neq \mathit{fail}$, then we replace rule R by a new rule of the form $l\theta \to r\theta \Leftarrow s_1\theta \twoheadrightarrow t_1\theta, \ldots, s_{i-1}\theta \twoheadrightarrow t_{i-1}\theta, s_{i+1}\theta \twoheadrightarrow t_{i+1}\theta, \ldots, s_n\theta \twoheadrightarrow t_n\theta$.

(removal-fail) Assume that $\mathcal{R}$ contains a rule of the form $R = (l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n})$ where s_i is a constructor term, $i \in \{1, \ldots, n\}$. If $mgu(s_i, t_i) = \mathit{fail}$, then we remove rule R from $\mathcal{R}$.

Trivially, by applying rule flattening-rhs as much as possible, we end up with a DCTRS where all the right-hand sides are constructor terms; analogously, the exhaustive application of rule flattening-condition allows us to ensure that the left-hand sides of all equations in the conditions of the rules are either constructor or basic; finally, the application of rules removal-unify and removal-fail produces a

pcDCTRS by removing those equations in which the left-hand side is a constructor term. Therefore, in the remainder of this paper, we only consider pcDCTRSs.

A nice property of pcDCTRSs is that one can consider reductions only at *topmost* positions. Formally, given a pcDCTRS $\mathcal{R}$, we say that $s \to_{p,l\to r\Leftarrow\overline{s_n\twoheadrightarrow t_n}} t$ is a *top* reduction step if $p = \epsilon$, there is a ground substitution σ with $s = l\sigma$, $s_i\sigma \to^*_{\mathcal{R}} t_i\sigma$ for all $i = 1, \ldots, n$, $t = r\sigma$, and all the steps in $s_i\sigma \to^*_{\mathcal{R}} t_i\sigma$ for $i = 1, \ldots, n$ are also top reduction steps. We denote top reductions with $\xrightarrow{\epsilon}$ for standard rewriting, and $\xrightarrow{\epsilon}_{\mathcal{R}}$, $\xleftarrow{\epsilon}_{\mathcal{R}}$ for our reversible rewrite relations.

The following result basically states that $\xrightarrow{\mathsf{i}}$ and $\xrightarrow{\epsilon}$ are equivalent for pcDCTRSs:

Theorem 2.28. *Let $\mathcal{R}$ be a constructor DCTRS with constructor terms in the right-hand sides of the equations and $\mathcal{R}'$ be a pcDCTRS obtained from $\mathcal{R}$ by a sequence of transformations of flattening and simplification of constructor conditions. Given ground terms s and t such that s is basic and t is normalized, we have $s \xrightarrow{\mathsf{i}}^*_{\mathcal{R}} t$ iff $s \xrightarrow{\epsilon}^*_{\mathcal{R}'} t$.*

Proof. First, it is straightforward to see that an innermost reduction in $\mathcal{R}'$ can only reduce the topmost positions of terms since defined functions can only occur at the root of terms and the terms introduced by instantiation are, by definition, irreducible. Therefore, the claim is a consequence of Theorems 2.25, 2.26 and 2.27, together with the above fact. □

Therefore, when considering pcDCTRSs and top reductions, storing the reduced positions in the trace terms becomes redundant since they are always ϵ. Thus, in practice, one can consider simpler trace terms without positions, $\beta(\sigma, \pi_1, \ldots, \pi_n)$, that implicitly represent the trace term $\beta(\epsilon, \sigma, \pi_1, \ldots, \pi_n)$.

Example 2.29. Consider the following TRS $\mathcal{R}$ defining addition and multiplication on natural numbers, and its associated pcDCTRS $\mathcal{R}'$:

$$\begin{aligned}\mathcal{R} = \{ \quad & \mathsf{add}(\mathsf{0}, y) \to y,\\ & \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)),\\ & \mathsf{mult}(\mathsf{0}, y) \to \mathsf{0},\\ & \mathsf{mult}(\mathsf{s}(x), y) \to \mathsf{add}(\mathsf{mult}(x, y), y)\}\end{aligned}$$

$$\begin{aligned}\mathcal{R}' = \{ \quad & \mathsf{add}(\mathsf{0}, y) \to y,\\ & \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(z) \Leftarrow \mathsf{add}(x, y) \twoheadrightarrow z,\\ & \mathsf{mult}(\mathsf{0}, y) \to \mathsf{0},\\ & \mathsf{mult}(\mathsf{s}(x), y) \to w \Leftarrow \mathsf{mult}(x, y) \twoheadrightarrow z, \mathsf{add}(z, y) \twoheadrightarrow w\}\end{aligned}$$

For instance, given the following reduction in $\mathcal{R}$:

$$\mathsf{mult}(\mathsf{s}(0),\mathsf{s}(0)) \xrightarrow{\mathsf{i}}_{\mathcal{R}} \mathsf{add}(\mathsf{mult}(0,\mathsf{s}(0)),\mathsf{s}(0)) \xrightarrow{\mathsf{i}}_{\mathcal{R}} \mathsf{add}(0,\mathsf{s}(0)) \xrightarrow{\mathsf{i}}_{\mathcal{R}} \mathsf{s}(0)$$

we have the following counterpart in $\mathcal{R}'$:

$$\mathsf{mult}(\mathsf{s}(0),\mathsf{s}(0)) \xrightarrow{\epsilon}_{\mathcal{R}'} \mathsf{s}(0) \quad \text{with } \mathsf{mult}(0,\mathsf{s}(0)) \xrightarrow{\epsilon}_{\mathcal{R}'} 0 \text{ and } \mathsf{add}(0,\mathsf{s}(0)) \xrightarrow{\epsilon}_{\mathcal{R}'} \mathsf{s}(0)$$

Trivially, all results in Section 2.3 hold for pcDCTRSs and top reductions starting from basic terms. The simpler trace terms without positions will allow us to introduce appropriate injectivization and inversion transformations in the next section.

2.5 Reversibilization

In this section, we aim at *compiling* the reversible extension of rewriting into the system rules. Intuitively speaking, given a pure constructor system $\mathcal{R}$, we aim at producing new systems $\mathcal{R}_f$ and $\mathcal{R}_b$ such that standard rewriting in $\mathcal{R}_f$, i.e., $\to_{\mathcal{R}_f}$, coincides with the forward reversible extension $\rightharpoonup_{\mathcal{R}}$ in the original system, and analogously $\to_{\mathcal{R}_b}$ is equivalent to $\leftharpoondown_{\mathcal{R}}$. Therefore, $\mathcal{R}_f$ can be seen as an *injectivization* of $\mathcal{R}$, and $\mathcal{R}_b$ as the *inversion* of $\mathcal{R}_f$.

In principle, we could easily introduce a transformation for pcDCTRSs that mimicks the behavior of the reversible extension of rewriting. For instance, given the pcDCTRS $\mathcal{R}$ of Example 2.16, we could produce the following injectivized version $\mathcal{R}_f$:[10]

$$\begin{aligned}
\langle \mathsf{f}(x,y,m), ws\rangle &\to \langle \mathsf{s}(w), \beta_1(m,x,w_1,w_2) : ws\rangle \\
&\quad \Leftarrow \langle \mathsf{h}(x), [\,]\rangle \twoheadrightarrow \langle x, w_1\rangle, \langle \mathsf{g}(y,4), [\,]\rangle \twoheadrightarrow \langle w, w_2\rangle \\
\langle \mathsf{h}(0), ws\rangle &\to \langle 0, \beta_2 : ws\rangle \\
\langle \mathsf{h}(1), ws\rangle &\to \langle 1, \beta_3 : ws\rangle \\
\langle \mathsf{g}(x,y), ws\rangle &\to \langle x, \beta_4(y) : ws\rangle
\end{aligned}$$

For instance, the reversible step $\langle \mathsf{f}(0,2,4), [\,]\rangle \xrightarrow{\epsilon}_{\mathcal{R}} \langle \mathsf{s}(2), [\beta_1(\sigma', \pi_1, \pi_2)]\rangle$ with $\sigma' = \{m \mapsto 4, x \mapsto 0\}$, $\pi_1 = [\beta_2(id)]$ and $\pi_2 = [\beta_4(\{y \mapsto 4\})]$, has the following counterpart in $\mathcal{R}_f$:

$$\langle \mathsf{f}(0,2,4), [\,]\rangle \xrightarrow{\epsilon}_{\mathcal{R}_f} \langle \mathsf{s}(2), [\beta_1(4,0,[\beta_2],[\beta_4(4)])]\rangle$$
$$\text{with } \langle \mathsf{h}(0), [\,]\rangle \xrightarrow{\epsilon}_{\mathcal{R}_f} \langle 0, [\beta_2]\rangle \text{ and } \langle \mathsf{g}(2,4), [\,]\rangle \xrightarrow{\epsilon}_{\mathcal{R}_f} \langle 2, [\beta_4(4)]\rangle$$

[10] We will write just β instead of $\beta()$ when no argument is required.

The only subtle difference here is that a trace term like

$$\beta_1(\{m \mapsto \mathsf{4}, x \mapsto \mathsf{0}\}, [\beta_2(id)], [\beta_4(\{y \mapsto \mathsf{4}\})])$$

is now stored in the transformed system as

$$\beta_1(\mathsf{4}, \mathsf{0}, [\beta_2], [\beta_4(\mathsf{4})])$$

Furthermore, we could produce an inverse $\mathcal{R}_b$ of the above system as follows:

$$\begin{aligned}
\langle \mathsf{s}(w), \beta_1(m, x, w_1, w_2) : ws\rangle^{-1} \to \langle \mathsf{f}(x, y, m), ws\rangle^{-1} & \\
\Leftarrow \langle w, w_2\rangle^{-1} \twoheadrightarrow \langle \mathsf{g}(y, \mathsf{4}), [\,]\rangle^{-1}, & \\
\langle x, w_1\rangle^{-1} \twoheadrightarrow \langle \mathsf{h}(x), [\,]\rangle^{-1} & \\
\langle \mathsf{0}, \beta_2 : ws\rangle^{-1} \to \langle \mathsf{h}(\mathsf{0}), ws\rangle^{-1} & \\
\langle \mathsf{1}, \beta_3 : ws\rangle^{-1} \to \langle \mathsf{h}(\mathsf{1}), ws\rangle^{-1} & \\
\langle x, \beta_4(y) : ws\rangle^{-1} \to \langle \mathsf{g}(x, y), ws\rangle^{-1} &
\end{aligned}$$

mainly by switching the left- and right-hand sides of each rule and condition. The correctness of these injectivization and inversion transformations would be straightforward.

These transformations are only aimed at mimicking, step by step, the reversible relations $\rightharpoonup_{\mathcal{R}}$ and $\leftharpoondown_{\mathcal{R}}$. Roughly speaking, for each step $\langle s, \pi\rangle \rightharpoonup_{\mathcal{R}} \langle t, \pi'\rangle$ in a system $\mathcal{R}$, we have $\langle s, \pi\rangle \to_{\mathcal{R}_f} \langle t, \pi'\rangle$, where $\mathcal{R}_f$ is the injectivized version of $\mathcal{R}$, and for each step $\langle s, \pi\rangle \leftharpoondown_{\mathcal{R}} \langle t, \pi'\rangle$ in $\mathcal{R}$, we have $\langle s, \pi\rangle \to_{\mathcal{R}_b} \langle t, \pi'\rangle$, where $\mathcal{R}_b$ is the inverse of $\mathcal{R}_f$. More details on this approach can be found in [108]. Unfortunately, it might be much more useful to produce injective and inverse versions of *each function* defined in a system $\mathcal{R}$. Note that, in the above approach, the system $\mathcal{R}_f$ only defines a single function $\langle \_, \_\rangle$ and $\mathcal{R}_b$ only defines $\langle \_, \_\rangle^{-1}$, i.e., we are computing systems that define the relations $\rightharpoonup_{\mathcal{R}}$ and $\leftharpoondown_{\mathcal{R}}$ rather than the injectivized and inverse versions of the functions in $\mathcal{R}$. In the following, we introduce more refined transformations that can actually produce injective and inverse versions of the original functions.

2.5.1 Injectivization

In principle, given a function f, one can consider that the injectivization of a rule of the form[11]

$$\beta : \mathsf{f}(\overline{s_0}) \to r \Leftarrow \mathsf{f}_1(\overline{s_1}) \twoheadrightarrow t_1, \ldots, \mathsf{f}_n(\overline{s_n}) \twoheadrightarrow t_n$$

[11] By abuse of notation, here we let $\overline{s_0}, \ldots, \overline{s_n}$ denote sequences of terms of arbitrary length, i.e., $\overline{s_0} = s_{0,1}, \ldots, s_{0,l_0}$, $\overline{s_1} = s_{1,1}, \ldots, s_{1,l_1}$, etc.

produces the following rule

$$\mathsf{f}^{\mathsf{i}}(\overline{s_0}) \to \langle r, \beta(\overline{y}, \overline{w_n})\rangle \Leftarrow \mathsf{f}^{\mathsf{i}}_1(\overline{s_1}) \twoheadrightarrow \langle t_1, w_1\rangle \ldots, \mathsf{f}^{\mathsf{i}}_n(\overline{s_n}) \twoheadrightarrow \langle t_n, w_n\rangle$$

where $\{\overline{y}\} = (\mathcal{V}\mathsf{ar}(l)\backslash\mathcal{V}\mathsf{ar}(r, \overline{s_n}, \overline{t_n}))\cup\bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(t_i)\backslash\mathcal{V}\mathsf{ar}(r, \overline{s_{i+1,n}})$ and $\overline{w_n}$ are fresh variables. The following example, though, illustrates that this is not correct in general.

Example 2.30. Consider the following pcDCTRS $\mathcal{R}$:

$$\begin{array}{lrcl} \beta_1: & \mathsf{f}(x,y) & \to & z \Leftarrow \mathsf{h}(y) \twoheadrightarrow w,\ \mathsf{first}(x,w) \twoheadrightarrow z \\ \beta_2: & \mathsf{h}(0) & \to & 0 \\ \beta_3: & \mathsf{first}(x,y) & \to & x \end{array}$$

together with the following top reduction:

$$\begin{array}{ll} \mathsf{f}(2,1) \xrightarrow{\epsilon}_{\mathcal{R}} 2 & \text{with } \sigma = \{x \mapsto 2, y \mapsto 1, w \mapsto \mathsf{h}(1), z \mapsto 2\} \\ & \text{where } \mathsf{h}(y)\sigma = \mathsf{h}(1) \xrightarrow{\epsilon}{}^{*}_{\mathcal{R}} \mathsf{h}(1) = w\sigma \\ & \text{and } \mathsf{first}(x,w)\sigma = \mathsf{first}(2,\mathsf{h}(1)) \xrightarrow{\epsilon}_{\mathcal{R}} 2 = z\sigma \end{array}$$

Following the scheme above, we would produce the following pcDCTRS

$$\begin{array}{rl} \mathsf{f}^{\mathsf{i}}(x,y) \to \langle z, \beta_1(w_1,w_2)\rangle \Leftarrow \mathsf{h}^{\mathsf{i}}(y) \twoheadrightarrow \langle w, w_1\rangle,\ \mathsf{first}^{\mathsf{i}}(x,w) \twoheadrightarrow \langle z, w_2\rangle \\ \mathsf{h}^{\mathsf{i}}(0) \to \langle 0, \beta_2\rangle \\ \mathsf{first}^{\mathsf{i}}(x,y) \to \langle x, \beta_3(y)\rangle \end{array}$$

Unfortunately, the corresponding reduction for $\mathsf{f}^{\mathsf{i}}(2,1)$ above cannot be done in this system since $\mathsf{h}^{\mathsf{i}}(1)$ cannot be reduced to $\langle \mathsf{h}^{\mathsf{i}}(1), [\,]\rangle$.

In order to overcome this drawback, one could *complete* the function definitions with rules that reduce each irreducible term t to a tuple of the form $\langle t, [\,]\rangle$. Although we find it a promising idea for future work, in this paper we propose a simpler approach. In the following, we consider a refinement of innermost reduction where only constructor substitutions are computed. Formally, the constructor reduction relation, $\xrightarrow{\mathsf{c}}$, is defined as follows: given ground terms $s, t \in \mathcal{T}(\mathcal{F})$, we have $s \xrightarrow{\mathsf{c}}_{\mathcal{R}} t$ iff there exist a position p in s such that no proper subterms of $s|_p$ are reducible, a rewrite rule $l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$, and a ground *constructor* substitution σ such that $s|_p = l\sigma$, $s_i\sigma \xrightarrow{\mathsf{c}}{}^{*}_{\mathcal{R}} t_i\sigma$ for all $i = 1, \ldots, n$, and $t = s[r\sigma]_p$. Note that the results in the previous section also hold for $\xrightarrow{\mathsf{c}}$.

In the following, given a basic term $t = \mathsf{f}(\overline{s})$, we denote by t^{i} the term $\mathsf{f}^{\mathsf{i}}(\overline{s})$. Now, we introduce our injectivization transformation as follows:

Definition 2.31 (injectivization)**.** Let $\mathcal{R}$ be a pcDCTRS. We produce a new CTRS $\mathbf{I}(\mathcal{R})$ by replacing each rule $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n}$ of $\mathcal{R}$ by a new rule of the form

$$l^{\mathtt{i}} \to \langle r, \beta(\overline{y}, \overline{w_n})\rangle \Leftarrow \overline{s_n^{\mathtt{i}} \twoheadrightarrow \langle t_n, w_n\rangle}$$

in $\mathbf{I}(\mathcal{R})$, where $\{\overline{y}\} = (\mathcal{V}\mathsf{ar}(l)\backslash\mathcal{V}\mathsf{ar}(r, \overline{s_n}, \overline{t_n})) \cup \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(t_i)\backslash\mathcal{V}\mathsf{ar}(r, \overline{s_{i+1,n}})$ and $\overline{w_n}$ are fresh variables. Here, we assume that the variables of $\overline{y}$ are in lexicographic order.

Observe that now we do not need to keep a trace in each term, but only a single trace term since all reductions finish in one step in a pcDCTRS. The relation between the original trace terms and the information stored in the injectivized system is formalized as follows:

Definition 2.32. Given a trace term $\pi = \beta(\{\overline{y_m \mapsto t_m}\}, \pi_1, \ldots, \pi_n)$, we define $\widehat{\pi}$ recursively as follows: $\widehat{\pi} = \beta(\overline{t_m}, \overline{\widehat{\pi_1}}, \ldots, \overline{\widehat{\pi_n}})$, where we assume that the variables $\overline{y_m}$ are in lexicographic order.

Moreover, in order to simplify the notation, we consider that a a trace term π and a singleton list of the form $[\pi]$ denote the same object. The correctness of the injectivization transformation is stated as follows:

Theorem 2.33. *Let $\mathcal{R}$ be a pcDCTRS and $\mathcal{R}_f = \mathbf{I}(\mathcal{R})$ be its injectivization. Then $\mathcal{R}_f$ is a pcDCTRS and, given a basic ground term s, we have $\langle s, [\,]\rangle \xrightarrow{\mathsf{c}}_{\mathcal{R}} \langle t, \pi\rangle$ iff $s^{\mathtt{i}} \xrightarrow{\mathsf{c}}_{\mathcal{R}_f} \langle t, \widehat{\pi}\rangle$.*

Proof. The fact that $\mathcal{R}_f$ is a pcDCTRS is trivial. Regarding the second part, we proceed as follows:

($\Rightarrow$) We proceed by induction on the depth k of the step $\langle s, [\,]\rangle \xrightarrow{\mathsf{c}}_{\mathcal{R}_k} \langle t, \pi\rangle$. Since the depth $k = 0$ is trivial, we consider the inductive case $k > 0$. Thus, there is a rule $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$, and a substitution σ such that $s = l\sigma$, $\langle s_i\sigma, [\,]\rangle \xrightarrow{\mathsf{c}}_{\mathcal{R}_{k_i}} \langle t_i\sigma, \pi_i\rangle, i = 1, \ldots, n, t = r\sigma, \sigma' = \sigma\restriction_{(\mathcal{V}\mathsf{ar}(l)\backslash\mathcal{V}\mathsf{ar}(r,\overline{s_n},\overline{t_n}))\cup\bigcup_{i=1}^{n}\mathcal{V}\mathsf{ar}(t_i)\backslash\mathcal{V}\mathsf{ar}(r,\overline{s_{i+1,n}})}$, and $\pi = \beta(\sigma', \pi_1, \ldots, \pi_n)$. By definition of $\rightharpoonup_{\mathcal{R}_k}$, we have that $k_i < k$ for all $i = 1, \ldots, n$ and, thus, by the induction hypothesis, we have $(s_i\sigma)^{\mathtt{i}} \xrightarrow{\mathsf{c}}_{\mathcal{R}_f} \langle t_i\sigma, \widehat{\pi}_i\rangle$ for all $i = 1, \ldots, n$. Consider now the equivalent rule in $\mathcal{R}_f$: $l^{\mathtt{i}} \to \langle r, \beta(\overline{y}, \overline{w_n})\rangle \Leftarrow s_1^{\mathtt{i}} \twoheadrightarrow \langle t_1, w_1\rangle, \ldots, s_n^{\mathtt{i}} \twoheadrightarrow \langle t_n, w_n\rangle$. Therefore, we have $s^{\mathtt{i}} \xrightarrow{\mathsf{c}}_{\mathcal{R}_f} \langle t, \beta(\overline{y}\sigma, \widehat{\pi}_1, \ldots, \widehat{\pi}_n)\rangle$ where $\{\overline{y}\} = (\mathcal{V}\mathsf{ar}(l)\backslash\mathcal{V}\mathsf{ar}(r, \overline{s_n}, \overline{t_n})) \cup \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(t_i)\backslash\mathcal{V}\mathsf{ar}(r, \overline{s_{i+1,n}})$ and, thus, we can conclude that $\widehat{\pi} = \beta(\overline{y}\sigma, \widehat{\pi}_1, \ldots, \widehat{\pi}_n)$.

($\Leftarrow$) This direction is analogous. We proceed by induction on the depth k of the step $s^{\mathtt{i}} \xrightarrow{\mathsf{c}}_{\mathcal{R}_{f_k}} \langle t, \widehat{\pi}\rangle$. Since the depth $k = 0$ is trivial, we consider the inductive

case $k > 0$. Thus, there is a rule $l^{\mathsf{i}} \to \langle r, \beta(\overline{y}, \overline{w_n})\rangle \Leftarrow s_1^{\mathsf{i}} \twoheadrightarrow \langle t_1, w_1\rangle, \ldots, s_n^{\mathsf{i}} \twoheadrightarrow \langle t_n, w_n\rangle$ in $\mathcal{R}_f$ and a substitution θ such that $l^{\mathsf{i}}\theta = s^{\mathsf{i}}$, $s_i^{\mathsf{i}}\theta \xrightarrow{\mathsf{c}}_{\mathcal{R}_{f_{k_i}}} \langle t_i, w_i\rangle\theta$, $i = 1, \ldots, n$, and $\langle r, \beta(\overline{y}, \overline{w_n})\rangle\theta = \langle t, \widehat{\pi}\rangle$. Assume that σ is the restriction of θ to the variables of the rule, excluding the fresh variables $\overline{w_n}$, and that $w_i\theta = \widehat{\pi}_i$ for all $i = 1, \ldots, n$. Therefore, $\langle s_i, [\,]\rangle\theta = \langle s_i\sigma, [\,]\rangle$ and $\langle t_i, w_i\rangle\theta = \langle t_i\sigma, \widehat{\pi}_i\rangle$, $i = 1, \ldots, n$. Then, by definition of $\mathcal{R}_{f_{k_i}}$, we have that $k_i < k$ for all $i = 1, \ldots, n$ and, thus, by the induction hypothesis, we have $\langle s_i\sigma, [\,]\rangle \xrightarrow{\mathsf{c}}_{\mathcal{R}} \langle t_i\sigma, \pi_i\rangle$, $i = 1, \ldots, n$. Consider now the equivalent rule in $\mathcal{R}$: $\beta : l \to r \Leftarrow \overline{s_n \twoheadrightarrow t_n} \in \mathcal{R}$. Therefore, we have $\langle s, [\,]\rangle \xrightarrow{\mathsf{c}}_{\mathcal{R}} \langle t, \pi\rangle$, $\sigma' = \sigma|_{(\mathcal{V}\mathsf{ar}(l)\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_n},\overline{t_n}))\cup\bigcup_{i=1}^{n}\mathcal{V}\mathsf{ar}(t_i)\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_{i+1,n}})}$, and $\pi = \beta(\sigma', \pi_1, \ldots, \pi_n)$. Finally, since $\{\overline{y}\} = (\mathcal{V}\mathsf{ar}(l)\setminus\mathcal{V}\mathsf{ar}(r, \overline{s_n}, \overline{t_n})) \cup \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(t_i)\setminus\mathcal{V}\mathsf{ar}(r, \overline{s_{i+1,n}})$, we can conclude that $\widehat{\pi} = \pi$. □

2.5.2 Inversion

Given an injectivized system, inversion basically amounts to switching the left- and right-hand sides of the rule and of every equation in the condition, as follows:

Definition 2.34 (inversion)**.** Let $\mathcal{R}$ be a pcDCTRS and $\mathcal{R}_f = \mathbf{I}(\mathcal{R})$ be its injectivization. The inverse system $\mathcal{R}_b = \mathbf{I}^{-1}(\mathcal{R}_f)$ is obtained from $\mathcal{R}_f$ by replacing each rule[12]

$$\mathsf{f}^{\mathsf{i}}(\overline{s_0}) \to \langle r, \beta(\overline{y}, \overline{w_n})\rangle \Leftarrow \mathsf{f}_1^{\mathsf{i}}(\overline{s_1}) \twoheadrightarrow \langle t_1, w_1\rangle, \ldots, \mathsf{f}_n^{\mathsf{i}}(\overline{s_n}) \twoheadrightarrow \langle t_n, w_n\rangle$$

of $\mathcal{R}_f$ by a new rule of the form

$$\mathsf{f}^{-1}(r, \beta(\overline{y}, \overline{w_n})) \to \langle \overline{s_0}\rangle \Leftarrow \mathsf{f}_n^{-1}(t_n, w_n) \twoheadrightarrow \langle \overline{s_n}\rangle, \ldots, \mathsf{f}_1^{-1}(t_1, w_1) \twoheadrightarrow \langle \overline{s_1}\rangle$$

in $\mathbf{I}^{-1}(\mathcal{R}_f)$, where the variables of $\overline{y}$ are in lexicographic order.

Example 2.35. Consider again the pcDCTRS of Example 2.16. Here, injectivization returns the following pcDCTRS $\mathbf{I}(\mathcal{R}) = \mathcal{R}_f$:

$$\begin{aligned}
\mathsf{f}^{\mathsf{i}}(x, y, m) &\to \langle \mathsf{s}(w), \beta_1(m, x, w_1, w_2)\rangle \\
&\qquad \Leftarrow \mathsf{h}^{\mathsf{i}}(x) \twoheadrightarrow \langle x, w_1\rangle, \mathsf{g}^{\mathsf{i}}(y, \mathsf{4}) \twoheadrightarrow \langle w, w_2\rangle \\
\mathsf{h}^{\mathsf{i}}(\mathsf{0}) &\to \langle \mathsf{0}, \beta_2\rangle \\
\mathsf{h}^{\mathsf{i}}(\mathsf{1}) &\to \langle \mathsf{1}, \beta_3\rangle \\
\mathsf{g}^{\mathsf{i}}(x, y) &\to \langle x, \beta_4(y)\rangle
\end{aligned}$$

[12] Here, we assume that $\overline{s_0}$, $\overline{s_1}$,…, $\overline{s_n}$ denote arbitrary sequences of terms, i.e., $\overline{s_0} = s_{0,1}, \ldots, s_{0,l_0}$, $\overline{s_1} = s_{1,1}, \ldots, s_{1,l_1}$, etc. We use this notation for clarity.

Then, inversion with $\mathbf{I}^{-1}$ produces the following pcDCTRS $\mathbf{I}^{-1}(\mathbf{I}(\mathcal{R})) = \mathcal{R}_b$:

$$\begin{aligned}\mathsf{f}^{-1}(\mathsf{s}(w), \beta_1(m, x, w_1, w_2)) &\to \langle x, y, m\rangle \\ &\Leftarrow \mathsf{g}^{-1}(w, w_2) \twoheadrightarrow \langle y, 4\rangle, \mathsf{h}^{-1}(x, w_1) \twoheadrightarrow \langle x\rangle \\ \mathsf{h}^{-1}(0, \beta_2) &\to \langle 0\rangle \\ \mathsf{h}^{-1}(1, \beta_3) &\to \langle 1\rangle \\ \mathsf{g}^{-1}(x, \beta_4(y)) &\to \langle x, y\rangle\end{aligned}$$

Finally, the correctness of the inversion transformation is stated as follows:

Theorem 2.36. *Let $\mathcal{R}$ be a pcDCTRS, $\mathcal{R}_f = \mathbf{I}(\mathcal{R})$ its injectivization, and $\mathcal{R}_b = \mathbf{I}^{-1}(\mathcal{R}_f)$ the inversion of $\mathcal{R}_f$. Then, $\mathcal{R}_b$ is a basic pcDCTRS and, given a basic ground term $\mathsf{f}(\overline{s})$ and a constructor ground term t with $\langle t, \pi\rangle$ a safe pair, we have $\langle t, \pi\rangle \overset{\mathsf{c}}{\leftarrow}_{\mathcal{R}} \langle \mathsf{f}(\overline{s}), [\,]\rangle$ iff $\mathsf{f}^{-1}(t, \widehat{\pi}) \overset{\mathsf{c}}{\to}_{\mathcal{R}_b} \langle \overline{s}\rangle$.*

Proof. The fact that $\mathcal{R}_f$ is a pcDCTRS is trivial. Regarding the second part, we proceed as follows.

($\Rightarrow$) We proceed by induction on the depth k of the step $\langle t, \pi\rangle \overset{\mathsf{c}}{\leftarrow}_{\mathcal{R}_k} \langle \mathsf{f}(\overline{s}), [\,]\rangle$. Since the depth $k = 0$ is trivial, we consider the inductive case $k > 0$. Let $\pi = \beta(\sigma', \overline{\pi_n})$. Thus, we have that $\langle t, \beta(\sigma', \overline{\pi_n})\rangle$ is a safe pair, there is a rule $\beta : \mathsf{f}(\overline{s_0}) \to r \Leftarrow \mathsf{f}_1(\overline{s_1}) \twoheadrightarrow t_1, \ldots, \mathsf{f}_n(\overline{s_n}) \twoheadrightarrow t_n$ and a substitution θ with $\mathcal{D}\mathsf{om}(\theta) = (\mathcal{V}\mathsf{ar}(r, \overline{s_1}, \ldots, \overline{s_n}) \backslash \mathcal{D}\mathsf{om}(\sigma'))$ such that $t = r\theta$, $\langle t_i\theta\sigma', \pi_i\rangle \overset{\mathsf{c}}{\to}_{\mathcal{R}_{k_i}} \langle \mathsf{f}(\overline{s_i})\theta\sigma', [\,]\rangle$ for all $i = 1, \ldots, n$, and $\mathsf{f}(\overline{s}) = \mathsf{f}(\overline{s_0})\theta\sigma'$. Note that $\overline{s_0}, \ldots, \overline{s_n}$ denote sequences of terms of arbitrary length, i.e., $\overline{s_0} = s_{0,1}, \ldots, s_{0,l_0}$, $\overline{s_1} = s_{1,1}, \ldots, s_{1,l_1}$, etc. Since $\langle t, \pi\rangle$ is a safe pair, we have that $\mathcal{D}\mathsf{om}(\sigma') = (\mathcal{V}\mathsf{ar}(\overline{s_0}) \backslash \mathcal{V}\mathsf{ar}(r, \overline{s_1}, \ldots, \overline{s_n}, \overline{t_n})) \cup \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(t_i) \backslash \mathcal{V}\mathsf{ar}(r, \overline{s_{i+1}}, \ldots, \overline{s_n})$. By definition of $\leftarrow_{\mathcal{R}_k}$, we have that $k_i < k$ for all $i = 1, \ldots, n$ and, by the induction hypothesis, we have $\mathsf{f}^{-1}(t_i\sigma, \widehat{\pi}_i) \overset{\mathsf{c}}{\to}_{\mathcal{R}_b} \langle \overline{s_i}\sigma\rangle$ for all $i = 1, \ldots, n$. Let us now consider the equivalent rule in $\mathcal{R}_b$:

$$\mathsf{f}^{-1}(r, \beta(\overline{y}, \overline{w_n}))) \to \langle \overline{s_0}\rangle \Leftarrow \mathsf{f}_n^{-1}(t_n, w_n) \twoheadrightarrow \langle \overline{s_n}\rangle, \ldots, \mathsf{f}_1^{-1}(t_1, w_1) \twoheadrightarrow \langle \overline{s_1}\rangle$$

Hence, we have $\mathsf{f}^{-1}(t, \beta(\overline{y}\sigma, \widehat{\pi}_1, \ldots, \widehat{\pi}_1)) \to_{\mathcal{R}_b} \langle \overline{s_0}\sigma\rangle = \langle \overline{s}\rangle$, where

$$\{\overline{y}\} = (\mathcal{V}\mathsf{ar}(\overline{s_0}) \backslash \mathcal{V}\mathsf{ar}(r, \overline{s_1}, \ldots, \overline{s_n}, \overline{t_n})) \cup \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(t_i) \backslash \mathcal{V}\mathsf{ar}(r, \overline{s_{i+1}}, \ldots, \overline{s_n})$$

and, thus, we can conclude that $\widehat{\pi} = \beta(\overline{y}\sigma, \widehat{\pi}_1, \ldots, \widehat{\pi}_n)$.

($\Leftarrow$) This direction is analogous. We proceed by induction on the depth k of the step $\mathsf{f}^{-1}(t, \widehat{\pi}) \overset{\mathsf{c}}{\to}_{\mathcal{R}_{b_k}} \langle \overline{s}\rangle$. Since the depth $k = 0$ is trivial, we consider the inductive case $k > 0$. Thus, there is a rule $\mathsf{f}^{-1}(r, \beta(\overline{y}, \overline{w_n}))) \to \langle \overline{s_0}\rangle \Leftarrow \mathsf{f}_n^{-1}(t_n, w_n) \twoheadrightarrow$

$\langle\overline{s_n}\rangle, \ldots, \mathsf{f}_1^{-1}(t_1, w_1) \twoheadrightarrow \langle\overline{s_1}\rangle$ in $\mathcal{R}_b$ and a substitution θ such that $\mathsf{f}^{-1}(r, \beta(\overline{y}, \overline{w_n}))\theta = \mathsf{f}^{-1}(t, \widehat{\pi}), \mathsf{f}_i^{-1}(t_i, w_i)\theta \xrightarrow{\mathsf{c}}_{\mathcal{R}_{b_{k_i}}} \langle\overline{s_i}\rangle\theta$, $i = n, \ldots, 1$, and $\mathsf{f}^{-1}(r, ws)\theta = \langle\overline{s}\rangle$. Assume that σ is the restriction of θ to the variables of the rule, excluding the fresh variables $\overline{w_n}$, and that $w_i\theta = \widehat{\pi}_i$ for all $i = 1, \ldots, n$. Therefore, $\mathsf{f}^{-1}(r, \beta(\overline{y}, \overline{w_n}))\theta = \mathsf{f}^{-1}(r\sigma, \beta(\overline{y}\sigma, \widehat{\pi}_1, \ldots, \widehat{\pi}_n), \mathsf{f}_i^{-1}(t_i, w_i)\theta = \mathsf{f}_i^{-1}(t_i\sigma, \widehat{\pi}_i)$ and $\langle\overline{s_i}\rangle\theta = \langle\overline{s_i}\sigma\rangle, i = 1, \ldots, n$. Then, by definition of $\mathcal{R}_{b_{k_i}}$, we have that $k_i < k$ for all $i = 1, \ldots, n$ and, thus, by the induction hypothesis, we have $\langle t_i\sigma, \pi_i\rangle \xleftarrow{\mathsf{c}}_{\mathcal{R}} \langle \mathsf{f}_i(\overline{s_i}\sigma), [\,]\rangle$, $i = 1, \ldots, n$. Consider now the equivalent rule in $\mathcal{R}$: $\beta : \mathsf{f}(\overline{s_0}) \to r \Leftarrow \mathsf{f}_1(\overline{s_1}) \twoheadrightarrow t_1, \ldots, \mathsf{f}_n(\overline{s_n}) \twoheadrightarrow t_n$ in $\mathcal{R}$. Therefore, we have $\langle t, \pi\rangle \xleftarrow{\mathsf{c}}_{\mathcal{R}} \langle \mathsf{f}(\overline{s}), [\,]\rangle$,

$$\sigma' = \sigma\restriction_{(\mathcal{V}\mathsf{ar}(\overline{s_0})\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_1},\ldots,\overline{s_n},\overline{t_n}))\cup\bigcup_{i=1}^{n}\mathcal{V}\mathsf{ar}(t_i)\setminus\mathcal{V}\mathsf{ar}(r,\overline{s_{i+1}},\ldots,\overline{s_n})}$$

and $\pi = \beta(\sigma', \pi_1, \ldots, \pi_n)$. Finally, since $\{\overline{y}\} = (\mathcal{V}\mathsf{ar}(\overline{s_0})\setminus\mathcal{V}\mathsf{ar}(r, \overline{s_1}, \ldots, \overline{s_n}, \overline{t_n})) \cup \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(t_i)\setminus\mathcal{V}\mathsf{ar}(r, \overline{s_{i+1}}, \ldots, \overline{s_n})$, we can conclude that $\widehat{\pi} = \pi$. □

2.5.3 Improving the transformation for injective functions

When a function is injective, one can expect the injectivization transformation to be unnecessary. This is not generally true, since some additional syntactic conditions might also be required. Furthermore, depending on the considered setting, it can be necessary to have an injective *system*, rather than an injective function. Consider, e.g., the following simple TRS:

$$\mathcal{R} = \{\ \mathsf{f}_1 \to \mathsf{f}_2, \mathsf{f}_2 \to \mathsf{0},\ \mathsf{g}_1 \to \mathsf{g}_2,\ \mathsf{g}_2 \to \mathsf{0}\ \}$$

Here, all functions are clearly injective. However, given a reduction like $\mathsf{f}_1 \to_{\mathcal{R}} \mathsf{f}_2 \to_{\mathcal{R}} \mathsf{0}$, we do not know which rule should be applied to $\mathsf{0}$ in order to go backwards until the initial term (actually, both the second and the fourth rules are applicable in the reverse direction).

Luckily, in our context, the injectivity of a function suffices since reductions in pcDCTRSs are performed in a single step. Therefore, given a reduction of the form $\mathsf{f}^{\mathtt{i}}(\overline{s_n}) \to_{\mathcal{R}} t$, a backward computation will have the form $\mathsf{f}^{-1}(t) \to_{\mathcal{R}} \langle\overline{s_n}\rangle$, so that we know that only the inverse rules of f are applicable.

Now, we present an improvement of the injectivization transformation presented in Section 2.5.1 which has some similarities with that in [97]. Here, we consider that the initial system is a TRS $\mathcal{R}$ since, to the best of our knowledge, there is no reachability analysis defined for DCTRSs. In the following, given a term s, we let

$$\mathsf{range}(s) = \{t \mid s\sigma \to_{\mathcal{R}}^{*} t,\ \sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{C}),\ \text{and } t \in \mathcal{T}(\mathcal{C})\}$$

i.e., $\mathsf{range}(s)$ returns a set with the constructor normal forms of all possible ground constructor instances of s. Although computing this set is generally undecidable, there are some overapproximations based on the use of tree automata (see, e.g., [49] and the most recent approach for innermost rewriting [50]). Let us consider that $\mathsf{range}^\alpha(s)$ is such an approximation, with $\mathsf{range}^\alpha(s) \supseteq \mathsf{range}(s)$ for all terms s. Here, we are interested in determining when the right-hand sides, r_1 and r_2, of two rules do not overlap, i.e., $\mathsf{range}(r_1) \cap \mathsf{range}(r_2) = \varnothing$. For this purpose, we will check whether $\mathsf{range}^\alpha(r_1) \cap \mathsf{range}^\alpha(r_2) = \varnothing$. Since finite tree automata are closed under intersection and the emptiness of a finite tree automata is decidable, checking the emptiness of $\mathsf{range}^\alpha(r_1) \cap \mathsf{range}^\alpha(r_2)$ is decidable and can be used to *safely* identify non-overlapping right-hand sides, i.e., if $\mathsf{range}^\alpha(r_1) \cap \mathsf{range}^\alpha(r_2) = \varnothing$, then r_1 and r_2 are definitely non-overlapping; otherwise, they may be overlapping or non-overlapping.

Now, we summarize our method to simplify some trace terms. Given a constructor TRS $\mathcal{R}$ and a rule $\beta : l \to r \in \mathcal{R}$, we check the following conditions:

1. the right-hand side r of the rule does not overlap with the right-hand side of any other rule defining the same function;

2. the rule is non-erasing, i.e., $\mathcal{V}\mathsf{ar}(l) = \mathcal{V}\mathsf{ar}(r)$;

3. the right-hand side r contains a single occurrence of a defined function symbol, say $\mathsf{f} \in \mathcal{D}$.

If these conditions hold, then the rule has the form $l \to r[\mathsf{f}(\overline{s})]_p$ with l and $\mathsf{f}(\overline{s})$ basic terms,[13] and $r[x]_p$ and $\overline{s}$ constructor terms, where x is a fresh variable. In this case, we can safely produce the following injective version:[14]

$$l^{\mathtt{i}} \to \langle r[x]_p, w\rangle \Leftarrow \mathsf{f}^{\mathtt{i}}(\overline{s}) \twoheadrightarrow \langle x, w\rangle$$

instead of

$$l^{\mathtt{i}} \to \langle r[x]_p, \beta(w)\rangle \Leftarrow \mathsf{f}^{\mathtt{i}}(\overline{s}) \twoheadrightarrow \langle x, w\rangle$$

Let us illustrate this improved transformation with a couple of examples.

[13] Note that l is a basic term since we initially consider a constructor TRS and, thus, all left-hand sides are basic terms by definition.

[14] Since $l \to r$ is non-erasing, the pcDCTRS rule $l \to r[x]_p \Leftarrow \mathsf{f}(\overline{s}) \twoheadrightarrow x$ is trivially non-erasing too (according to [110], i.e., $(\mathcal{V}\mathsf{ar}(l) \backslash \mathcal{V}\mathsf{ar}(r[x]_p, \mathsf{f}(\overline{s}), x)) \cup \mathcal{V}\mathsf{ar}(x) \backslash \mathcal{V}\mathsf{ar}(r[x]_p) = \varnothing$) and, thus, no binding should be stored during the injectivization process.

Example 2.37. Consider the following TRS:

$$\mathcal{R} = \{\ \mathsf{f}(\mathsf{s}(x)) \to \mathsf{g}(x),\ \mathsf{f}(\mathsf{c}(x)) \to \mathsf{h}(x),\ \mathsf{g}(x) \to \mathsf{s}(x),\ \mathsf{h}(x) \to \mathsf{c}(x)\}$$

Here, it can easily be shown that $\mathsf{range}^{\alpha}(\mathsf{g}(x)) \cap \mathsf{range}^{\alpha}(\mathsf{h}(x)) = \varnothing$, the two rules defining f are non-erasing, and both contain a single occurrence of a defined function symbol in the right-hand sides. Therefore, our improved injectivization applies and we get the following pcDCTRS $\mathcal{R}_f$:

$$\begin{array}{ll} \mathsf{f}^{\mathsf{i}}(\mathsf{s}(x)) \to \langle y, w\rangle \Leftarrow \mathsf{g}^{\mathsf{i}}(x) \twoheadrightarrow \langle y, w\rangle & \mathsf{g}^{\mathsf{i}}(x) \to \langle \mathsf{s}(x), \beta_3\rangle \\ \mathsf{f}^{\mathsf{i}}(\mathsf{c}(x)) \to \langle y, w\rangle \Leftarrow \mathsf{h}^{\mathsf{i}}(x) \twoheadrightarrow \langle y, w\rangle & \mathsf{h}^{\mathsf{i}}(x) \to \langle \mathsf{c}(x), \beta_4\rangle \end{array}$$

In contrast, the original injectivization transformation would return the following system:

$$\begin{array}{ll} \mathsf{f}^{\mathsf{i}}(\mathsf{s}(x)) \to \langle y, \beta_1(w)\rangle \Leftarrow \mathsf{g}^{\mathsf{i}}(x) \twoheadrightarrow \langle y, w\rangle & \mathsf{g}^{\mathsf{i}}(x) \to \langle \mathsf{s}(x), \beta_3\rangle \\ \mathsf{f}^{\mathsf{i}}(\mathsf{c}(x)) \to \langle y, \beta_2(w)\rangle \Leftarrow \mathsf{h}^{\mathsf{i}}(x) \twoheadrightarrow \langle y, w\rangle & \mathsf{h}^{\mathsf{i}}(x) \to \langle \mathsf{c}(x), \beta_4\rangle \end{array}$$

Finally, the inverse system $\mathcal{R}_b$ obtained from $\mathcal{R}_f$ using the original transformation has the following form:

$$\begin{array}{ll} \mathsf{f}^{-1}(y, w) \to \langle \mathsf{s}(x)\rangle \Leftarrow \mathsf{g}^{-1}(y, w) \twoheadrightarrow \langle x\rangle & \mathsf{g}^{-1}(\mathsf{s}(x), \beta_3) \to \langle x\rangle \\ \mathsf{f}^{-1}(y, w) \to \langle \mathsf{c}(x)\rangle \Leftarrow \mathsf{h}^{-1}(y, w) \twoheadrightarrow \langle x\rangle & \mathsf{h}^{-1}(\mathsf{c}(x), \beta_4) \to \langle x\rangle \end{array}$$

For instance, given the forward reduction $\mathsf{f}^{\mathsf{i}}(\mathsf{s}(0)) \to_{\mathcal{R}_f} \langle \mathsf{s}(0), \beta_3\rangle$, we can build the corresponding backward reduction: $\mathsf{f}^{-1}(\mathsf{s}(0), \beta_3) \to_{\mathcal{R}_b} \langle \mathsf{s}(0)\rangle$.

Note, however, that the left-hand sides of f^{-1} overlap and we should reduce the conditions in order to determine which rule to apply. Therefore, in some cases, there is a trade-off between the size of the trace terms and the complexity of the reduction steps.

The example above, though, only produces a rather limited improvement since the considered functions are not recursive. Our next example shows a much significant improvement. Here, we consider the function zip (also used in [97] to illustrate the benefits of an injectivity analysis).

Example 2.38. Consider the following TRS $\mathcal{R}$ defining the function zip:

$$\begin{array}{c} \mathsf{zip}([\,], ys) \to [\,] \\ \mathsf{zip}(xs, [\,]) \to [\,] \\ \mathsf{zip}(x : xs, y : ys) \to \mathsf{pair}(x, y) : \mathsf{zip}(xs, ys) \end{array}$$

Here, since the third rule is non-erasing, its right-hand side contains a single occurrence of a defined function, zip, and it does not overlap with any other right-hand side, our improved injectivization applies and we get the following pcDCTRS $\mathcal{R}_f$:

$$\begin{aligned}
\mathsf{zip}^{\mathtt{i}}([\,], ys) &\to \langle [\,], \beta_1(ys)\rangle \\
\mathsf{zip}^{\mathtt{i}}(xs, [\,]) &\to \langle [\,], \beta_2(xs)\rangle \\
\mathsf{zip}^{\mathtt{i}}(x : xs, y : ys) &\to \langle \mathsf{pair}(x, y) : zs, w\rangle \Leftarrow \mathsf{zip}^{\mathtt{i}}(xs, ys) \twoheadrightarrow \langle zs, w\rangle
\end{aligned}$$

In contrast, the original injectivization transformation would return the following system $\mathcal{R}'_f$:

$$\begin{aligned}
\mathsf{zip}^{\mathtt{i}}([\,], ys) &\to \langle [\,], \beta_1(ys)\rangle \\
\mathsf{zip}^{\mathtt{i}}(xs, [\,]) &\to \langle [\,], \beta_2(xs)\rangle \\
\mathsf{zip}^{\mathtt{i}}(x : xs, y : ys) &\to \langle \mathsf{pair}(x, y) : zs, \beta_3(w)\rangle \Leftarrow \mathsf{zip}^{\mathtt{i}}(xs, ys) \twoheadrightarrow \langle zs, w\rangle
\end{aligned}$$

It might seem a small difference, but if we call $\mathsf{zip}^{\mathtt{i}}$ with two lists of n elements, the system $\mathcal{R}'_f$ would build a trace term of the form $\beta_3(\ldots\beta_3(\beta_1(\ldots))\ldots)$ with n nested constructors β_3, while $\mathcal{R}_f$ would just build the trace term $\beta_1(\ldots)$. For large values of n, this is a significant improvement in memory usage.

2.6 Bidirectional Program Transformation

We illustrate a practical application of our reversibilization technique in the context of *bidirectional program transformation* (see [34] for a survey). In particular, we consider the so-called *view-update* problem. Here, we have a data structure (e.g., a database) called the *source*, which is transformed to another data structure, called the *view*. Typically, we have a *view function*, $\mathsf{view}\colon \mathit{Source} \to \mathit{View}$ that takes the source and returns the corresponding view, together with an *update* function, $\mathsf{upd}\colon \mathit{View} \times \mathit{Source} \to \mathit{Source}$ that propagates the changes in a modified view to the original source. Two basic properties that these functions should satisfy in order to be well-behaved are the following [45]:

$$\begin{aligned}
\forall s \in \mathit{Source}, \forall v \in \mathit{View}\colon\quad & \mathsf{view}(\mathsf{upd}(v, s)) = v \\
\forall s \in \mathit{Source}\colon\quad & \mathsf{upd}(\mathsf{view}(s), s) = s
\end{aligned}$$

Bidirectionalization (first proposed in the database community [10]) basically consists in, given a view function, "bidirectionalize" it in order to derive an appropriate update function. For this purpose, first, a *view complement* function is usually defined, say view^c, so that the tupled function

$$\mathsf{view} \vartriangle \mathsf{view}^c\colon \mathit{Source} \to \mathit{View} \times \mathit{Comp}$$

becomes injective. Therefore, the update function can be defined as follows:

$$\mathsf{upd}(v, s) = (\mathsf{view} \vartriangle \mathsf{view}^c)^{-1}(v, \mathsf{view}^c(s))$$

This approach has been applied to bidirectionalize view functions in a functional language in [97].

In the following, we apply our injectivization and inversion transformations in order to produce a bidirectionalization transformation that may be useful in the context of the view-update problem (with some limitations). Let us assume that we have a view function, view, that takes a source and returns the corresponding view, and which is defined by means of a pcDCTRS. Following our approach, given the original program $\mathcal{R}$, we produce an injectivized version $\mathcal{R}_f$ and the corresponding inverse $\mathcal{R}_b$. Therefore, in principle, one can use $\mathcal{R}_f \cup \mathcal{R}_b$, which will include the functions $\mathsf{view}^{\mathtt{i}}$ and view^{-1}, to define an update function as follows:

$$\mathsf{upd}(v, s) \to s' \Leftarrow \mathsf{view}^{\mathtt{i}}(s) \twoheadrightarrow \langle v', \pi\rangle, \mathsf{view}^{-1}(v, \pi) \twoheadrightarrow \langle s'\rangle$$

where s is the original source, v is the updated view, and s', the returned value, is the corresponding updated source. Note that, in our context, the function $\mathsf{view}^{\mathtt{i}}$ is somehow equivalent to $\mathsf{view} \vartriangle \mathsf{view}^c$ above.

Let us now illustrate the bidirectionalization process with an example. Consider a particular data structure, a list of *records* of the form $\mathsf{r}(t, v)$ where t is the type of the record (e.g., book, dvd, pen, etc.) and v is its price tag. The following system defines a view function that takes a type and a list of records, and returns a list with the price tags of the records of the given type:[15]

$$\begin{array}{rcl}
\mathsf{view}(t, \mathsf{nil}) & \to & \mathsf{nil} \\
\mathsf{view}(t, \mathsf{r}(t', v) : rs) & \to & \mathsf{val}(\mathsf{r}(t', v)) : \mathsf{view}(t, rs) \Leftarrow \mathsf{eq}(t, t') \twoheadrightarrow \mathsf{true} \\
\mathsf{view}(t, \mathsf{r}(t', v) : rs) & \to & \mathsf{view}(t, rs) \Leftarrow \mathsf{eq}(t, t') \twoheadrightarrow \mathsf{false} \\
\mathsf{eq}(\mathsf{book}, \mathsf{book}) & \to & \mathsf{true} \qquad\qquad \mathsf{eq}(\mathsf{dvd}, \mathsf{dvd}) \to \mathsf{true} \\
\mathsf{eq}(\mathsf{book}, \mathsf{dvd}) & \to & \mathsf{false} \qquad\quad \mathsf{eq}(\mathsf{dvd}, \mathsf{book}) \to \mathsf{false} \\
\mathsf{val}(\mathsf{r}(t, v)) & \to & v
\end{array}$$

However, this system is not a pcDCTRS. Here, we use a flattening transformation to produce the following (labeled) pcDCTRS $\mathcal{R}$ which is equivalent for constructor

[15] For simplicity, we restrict the record types to only book and dvd.

derivations:

$$
\begin{array}{ll}
\beta_1: & \mathsf{view}(t, \mathsf{nil}) \to \mathsf{nil} \\
\beta_2: & \mathsf{view}(t, \mathsf{r}(t', v) : rs) \to p : r \\
 & \qquad \Leftarrow \mathsf{eq}(t, t') \twoheadrightarrow \mathsf{true}, \mathsf{val}(\mathsf{r}(t', v)) \twoheadrightarrow p, \mathsf{view}(t, rs) \twoheadrightarrow r \\
\beta_3: & \mathsf{view}(t, \mathsf{r}(t', v) : rs) \to r \Leftarrow \mathsf{eq}(t, t') \twoheadrightarrow \mathsf{false}, \mathsf{view}(t, rs) \twoheadrightarrow r
\end{array}
$$

$$
\begin{array}{lll}
\beta_4: & \mathsf{eq}(\mathsf{book}, \mathsf{book}) \to \mathsf{true} & \beta_5: \ \mathsf{eq}(\mathsf{dvd}, \mathsf{dvd}) \ \to \ \mathsf{true} \\
\beta_6: & \mathsf{eq}(\mathsf{book}, \mathsf{dvd}) \to \mathsf{false} & \beta_7: \ \mathsf{eq}(\mathsf{dvd}, \mathsf{book}) \ \to \ \mathsf{false}
\end{array}
$$

$$
\beta_8: \quad \mathsf{val}(\mathsf{r}(t, v)) \to v
$$

Now, we can apply our injectivization transformation which returns the following pcDCTRS $\mathcal{R}_f = \mathbf{I}(\mathcal{R})$:

$$
\begin{array}{l}
\mathsf{view}^{\mathtt{i}}(t, \mathsf{nil}) \to \langle \mathsf{nil}, \beta_1(t) \rangle \\
\mathsf{view}^{\mathtt{i}}(t, \mathsf{r}(t', v) : rs) \to \langle p : r, \beta_2(w_1, w_2, w_3) \rangle \\
\quad \Leftarrow \mathsf{eq}^{\mathtt{i}}(t, t') \twoheadrightarrow \langle \mathsf{true}, w_1 \rangle, \mathsf{val}^{\mathtt{i}}(\mathsf{r}(t', v)) \twoheadrightarrow \langle p, w_2 \rangle, \mathsf{view}^{\mathtt{i}}(t, rs) \twoheadrightarrow \langle r, w_3 \rangle \\
\mathsf{view}^{\mathtt{i}}(t, \mathsf{r}(t', v) : rs) \to \langle r, \beta_3(v, w_1, w_2) \rangle \\
\quad \Leftarrow \mathsf{eq}^{\mathtt{i}}(t, t') \twoheadrightarrow \langle \mathsf{false}, w_1 \rangle, \mathsf{view}^{\mathtt{i}}(t, rs) \twoheadrightarrow \langle r, w_2 \rangle
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{eq}^{\mathtt{i}}(\mathsf{book}, \mathsf{book}) \to \langle \mathsf{true}, \beta_4 \rangle & \mathsf{eq}^{\mathtt{i}}(\mathsf{dvd}, \mathsf{dvd}) \ \to \ \langle \mathsf{true}, \beta_5 \rangle \\
\mathsf{eq}^{\mathtt{i}}(\mathsf{book}, \mathsf{dvd}) \to \langle \mathsf{false}, \beta_6 \rangle & \mathsf{eq}^{\mathtt{i}}(\mathsf{dvd}, \mathsf{book}) \ \to \ \langle \mathsf{false}, \beta_7 \rangle
\end{array}
$$

$$
\mathsf{val}^{\mathtt{i}}(\mathsf{r}(t, v)) \to \langle v, \beta_8(t) \rangle
$$

Finally, inversion returns the following pcDCTRS $\mathcal{R}_b = \mathbf{I}(\mathcal{R}_f)$:

$$
\begin{array}{l}
\mathsf{view}^{-1}(\mathsf{nil}, \beta_1(t)) \to \langle t, \mathsf{nil} \rangle \\
\mathsf{view}^{-1}(p : r, \beta_2(w_1, w_2, w_3)) \to \langle t, \mathsf{r}(t', v) : rs \rangle \\
\quad \Leftarrow \mathsf{eq}^{-1}(\mathsf{true}, w_1) \twoheadrightarrow \langle t, t' \rangle, \mathsf{val}^{-1}(p, w_2) \twoheadrightarrow \langle \mathsf{r}(t', v) \rangle, \mathsf{view}^{-1}(r, w_3) \twoheadrightarrow \langle t, rs \rangle \\
\quad \mathsf{view}^{-1}(r, \beta_3(v, w_1, w_2)) \to \langle t, \mathsf{r}(t', v) : rs \rangle \\
\qquad \Leftarrow \mathsf{eq}^{-1}(\mathsf{false}, w_1) \twoheadrightarrow \langle t, t' \rangle, \mathsf{view}^{-1}(r, w_2) \twoheadrightarrow \langle t, rs \rangle
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{eq}^{-1}(\mathsf{true}, \beta_4) \to \langle \mathsf{book}, \mathsf{book} \rangle & \mathsf{eq}^{-1}(\mathsf{true}, \beta_5) \ \to \ \langle \mathsf{dvd}, \mathsf{dvd} \rangle \\
\mathsf{eq}^{-1}(\mathsf{false}, \beta_6) \to \langle \mathsf{book}, \mathsf{dvd} \rangle & \mathsf{eq}^{-1}(\mathsf{false}, \beta_7) \ \to \ \langle \mathsf{dvd}, \mathsf{book} \rangle
\end{array}
$$

$$
\mathsf{val}^{-1}(v, \beta_8(t)) \to \langle \mathsf{r}(t, v) \rangle
$$

For instance, the term $\mathsf{view}(\mathsf{book}, [\mathsf{r}(\mathsf{book}, 12), \mathsf{r}(\mathsf{dvd}, 24)])$, reduces to $[12]$ in the original system $\mathcal{R}$. Given a modified view, e.g., $[15]$, we can compute the modified source using function upd above:

$$
\mathsf{upd}([\mathsf{r}(\mathsf{book}, 12), \mathsf{r}(\mathsf{dvd}, 24)], \ [15])
$$

Here, we have the following subcomputations:[16]

$$\begin{aligned}
&\mathsf{view}^{\mathtt{i}}(\mathsf{book}, [\mathsf{r}(\mathsf{book}, 12), \mathsf{r}(\mathsf{dvd}, 24)]) \\
&\qquad\qquad \to_{\mathcal{R}_f} \langle [12], \beta_2(\beta_4, \beta_8(\mathsf{book}), \beta_3(24, \beta_6, \beta_1(\mathsf{book})))\rangle \\
&\mathsf{view}^{-1}([15], \beta_2(\beta_4, \beta_8(\mathsf{book}), \beta_3(24, \beta_6, \beta_1(\mathsf{book})))) \\
&\qquad\qquad \to_{\mathcal{R}_b} \langle \mathsf{book}, [\mathsf{r}(\mathsf{book}, 15), \mathsf{r}(\mathsf{dvd}, 24)]\rangle
\end{aligned}$$

Thus upd returns the updated source $[\mathsf{r}(\mathsf{book}, 15), \mathsf{r}(\mathsf{dvd}, 24)]$, as expected. We note that the considered example cannot be transformed using the technique in [97], the closer to our approach, since the right-hand sides of some rules contain functions which are not *treeless*.[17] Nevertheless, one could consider a transformation from pcDCTRS to functional programs with treeless functions so that the technique in [97] becomes applicable.

Our approach can solve a view-update problem as long as the view function can be encoded in a pcDCTRS. When this is the case, the results from Section 2.5 guarantee that function upd is well defined. Formally analyzing the class of view functions that can be represented with a pcDCTRS is an interesting topic for further research.

2.7 Related Work

There is no widely accepted notion of reversible computing. In this work, we have considered one of its most popular definitions, according to which a computation principle is reversible if there is a method to *undo* a (forward) computation. Moreover, we expect to get back to an *exact* past state of the computation. This is often referred to as *full reversibility*.

As we have mentioned in the introduction, some of the most promising applications of reversibility include cellular automata [103], bidirectional program transformation [97], already discussed in Section 2.6, reversible debugging [51], where the ability to go both forward and backward when seeking the cause of an error can be very useful for the programmer, parallel discrete event simulation [124], where reversibility is used to undo the effects of speculative computations made on a wrong assumption, quantum computing [143], where all computations should be reversible, and so forth. The interested reader can find detailed surveys in the

[16] Note that, in this case, the function view requires not only the source but also the additional parameter book.

[17] A call is *treeless* if it has the form $\mathsf{f}(x_1, \ldots, x_n)$ and $x_1, \ldots, x_n$ are different variables.

state of the art reports of the different working groups of COST Action IC1405 on Reversible Computation [68].

Intuitively speaking, there are two broad approaches to reversibility from a programming language perspective:

Reversible programming languages. In this case, all constructs of the programming language are reversible. One of the most popular languages within the first approach is the reversible (imperative) language Janus [93]. The language was recently rediscovered [148, 147, 149] and has since been formalized and further developed.

Irreversible programming languages and Landauer's embedding. Alternatively, one can consider an irreversible programming language, and enhance the states with some additional information (typically, the *history* of the computation so far) so that computations become reversible. This is called *Landauer's embedding*.

In this work, we consider reversibility in the context of term rewriting. To the best of our knowledge, we have presented the first approach to reversibility in term rewriting. A closest approach was introduced by Abramsky in the context of pattern matching automata [2], though his developments could easily be applied to rewrite systems as well. In Abramsky's approach, *biorthogonality* was required to ensure reversibility, which would be a very significant restriction for term rewriting systems. Basically, biorthogonality requires that, for every pair of (different) rewrite rules $l \to r$ and $l' \to r'$, l and l' do not *overlap* (roughly, they do not unify) and r and r' do not overlap too. Trivially, the functions of a biorthogonal system are injective and, thus, computations are reversible without the need of a Landauer embedding. Therefore, Abramsky's work is aimed at defining a reversible language, in contrast to our approach that is based on defining a Landauer embedding for standard term rewriting and a general class of rewrite systems.

Defining a Landauer embedding in order to make a computation mechanism reversible has been applied in different contexts and computational models, e.g., a probabilistic guarded command language [150], a low level virtual machine [128], the call-by-name lambda calculus [67, 75], cellular automata [135, 102], combinatory logic [37], a flowchart language [147], etc.

In the context of declarative languages, we find the work by Mu *et al.* [104], where a relational reversible language is presented (in the context of bidirectional programming). A similar approach was then introduced by Matsuda *et al.* [97, 98] in the context of functional programs and bidirectional transformation. The functional

programs considered in [97] can be seen as linear and *right-treeless*[18] constructor TRSs. The class of functional programs is more general in [98], which would correspond to left-linear, right-treeless TRSs. The reversibilization technique of [97, 98] includes both an injectivization stage (by introducing a *view complement* function) and an inversion stage. These methods are closely related to the transformations of injectivization and inversion that we have presented in Section 2.5, although we developed them from a rather different starting point. Moreover, their methods for injectivization and inversion consider a more restricted class of systems than those considered in this paper. On the other hand, they apply a number of analyses to improve the result, which explains the smaller traces in their approach. All in all, we consider that our approach gives better insights to understand the need for some of the requirements of the program transformations and the class of considered programs. For instance, most of our requirements come from the need to remove programs positions from the traces, as shown in Section 2.4.

Finally, [133] considers the reversible language RFUN. Similarly to Janus, computations in RFUN are reversible without the need of a Landauer embedding. The paper also presents a transformation from a simple (irreversible) functional language, FUN, to RFUN, in order to highlight how irreversibilities are handled in RFUN. The transformation has some similarities with both the approach of [97] and our improved transformation in Section 2.5.3; on the other hand, though, [133] also applies the Bennett *trick* [11] in order to avoid some unnecessary information.

2.8 Discussion and Future Work

In this paper, we have introduced a reversible extension of term rewriting. In order to keep our approach as general as possible, we have initially considered DCTRSs as input systems, and proved the soundness and reversibility of our extension of rewriting. Then, in order to introduce a reversibilization transformation for these systems, we have also presented a transformation from DCTRSs to pure constructor systems (pcDCTRSs) which is correct for constructor reduction. A further improvement is presented for injective functions, which may have a significant impact in memory usage in some cases. Finally, we have successfully applied our approach in the context of bidirectional program transformation.

We have developed a prototype implementation of the reversibilization transformations introduced in Section 2.5. The tool can read an input TRS file (format `.trs`

[18] There are no nested defined symbols in the right-hand sides, and, moreover, any term rooted by a defined function in the right-hand sides can only take different variables as its proper subterms.

[1]) and then it applies in a sequential way the following transformations: flattening, simplification of constructor conditions, injectivization, and inversion. The tool prints out the CTRSs obtained at each transformation step. It is publicly available through a web interface from `http://kaz.dsic.upv.es/rev-rewriting.html`, where we have included a number of examples to easily test the tool.

As for future work, we plan to investigate new methods to further reduce the size of the traces. In particular, we find it interesting to define a reachability analysis for DCTRSs. A reachability analysis for CTRSs without extra-variables (1-CTRSs) can be found in [42], but the extension to deal with extra-variables in DCTRSs (since a DCTRS is a particular case of 3-CTRS) seems challenging. Furthermore, as mentioned in the paper, a completion procedure to add *default* cases to some functions (as suggested in Section 2.5.1) may help to broaden the applicability of the technique and avoid the restriction to constructor reduction. Finally, our injectivization and inversion transformations are correct w.r.t. innermost reduction. Extending our results to a lazy strategy is also an interesting topic for further research.

Acknowledgments

We thank the anonymous reviewers for their useful comments and suggestions to improve this paper.

Chapter 3

A Theory of Reversibility for Erlang

Ivan Lanese[1], Naoki Nishida[2], Adrián Palacios[3], Germán Vidal[3]

[1] Focus Team, University of Bologna/INRIA
Mura Anteo Zamboni, 7, Bologna, Italy
ivan.lanese@gmail.com

[2] Graduate School of Informatics, Nagoya University
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
nishida@i.nagoya-u.ac.jp

[3] MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
{apalacios,gvidal}@dsic.upv.es

Abstract. In a reversible language, any forward computation can be undone by a finite sequence of backward steps. Reversible computing has been studied in the context of different programming languages and formalisms, where it has been used for testing and verification, among others. In this paper, we consider a subset of Erlang, a functional and concurrent programming language

This work has been partially supported by MINECO/AEI/FEDER (EU) under grants TIN2013-44742-C4-1-R and TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic), by the COST Action IC1405 on Reversible Computation - extending horizons of computing, and by JSPS KAKENHI Grant Number JP17H01722. Adrián Palacios was partially supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores* and *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D*, MINECO (SEIDI), under FPI grants BES-2014-069749 and EEBB-I-16-11469. Ivan Lanese was partially supported by INdAM as a member of GNCS (Gruppo Nazionale per il Calcolo Scientifico). Part of this research was done while the third and fourth authors were visiting Nagoya and Bologna Universities; they gratefully acknowledge their hospitality. Finally, we thank Salvador Tamarit and the anonymous reviewers for their helpful suggestions and comments. This chapter is an adapted author version of the paper published in *"Ivan Lanese, Naoki Nishida, Adrián Palacios, Germán Vidal: A theory of reversibility for Erlang. Journal of Logical and Algebraic Methods in Programming 100: 71–97 (2018)"*. DOI: https://doi.org/10.1016/j.jlamp.2018.06.004

based on the actor model. We present a formal semantics for reversible computation in this language and prove its main properties, including its causal consistency. We also build on top of it a rollback operator that can be used to undo the actions of a process up to a given checkpoint.

3.1 Introduction

Let us consider that the operational semantics of a programming language is specified by a state transition relation R such that $R(s, s')$ holds if the state s' is reachable—in one step—from state s. Then, we say that a programming language (or formalism) is *reversible* if there exists a constructive algorithm that can be used to recover the predecessor state s from s'. In general, such a property does not hold for most programming languages and formalisms. We refer the interested reader to, e.g., [12, 46, 145, 146] for a high level account of the principles of reversible computation.

The notion of *reversible computation* was first introduced in Landauer's seminal work [80] and, then, further improved by Bennett [11] in order to avoid the generation of "garbage" data. The idea underlying these works is that any programming language or formalism can be made reversible by adding the *history* of the computation to each state, which is usually called a *Landauer embedding*. Although carrying the history of a computation might seem infeasible because of its size, there are several successful proposals that are based on this idea. In particular, one can restrict the original language or apply a number of analysis in order to restrict the required information in the history as much as possible, as in, e.g., [97, 108, 133] in the context of a functional language.

In this paper, we aim at introducing a form of reversibility in the context of a programming language that follows the actor model (concurrency based on message passing), a first-order subset of the concurrent and functional language Erlang [5]. Previous approaches have mainly considered reversibility in—mostly synchronous—concurrent calculi like CCS [35, 36] and π-calculus [33]; a general framework for reversibility of algebraic process calculi [117], or the recent approach to reversible session-based π-calculus [134]. However, we can only find a few approaches that considered the reversibility of *asynchronous* calculi, e.g., Cardelli and Laneve's reversible structures [21], and reversible extensions of the concurrent functional language μOz [90], of a higher-order asynchronous π-calculus [82], and of the coordination language μKlaim [52]. In the last two cases, a form of control of the backward execution using a rollback operator has also been studied [81, 52]. In the case of μOz, reversibility has been exploited for debugging [51].

To the best of our knowledge, our work is the first one that considers reversibility in the context of the functional, concurrent, and distributed language Erlang. Here, given a running Erlang system consisting of a pool of interacting processes, possibly distributed in several computers, we aim at allowing a *single* process to undo its actions in a stepwise manner, including the interactions with other processes, following a rollback fashion. In this context, we must ensure *causal consistency* [35], i.e., an action cannot be undone until all the actions that depend on it have already been undone. E.g., if a process p1 spawns a process p2, we cannot undo the spawning of process p2 until all the actions performed by the process p2 are undone too. This is particularly challenging in an asynchronous and distributed setting, where ensuring causal consistency for backward computations is far from trivial.

In this paper, we consider a simple Erlang-like language that can be seen as a subset of *Core Erlang* [25]. We present the following contributions:

- First, we introduce an appropriate semantics for the language. In contrast to previous semantics like that in [19] which were monolithic, ours is modular, which simplifies the definition of a reversible extension. Here, we follow some of the ideas in [130], e.g., the use of a global mailbox (there called "ether"). There are also some differences though. In the semantics of [130], at the expression level, the semantics of a receive statement is, in principle, infinitely branching, since their formulation allows for an infinite number of possible queues and selected messages (see [47, page 53] for a detailed explanation). This source of nondeterminism is avoided in our semantics.

- We then introduce a reversible semantics that can go both forward and backward (basically, a Landauer embedding), in a nondeterministic fashion, called an *uncontrolled* reversible semantics according to the terminology in [83]. Here, we focus on the concurrent actions (namely, process spawning, message sending and receiving) and, thus, we do not define a reversible semantics for the functional component of the language; rather, we assume that the state of the process—the current expression and its environment—is stored in the history after each execution step. This approach could be improved following, e.g., the techniques presented in [97, 108, 133]. We state and formally prove several properties of the semantics and, particularly, its causal consistency.

- Finally, we add control to the reversible semantics by introducing a *rollback operator* that can be used to undo the actions of a given process until a given checkpoint—introduced by the programmer—is reached. In order to ensure

$$
\begin{array}{rcl}
module & ::= & \mathsf{module}\ Atom = fun_1\ \ldots\ fun_n \\
fun & ::= & fname = \mathsf{fun}\ (Var_1, \ldots, Var_n) \rightarrow expr \\
fname & ::= & Atom/Integer \\
lit & ::= & Atom \mid Integer \mid Float \mid Pid \mid [\,] \\
expr & ::= & Var \mid lit \mid fname \mid [expr_1|expr_2] \mid \{expr_1, \ldots, expr_n\} \\
 & \mid & \mathsf{call}\ Op\ (expr_1, \ldots, expr_n) \mid \mathsf{apply}\ fname\ (expr_1, \ldots, expr_n) \\
 & \mid & \mathsf{case}\ expr\ \mathsf{of}\ clause_1; \ldots; clause_m\ \mathsf{end} \\
 & \mid & \mathsf{let}\ Var = expr_1\ \mathsf{in}\ expr_2 \mid \mathsf{receive}\ clause_1; \ldots; clause_n\ \mathsf{end} \\
 & \mid & \mathsf{spawn}(fname, [expr_1, \ldots, expr_n]) \mid expr\ !\ expr \mid \mathsf{self}() \\
clause & ::= & pat\ \mathsf{when}\ expr_1 \rightarrow expr_2 \\
pat & ::= & Var \mid lit \mid [pat_1|pat_2] \mid \{pat_1, \ldots, pat_n\}
\end{array}
$$

Figure 3.1: Language syntax rules

causal consistency, the rollback action might be propagated to other, dependent processes.

This paper is an extended version of [109]. Compared to [109], we introduce an uncontrolled reversible semantics and prove a number of fundamental theoretical properties, including its causal consistency. The rollback semantics, originally introduced in [109], has been refined and improved (see Section 3.7 for more details).

The paper is organized as follows. The syntax and semantics of the considered language are presented in Sections 3.2 and 3.3, respectively. Our (uncontrolled) reversible semantics is then introduced in Section 3.4, while the rollback operator is defined in Section 3.5. A proof-of-concept implementation of the reversible semantics is described in Section 3.6. Finally, some related work is discussed in Section 3.7, and Section 3.8 concludes and points out some directions for future work.

3.2 Language Syntax

In this section, we present the syntax of a first-order concurrent and distributed functional language that follows the actor model. Our language is equivalent to a subset of Core Erlang [25].

The syntax of the language can be found in Figure 3.1. Here, a module is a sequence of function definitions, where each function name f/n (atom/arity) has an associated definition of the form $\mathsf{fun}\ (X_1, \ldots, X_n) \rightarrow e$. We consider that a program consists of a single module for simplicity. The body of a function is an *expression*,

which can include variables, literals, function names, lists, tuples, calls to built-in functions—mainly arithmetic and relational operators—, function applications, case expressions, let bindings, and receive expressions; furthermore, we also include the functions spawn, "!" (for sending a message), and self() that are usually considered built-ins in the Erlang language. As is common practice, we assume that X is a fresh variable in a let binding of the form let $X = expr_1$ in $expr_2$.

As shown by the syntax in Figure 3.1, we only consider first-order expressions. Therefore, the first argument in applications and spawns is a function name (instead of an arbitrary expression or closure). Analogously, the first argument in calls is a built-in operation Op.

In this language, we distinguish expressions, patterns, and values. Here, *patterns* are built from variables, literals, lists, and tuples, while *values* are built from literals, lists, and tuples, i.e., they are *ground*—without variables—patterns. Expressions are denoted by $e, e', e_1, e_2, \ldots$, patterns by $pat, pat', pat_1, pat_2, \ldots$ and values by $v, v', v_1, v_2, \ldots$ Atoms are typically denoted with roman letters, while variables start with an uppercase letter. As it is common practice, a *substitution* θ is a mapping from variables to expressions, and $\mathcal{D}\mathsf{om}(\theta) = \{X \in Var \mid X \neq \theta(X)\}$ is its domain.[1] Substitutions are usually denoted by sets of bindings like, e.g., $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$. Substitutions are extended to morphisms from expressions to expressions in the natural way. The identity substitution is denoted by id. Composition of substitutions is denoted by juxtaposition, i.e., $\theta\theta'$ denotes a substitution θ'' such that $\theta''(X) = \theta'(\theta(X))$ for all $X \in Var$. Also, we denote by $\theta[X_1 \mapsto v_1, \ldots, X_n \mapsto v_n]$ the *update* of θ with the mapping $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$, i.e., it denotes a new substitution θ' such that $\theta'(X) = v_i$ if $X = X_i$, for some $i \in \{1, \ldots, n\}$, and $\theta'(X) = \theta(X)$ otherwise.

In a case expression "case e of pat_1 when $e_1 \to e'_1$; $\ldots$; pat_n when $e_n \to e'_n$ end", we first evaluate e to a value, say v; then, we should find (if any) the first clause pat_i when $e_i \to e'_i$ such that v matches pat_i (i.e., there exists a substitution σ for the variables of pat_i such that $v = pat_i\sigma$) and $e_i\sigma$—the *guard*—reduces to *true*; then, the case expression reduces to $e'_i\sigma$. Note that guards can only contain calls to built-in functions (typically, arithmetic and relational operators).

As for the concurrent features of the language, we consider that a *system* is a pool of processes that can only interact through message sending and receiving (i.e., there is no shared memory). Each process has an associated *pid* (process identifier), which is unique in a system. As in Erlang, we consider a specific type or domain Pid for pids. Furthermore, in this work, we assume that pids can only be introduced

[1] Since we consider an eager language, variables are bound to values.

in a computation from the evaluation of functions spawn and self (see below). By abuse of notation, when no confusion can arise, we refer to a process with its pid.

An expression of the form spawn$(f/n, [e_1, \ldots, e_n])$ has, as a *side effect*, the creation of a new process, with a fresh pid p, initialized with the expression apply f/n $(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ are the evaluations of $e_1, \ldots, e_n$, respectively; the expression spawn$(f/n, [e_1, \ldots, e_n])$ itself evaluates to the new pid p. The function self() just returns the pid of the current process. An expression of the form $e_1 \,!\, e_2$, where e_1 evaluates to a pid p and e_2 to a value v, also evaluates to the value v and, as a side effect, the value v—the *message*—will be stored in the queue or *mailbox* of process p at some point in the future.

Finally, an expression "receive pat_1 when $e_1 \to e'_1; \ldots; pat_n$ when $e_n \to e'_n$ end" traverses the messages in the process' queue until one of them matches a branch in the receive statement; i.e., it should find the *first* message v in the process' queue (if any) such that case v of pat_1 when $e_1 \to e'_1; \ldots; pat_n$ when $e_n \to e'_n$ end can be reduced; then, the receive expression evaluates to the same expression to which the above case expression would be evaluated, with the additional side effect of deleting the message v from the process' queue. If there is no matching message in the queue, the process suspends its execution until a matching message arrives.

Example 3.1. Consider the program shown in Figure 3.2, where the symbol "_" is used to denote an *anonymous* variable, i.e., a variable whose name is not relevant. The computation starts with "apply $\mathrm{main}/0$ ()." This creates a process, say p1. Then, p1 spawns two new processes, say p2 and p3, and then sends the message hello to process p3 and the message $\{\mathrm{p3}, \mathrm{world}\}$ to process p2, which then resends world to p3. Note that we consider that variables $P2$ and $P3$ are bound to pids p2 and p3, respectively.

In our language, there is no guarantee regarding which message arrives first to p3, i.e., both interleavings (a) and (b) in Figure 3.3 are possible (resulting in function $\mathrm{target}/0$ returning either $\{\mathrm{hello}, \mathrm{world}\}$ or $\{\mathrm{world}, \mathrm{hello}\}$). This is coherent with the semantics of Erlang, where the only guarantee is that if two messages are sent from process p to process p', and both are delivered, then the order of these messages is kept.[2]

3.3 The Language Semantics

In order to precisely set the framework for our proposal, in this section we formalize the semantics of the considered language.

[2]Current implementations only guarantee this restriction within the same node though.

$$
\begin{aligned}
\mathrm{main}/0 = \mathsf{fun}\ () \to\ & \mathsf{let}\ P2 = \mathsf{spawn}(\mathrm{echo}/0, [\,]) \\
& \mathsf{in\ let}\ P3 = \mathsf{spawn}(\mathrm{target}/0, [\,]) \\
& \mathsf{in\ let}\ \_ = P3\,!\,\mathrm{hello} \\
& \mathsf{in}\ P2\,!\,\{P3, \mathrm{world}\} \\
\mathrm{target}/0 = \mathsf{fun}\ () \to\ & \mathsf{receive} \\
& \quad A \to \mathsf{receive} \\
& \qquad\quad B \to \{A, B\} \\
& \qquad \mathsf{end} \\
& \mathsf{end} \\
\mathrm{echo}/0 = \mathsf{fun}\ () \to\ & \mathsf{receive} \\
& \quad \{P, M\} \to P\,!\,M \\
& \mathsf{end}
\end{aligned}
$$

Figure 3.2: A simple concurrent program

Definition 3.2 (Process)**.** A process is denoted by a tuple $\langle p, (\theta, e), q\rangle$ where p is the pid of the process, (θ, e) is the control—which consists of an environment (a substitution) and an expression to be evaluated—and q is the process' mailbox, a FIFO queue with the sequence of messages that have been sent to the process.

We consider the following operations on local mailboxes. Given a message v and a local mailbox q, we let $v : q$ denote a new mailbox with message v on top of it (i.e., v is the newer message). We also denote with $q \backslash\backslash v$ a new queue that results from q by removing the oldest occurrence of message v (which is not necessarily the oldest message in the queue).

A running *system* can then be seen as a pool of processes, which we formally define as follows:

Definition 3.3 (System)**.** A system is denoted by $\Gamma; \Pi$, where Γ, the *global mailbox*, is a multiset of pairs of the form $(target\_process\_pid, message)$, and Π is a pool of processes, denoted by an expression of the form

$$\langle p_1, (\theta_1, e_1), q_1\rangle \ \mid\ \cdots \ \mid\ \langle p_n, (\theta_n, e_n), q_n\rangle$$

where "$\mid$" denotes an associative and commutative operator. Given a global mailbox Γ, we let $\Gamma \cup \{(p, v)\}$ denote a new mailbox also including the pair (p, v), where we use "$\cup$" as multiset union.

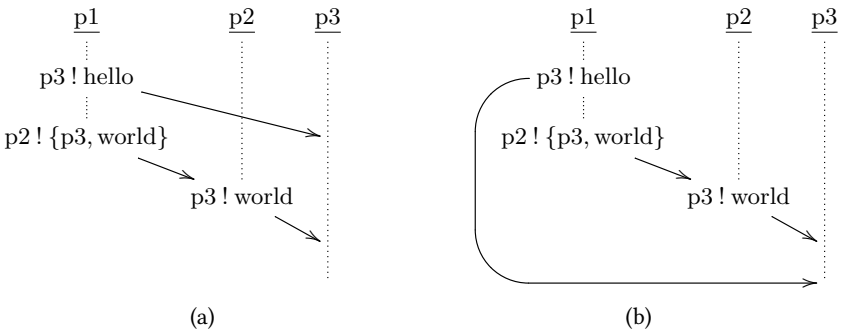


Figure 3.3: Admissible interleavings in Example 3.1

We often denote a system by an expression of the form $\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi$ to point out that $\langle p, (\theta, e), q\rangle$ is an arbitrary process of the pool (thanks to the fact that " $\mid$ " is associative and commutative).

Intuitively, Γ stores messages after they are sent, and before they are inserted in the target mailbox, hence it models messages which are in the network. The use of Γ (which is similar to the "ether" in [130]) is needed to guarantee that all message interleavings admissible in an asynchronous communication model (where the order of messages is not preserved) can be generated by our semantics.

In the following, we denote by $\overline{o_n}$ a sequence of syntactic objects $o_1, \ldots, o_n$ for some n. We also write $\overline{o_{i,j}}$ for the sequence $o_i, \ldots, o_j$ when $i \leq j$ (and the empty sequence otherwise). We write $\overline{o}$ when the number of elements is not relevant.

The semantics is defined by means of two transition relations: $\longrightarrow$ for expressions and $\hookrightarrow$ for systems. Let us first consider the labeled transition relation

$$\longrightarrow : (Env, Exp) \times Label \times (Env, Exp)$$

where Env and Exp are the domains of environments (i.e., substitutions) and expressions, respectively, and $Label$ denotes an element of the set

$$\{\tau, \mathsf{send}(v_1, v_2), \mathsf{rec}(\kappa, \overline{cl_n}), \mathsf{spawn}(\kappa, a/n, [\overline{v_n}]), \mathsf{self}(\kappa)\}$$

whose meaning will be explained below. We use ℓ to range over labels. For clarity, we divide the transition rules of the semantics for expressions in two sets: rules for sequential expressions are depicted in Figure 3.4, while rules for concurrent ones are in Figure 3.5.[3] Note, however, that concurrent expressions can occur inside sequential expressions.

[3]By abuse, we include the rule for $\mathsf{self}()$ together with the concurrent actions.

$$(Var)\ \frac{}{\theta, X \xrightarrow{\tau} \theta, \theta(X)} \qquad (Tuple)\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i'}{\theta, \{\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}\} \xrightarrow{\ell} \theta', \{\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}\}}$$

$$(List1)\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, [e_1|e_2] \xrightarrow{\ell} \theta', [e_1'|e_2]} \qquad (List2)\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, [v_1|e_2] \xrightarrow{\ell} \theta', [v_1|e_2']}$$

$$(Let1)\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 \xrightarrow{\ell} \theta', \mathsf{let}\ X = e_1'\ \mathsf{in}\ e_2} \qquad (Let2)\ \frac{}{\theta, \mathsf{let}\ X = v\ \mathsf{in}\ e \xrightarrow{\tau} \theta[X \mapsto v], e}$$

$$(Case1)\ \frac{\theta, e \xrightarrow{\ell} \theta', e'}{\theta, \mathsf{case}\ e\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\ell} \theta', \mathsf{case}\ e'\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end}}$$

$$(Case2)\ \frac{\mathsf{match}(\theta, v, cl_1, \ldots, cl_n) = \langle \theta_i, e_i \rangle}{\theta, \mathsf{case}\ v\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\tau} \theta\theta_i, e_i}$$

$$(Call1)\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, \ldots, n\}}{\theta, \mathsf{call}\ op\ (\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \xrightarrow{\ell} \theta', \mathsf{call}\ op\ (\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}})}$$

$$(Call2)\ \frac{\mathsf{eval}(op, v_1, \ldots, v_n) = v}{\theta, \mathsf{call}\ op\ (v_1, \ldots, v_n) \xrightarrow{\tau} \theta, v}$$

$$(Apply1)\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, \ldots, n\}}{\theta, \mathsf{apply}\ a/n\ (\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \xrightarrow{\ell} \theta', \mathsf{apply}\ a/n\ (\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}})}$$

$$(Apply2)\ \frac{\mu(a/n) = \mathsf{fun}\ (X_1, \ldots, X_n) \to e}{\theta, \mathsf{apply}\ a/n\ (v_1, \ldots, v_n) \xrightarrow{\tau} \theta \cup \{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}, e}$$

Figure 3.4: Standard semantics: evaluation of sequential expressions

Most of the rules are self-explanatory. In the following, we only discuss some subtle or complex issues. In principle, the transitions are labeled either with τ (a sequential reduction without side effects) or with a label that identifies the reduction of a (possibly concurrent) action with some side-effects. Labels are used in the system rules (Figure 3.6) to determine the associated side effects and/or the information to be retrieved.

As in Erlang, we consider that the order of evaluation of the arguments in a tuple, list, etc., is fixed from left to right.

For case evaluation, we assume an auxiliary function match which selects the first clause, $cl_i = (pat_i\ \mathsf{when}\ e_i' \to e_i)$, such that v matches pat_i, i.e., $v = \theta_i(pat_i)$, and the guard holds, i.e., $\theta\theta_i, e_i' \longrightarrow^* \theta', true$. As in Core Erlang, we assume that the

$$(Send1)\quad \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1 \,!\, e_2 \xrightarrow{\ell} \theta', e_1' \,!\, e_2} \qquad (Send2)\quad \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, v_1 \,!\, e_2 \xrightarrow{\ell} \theta', v_1 \,!\, e_2'}$$

$$(Send3)\quad \frac{}{\theta, v_1 \,!\, v_2 \xrightarrow{\mathsf{send}(v_1, v_2)} \theta, v_2}$$

$$(Receive)\quad \frac{}{\theta, \mathsf{receive}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta, \kappa}$$

$$(Spawn1)\quad \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, \ldots, n\}}{\theta, \mathsf{spawn}(a/n, [\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}]) \xrightarrow{\ell} \theta', \mathsf{spawn}(a/n, [\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}])}$$

$$(Spawn2)\quad \frac{}{\theta, \mathsf{spawn}(a/n, [\overline{v_n}]) \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta, \kappa}$$

$$(Self)\quad \frac{}{\theta, \mathsf{self}() \xrightarrow{\mathsf{self}(\kappa)} \theta, \kappa}$$

Figure 3.5: Standard semantics: evaluation of concurrent expressions

patterns can only contain fresh variables (but guards might have bound variables, thus we pass the current environment θ to function match). Note that, for simplicity, we assume here that if the argument v matches no clause then the evaluation is blocked.[4]

Functions can either be defined in the program (in this case they are invoked by apply) or be a built-in (invoked by call). In the latter case, they are evaluated using the auxiliary function eval. In rule $Apply2$, we consider that the mapping μ stores all function definitions in the program, i.e., it maps every function name a/n to a copy of its definition fun $(X_1, \ldots, X_n) \to e$, where $X_1, \ldots, X_n$ are (distinct) fresh variables and are the only variables that may occur free in e. As for the applications, note that we only consider first-order functions. In order to extend our semantics to also consider higher-order functions, one should reduce the function name to a *closure* of the form $(\theta', \mathsf{fun}\ (X_1, \ldots, X_n) \to e)$. We skip this extension since it is orthogonal to our contribution.

Let us now consider the evaluation of concurrent expressions that produce some side effect (Figure 3.5). Here, we can distinguish two kinds of rules. On the one hand, we have rules $Send1$, $Send2$ and $Send3$ for "!". In this case, we know *locally* what the expression should be reduced to (i.e., v_2 in rule $Send3$). For the remaining rules, this is not known locally and, thus, we return a fresh distinguished symbol,

[4]This is not an issue in practice since, when an Erlang program is translated to the intermediate representation Core Erlang, a catch-all clause is added to every case expression in order to deal with pattern matching errors.

$$(Seq)\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'), q\rangle \mid \Pi}$$

$$(Send)\quad \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma \cup (p'', v); \langle p, (\theta', e'), q\rangle \mid \Pi}$$

$$(Receive)\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q) = (\theta_i, e_i, v)}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\backslash v\rangle \mid \Pi}$$

$$(Spawn)\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \mid \langle p', (id, \mathsf{apply}\ a/n\ (\overline{v_n})), [\,]\rangle \mid \Pi}$$

$$(Self)\quad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi}$$

$$(Sched)\quad \frac{}{\Gamma \cup \{(p, v)\}; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta, e), v : q\rangle \mid \Pi}$$

Figure 3.6: Standard semantics: system rules

κ—by abuse, κ is dealt with as a variable—so that the system rules of Figure 3.6 will eventually bind κ to its correct value:[5] the selected expression in rule *Receive* and a pid in rules *Spawn* and *Self*. In these cases, the label of the transition contains all the information needed by system rules to perform the evaluation at the system level, including the symbol κ. This *trick* allows us to keep the rules for expressions and systems separated (i.e., the semantics shown in Figures 3.4 and 3.5 is mostly independent from the rules in Figure 3.6), in contrast to other Erlang semantics, e.g., [19], where they are combined into a single transition relation.

Finally, we consider the system rules, which are depicted in Figure 3.6. In most of the transition rules, we consider an arbitrary system of the form $\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi$, where Γ is the global mailbox and $\langle p, (\theta, e), q\rangle \mid \Pi$ is a pool of processes that contains at least one process $\langle p, (\theta, e), q\rangle$. Let us briefly describe the system rules.

Rule *Seq* just updates the control (θ, e) of the considered process when a sequential expression is reduced using the expression rules.

Rule *Send* adds the pair (p'', v) to the global mailbox Γ instead of adding it to the queue of process p''. This is necessary to ensure that all possible message interleavings are correctly modeled (as discussed in Example 3.1). Observe that e'

[5] Note that κ takes values on the domain $expr \cup Pid$, in contrast to ordinary variables that can only be bound to values.

is usually different from v since e may have different nested operators. E.g., if e has the form "case p ! v of $\{\ldots\}$," then e' will be "case v of $\{\ldots\}$" with label $\mathsf{send}(\mathrm{p}, v)$.

In rule $Receive$, we use the auxiliary function matchrec to evaluate a receive expression. The main difference w.r.t. match is that matchrec also takes a queue q and returns the selected message v. More precisely, function matchrec scans the queue q looking for the *first* message v matching a pattern of the receive statement. Then, κ is bound to the expression in the selected clause, e_i, and the environment is extended with the matching substitution. If no message in the queue q matches any clause, then the rule is not applicable and the selected process cannot be reduced (i.e., it suspends). As in case expressions, we assume that the patterns can only contain fresh variables.

The rules presented so far allow one to store messages in the global mailbox, but not to remove messages from it. This is precisely the task of the scheduler, which is modeled by rule $Sched$. This rule nondeterministically chooses a pair (p, v) in the global mailbox Γ and delivers the message v to the target process p. Here, we deliberately ignore the restriction mentioned in Example 3.1: "the messages sent—directly—between two given processes arrive in the same order they were sent", since current implementations only guarantee it within the same node. In practice, ignoring this restriction amounts to consider that each process is potentially run in a different node. An alternative definition ensuring this restriction can be found in [109].

Example 3.4. Consider again the program shown in Example 3.1. Figures 3.7 and 3.8 show a derivation from "apply $\mathrm{main}/0$ ()" where the call to function target reduces to $\{\mathrm{world}, \mathrm{hello}\}$, as discussed in Example 3.1 (i.e., the interleaving shown in Figure 3.2 (b)). Processes' pids are denoted with $\mathrm{p1}$, $\mathrm{p2}$ and $\mathrm{p3}$. For clarity, we label each transition step with the applied rule and underline the reduced expression.

3.3.1 Erlang Concurrency

In order to define a causal-consistent reversible semantics for Erlang we need not only an interleaving semantics such as the one we just presented, but also a notion of concurrency (or, equivalently, the opposite notion of conflict). While concurrency is a main feature of Erlang, as far as we know no formal definition of the concurrency model of Erlang exists in the literature. We propose below one such definition.

Given systems s_1, s_2, we call $s_1 \hookrightarrow^* s_2$ a *derivation*. One-step derivations are simply called *transitions*. We use $d, d', d_1, \ldots$ to denote derivations and $t, t', t_1, \ldots$

$$
\begin{array}{lll}
 & \{\,\}; & \langle \mathrm{p1}, (id, \underline{\mathsf{apply}\ \mathrm{main}/0\ ()}), [\,]\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{p1}, (id, \mathsf{let}\ P2 = \underline{\mathsf{spawn}(\mathrm{echo}/0, [\,])}\ \mathsf{in}\ \ldots), [\,]\rangle \\
\hookrightarrow_{Spawn} & \{\,\}; & \langle \mathrm{p1}, (id, \underline{\mathsf{let}\ P2 = \mathrm{p2}\ \mathsf{in}\ \ldots}), [\,]\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply}\ \mathrm{echo}/0\ [\,]), [\,]\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}\}, \mathsf{let}\ P3 = \underline{\mathsf{spawn}(\mathrm{target}/0, [\,])}\ \mathsf{in}\ \ldots), [\,]\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply}\ \mathrm{echo}/0\ [\,]), [\,]\rangle \\
\hookrightarrow_{Spawn} & \{\,\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}\}, \underline{\mathsf{let}\ P3 = \mathrm{p3}\ \mathsf{in}\ \ldots}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply}\ \mathrm{echo}/0\ [\,]), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply}\ \mathrm{target}/0\ [\,]), [\,]\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \mathsf{let}\ \_ = \underline{P3}\ !\ \mathrm{hello}\ \mathsf{in}\ \ldots, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply}\ \mathrm{echo}/0\ [\,]), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply}\ \mathrm{target}/0\ [\,]), [\,]\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \mathsf{let}\ \_ = \underline{\mathrm{p3}\ !\ \mathrm{hello}}\ \mathsf{in}\ \ldots, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply}\ \mathrm{echo}/0\ [\,]), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply}\ \mathrm{target}/0\ [\,]), [\,]\rangle \\
\hookrightarrow_{Send} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \underline{\mathsf{let}\ \_ = \mathrm{hello}\ \mathsf{in}\ \ldots}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply}\ \mathrm{echo}/0\ [\,]), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply}\ \mathrm{target}/0\ [\,]), [\,]\rangle \\
\hookrightarrow_{Seq} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \underline{P2}\ !\ \{P3, \mathrm{world}\}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply}\ \mathrm{echo}/0\ [\,]), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply}\ \mathrm{target}/0\ [\,]), [\,]\rangle \\
\hookrightarrow_{Seq} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \mathrm{p2}\ !\ \{\underline{P3}, \mathrm{world}\}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply}\ \mathrm{echo}/0\ [\,]), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply}\ \mathrm{target}/0\ [\,]), [\,]\rangle \\
\hookrightarrow_{Seq} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \underline{\mathrm{p2}\ !\ \{\mathrm{p3}, \mathrm{world}\}}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{apply}\ \mathrm{echo}/0\ [\,]), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply}\ \mathrm{target}/0\ [\,]), [\,]\rangle \\
\hookrightarrow_{Send} & \{m_1, m_2\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \underline{\mathsf{apply}\ \mathrm{echo}/0\ [\,]}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{apply}\ \mathrm{target}/0\ [\,]), [\,]\rangle \\
\hookrightarrow_{Seq} & \{m_1, m_2\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{receive}\ \{P, M\} \to P\ !\ M\ \mathsf{end}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \underline{\mathsf{apply}\ \mathrm{target}/0\ [\,]}), [\,]\rangle \\
\hookrightarrow_{Seq} & \{m_1, m_2\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \mathsf{receive}\ \{P, M\} \to P\ !\ M\ \mathsf{end}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{receive}\ A \to \ldots\ \mathsf{end}), [\,]\rangle
\end{array}
$$

Figure 3.7: A derivation from "apply $\mathrm{main}/0\ ()$", where $m_1 = (\mathrm{p3}, \mathrm{hello})$, $m_2 = (\mathrm{p2}, \{\mathrm{p3}, \mathrm{world}\})$, and $m_3 = (\mathrm{p3}, \mathrm{world})$ (part 1/2)

$$
\begin{array}{lll}
\hookrightarrow_{Sched} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (id, \underline{\mathsf{receive}\ \{P, M\} \to P\,!\,M\ \mathsf{end}}), [\{\mathrm{p3}, \mathrm{world}\}]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{receive}\ A \to \ldots\ \mathsf{end}), [\,]\rangle \\
\hookrightarrow_{Receive} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (\{P \mapsto \mathrm{p3}, M \mapsto \mathrm{world}\}, \underline{P}\,!\,M), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{receive}\ A \to \ldots\ \mathsf{end}), [\,]\rangle \\
\hookrightarrow_{Seq} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (\{P \mapsto \mathrm{p3}, M \mapsto \mathrm{world}\}, \mathrm{p3}\,!\,\underline{M}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{receive}\ A \to \ldots\ \mathsf{end}), [\,]\rangle \\
\hookrightarrow_{Seq} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (\{P \mapsto \mathrm{p3}, M \mapsto \mathrm{world}\}, \underline{\mathrm{p3}\,!\,\mathrm{world}}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{receive}\ A \to \ldots\ \mathsf{end}), [\,]\rangle \\
\hookrightarrow_{Send} & \{m_1, m_3\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (\{P \mapsto \mathrm{p3}, M \mapsto \mathrm{world}\}, \mathrm{world}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \mathsf{receive}\ A \to \ldots\ \mathsf{end}), [\,]\rangle \\
\hookrightarrow_{Sched} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])\rangle \\
 & & \mid \langle \mathrm{p2}, (\{P \mapsto \mathrm{p3}, M \mapsto \mathrm{world}\}, \mathrm{world}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (id, \underline{\mathsf{receive}\ A \to \ldots\ \mathsf{end}}), [\mathrm{world}]\rangle \\
\hookrightarrow_{Receive} & \{m_1\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])), [\,]\rangle \\
 & & \mid \langle \mathrm{p2}, (\{P \mapsto \mathrm{p3}, M \mapsto \mathrm{world}\}, \mathrm{world}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (\{A \mapsto \mathrm{world}\}, \mathsf{receive}\ B \to \{A, B\}\ \mathsf{end}), [\,]\rangle \\
\hookrightarrow_{Sched} & \{\,\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])), [\,]\rangle \\
 & & \mid \langle \mathrm{p2}, (\{P \mapsto \mathrm{p3}, M \mapsto \mathrm{world}\}, \mathrm{world}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (\{A \mapsto \mathrm{world}\}, \underline{\mathsf{receive}\ B \to \{A, B\}\ \mathsf{end}}), [\mathrm{hello}]\rangle \\
\hookrightarrow_{Receive} & \{\,\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])), [\,]\rangle \\
 & & \mid \langle \mathrm{p2}, (\{P \mapsto \mathrm{p3}, M \mapsto \mathrm{world}\}, \mathrm{world}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (\{A \mapsto \mathrm{world}, B \mapsto \mathrm{hello}\}, \{\underline{A}, B\}), [\,]\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])), [\,]\rangle \\
 & & \mid \langle \mathrm{p2}, (\{P \mapsto \mathrm{p3}, M \mapsto \mathrm{world}\}, \mathrm{world}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (\{A \mapsto \mathrm{world}, B \mapsto \mathrm{hello}\}, \{\mathrm{world}, \underline{B}\}), [\,]\rangle \\
\hookrightarrow_{Seq} & \{\,\}; & \langle \mathrm{p1}, (\{P2 \mapsto \mathrm{p2}, P3 \mapsto \mathrm{p3}\}, \{\mathrm{p3}, \mathrm{world}\}, [\,])), [\,]\rangle \\
 & & \mid \langle \mathrm{p2}, (\{P \mapsto \mathrm{p3}, M \mapsto \mathrm{world}\}, \mathrm{world}), [\,]\rangle \\
 & & \mid \langle \mathrm{p3}, (\{A \mapsto \mathrm{world}, B \mapsto \mathrm{hello}\}, \{\mathrm{world}, \mathrm{hello}\}), [\,]\rangle
\end{array}
$$

Figure 3.8: A derivation from "apply $\mathrm{main}/0$ ()", where $m_1 = (\mathrm{p3}, \mathrm{hello})$, $m_2 = (\mathrm{p2}, \{\mathrm{p3}, \mathrm{world}\})$, and $m_3 = (\mathrm{p3}, \mathrm{world})$ (part 2/2)

for transitions. We label transitions as follows: $s_1 \hookrightarrow_{p,r} s_2$ where[6]

- p is the pid of the selected process in the transition or of the process to which a message is delivered (if the applied rule is $Sched$);
- r is the label of the applied transition rule.

We ignore some labels when they are clear from the context.

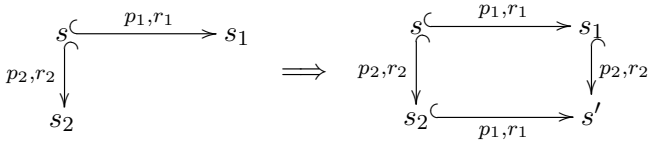
Given a derivation $d = (s_1 \hookrightarrow^* s_2)$, we define $\mathsf{init}(d) = s_1$ and $\mathsf{final}(d) = s_2$. Two derivations, d_1 and d_2, are *composable* if $\mathsf{final}(d_1) = \mathsf{init}(d_2)$. In this case, we let $d_1; d_2$ denote their composition with $d_1; d_2 = (s_1 \hookrightarrow s_2 \hookrightarrow \cdots \hookrightarrow s_n \hookrightarrow s_{n+1} \hookrightarrow \cdots \hookrightarrow s_m)$ if $d_1 = (s_1 \hookrightarrow s_2 \hookrightarrow \cdots \hookrightarrow s_n)$ and $d_2 = (s_n \hookrightarrow s_{n+1} \hookrightarrow \cdots \hookrightarrow s_m)$. Two derivations, d_1 and d_2, are said *coinitial* if $\mathsf{init}(d_1) = \mathsf{init}(d_2)$, and *cofinal* if $\mathsf{final}(d_1) = \mathsf{final}(d_2)$.

We let ϵ_s denote the zero-step derivation $s \hookrightarrow^* s$.

Definition 3.5 (Concurrent transitions)**.** Given two coinitial transitions, $t_1 = (s \hookrightarrow_{p_1,r_1} s_1)$ and $t_2 = (s \hookrightarrow_{p_2,r_2} s_2)$, we say that they are *in conflict* if they consider the same process, i.e., $p_1 = p_2$, and either $r_1 = r_2 = Sched$ or one transition applies rule $Sched$ and the other transition applies rule $Receive$. Two coinitial transitions are *concurrent* if they are not in conflict.

We show below that our definition of concurrent transitions makes sense.

Lemma 3.6 (Square lemma)**.** *Given two coinitial concurrent transitions $t_1 = (s \hookrightarrow_{p_1,r_1} s_1)$ and $t_2 = (s \hookrightarrow_{p_2,r_2} s_2)$, there exist two cofinal transitions $t_2/t_1 = (s_1 \hookrightarrow_{p_2,r_2} s')$ and $t_1/t_2 = (s_2 \hookrightarrow_{p_1,r_1} s')$. Graphically,*



Proof. We have the following cases:

- Two transitions t_1 and t_2 where $r_1 \neq Sched$ and $r_2 \neq Sched$. Trivially, they apply to different processes, i.e., $p_1 \neq p_2$. Then, we can easily prove that by applying rule r_2 to p_1 in s_1 and rule r_1 to p_2 in s_2 we have two transitions t_1/t_2 and t_2/t_1 which are cofinal.

[6]Note that p, r in $\hookrightarrow_{p,r}$ are not parameters of the transition relation $\hookrightarrow$ but just labels with some information on the reduction step. This information becomes useful to formally define the notion of concurrent transitions.

- One transition t_1 which applies rule $r_1 = \mathit{Sched}$ to deliver message v_1 to process $p_1 = \mathrm{p}$, and another transition which applies a rule r_2 different from Sched. All cases but $r_2 = \mathit{Receive}$ with $p_2 = \mathrm{p}$ are straightforward. This last case, though, cannot happen since transitions using rules Sched and $\mathit{Receive}$ are not concurrent.
- Two transitions t_1 and t_2 with rules $r_1 = r_2 = \mathit{Sched}$ delivering messages v_1 and v_2, respectively. Since the transitions are concurrent, they should deliver the messages to different processes, i.e., $p_1 \neq p_2$. Therefore, we can see that delivering v_2 from s_1 and v_1 from s_2 we get two cofinal transitions.

□

We remark here that other definitions of concurrent transitions are possible. Changing the concurrency model would require to change the stored information in the reversible semantics in order to preserve causal consistency. We have chosen the notion above since it is reasonably simple to define and to work with, and captures most of the pairs of coinitial transitions that satisfy the Square lemma.

3.4 A Reversible Semantics for Erlang

In this section, we introduce a reversible—uncontrolled—semantics for the considered language. Thanks to the modular design of the concrete semantics, the transition rules for the language expressions need not be changed in order to define the reversible semantics.

To be precise, in this section we introduce two transition relations: $\rightharpoonup$ and $\leftharpoondown$. The first relation, $\rightharpoonup$, is a conservative extension of the standard semantics $\hookrightarrow$ (Figure 3.6) to also include some additional information in the states, following a typical Landauer embedding. We refer to $\rightharpoonup$ as the *forward* reversible semantics (or simply the forward semantics). In contrast, the second relation, $\leftharpoondown$, proceeds in the backward direction, "undoing" actions step by step. We refer to $\leftharpoondown$ as the backward (reversible) semantics. We denote the union $\rightharpoonup \cup \leftharpoondown$ by $\rightleftharpoons$.

In the next section, we will introduce a rollback operator that starts a reversible computation for a process. In order to avoid undoing all actions until the beginning of the process, we will also let the programmer introduce *checkpoints*. Syntactically, they are denoted with the built-in function check, which takes an identifier `t` as an argument. Such identifiers are supposed to be unique in the program. Given an expression, expr, we can introduce a checkpoint by replacing expr with "let $X =$ check(t) in expr". A call of the form check(t) just returns `t` (see below).

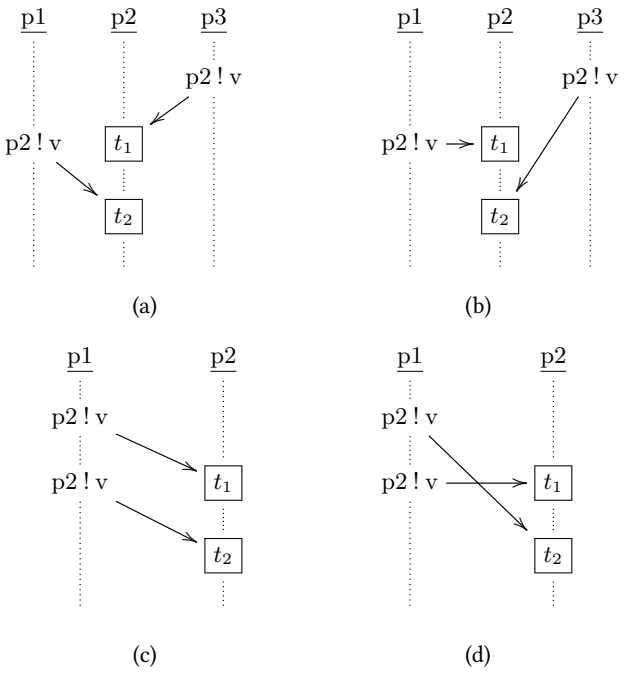


Figure 3.9: Interleavings and the need for unique identifiers for messages

In the following, we consider that the rules to evaluate the language expressions (Figures 3.4 and 3.5) are extended with the following rule:

$$(Check)\ \frac{}{\theta, \mathsf{check(t)} \xrightarrow{\mathsf{check(t)}} \theta, \mathtt{t}}$$

In this section, we will mostly ignore checkpoints, but they will become relevant in the next section.

The most significant novelty in the forward semantics is that messages now include a unique identifier (e.g., a timestamp λ). Let us illustrate with some examples why we introduce these identifiers. Consider first diagram (a) in Figure 3.9, where two different processes, p1 and p3, send the same message v to process p2. In order to undo the action p2 ! v in process p3, one needs to first undo all actions of p2 up to $\boxed{t_1}$ (to ensure causal consistency). However, currently, messages only store information about the target process and the value sent, therefore it is not possible to know whether it is safe to stop undoing actions at $\boxed{t_1}$ or at $\boxed{t_2}$. Actually, the situations in diagrams (a) and (b) are not distinguishable. In

this case, it would suffice to add the pid of the sender to every message in order to avoid the confusion. However, this is not always sufficient. Consider now diagram (c). Here, a process p1 sends two identical messages to another process p2 (which is not unusual, say an "ack" after receiving a request). In this case, in order to undo the first action p2 ! v of process p1 one needs to undo all actions of process p2 up to $\boxed{t_1}$. However, we cannot distinguish $\boxed{t_1}$ from $\boxed{t_2}$ unless some additional information is taken into account (and considering triples of the form $(source\_process\_pid, target\_process\_pid, message)$ would not help). Therefore, one needs to introduce some unique identifier in order to precisely distinguish case (c) from case (d).

Of course, we could have a less precise semantics where just the message, v, is observable. However, that would make the backward semantics unpredictable (e.g., we could often undo the "wrong" message delivery). Also, defining the corresponding notion of *conflicting* transitions (see Definition 3.12 below) would be challenging, since one would like to have only a conflict between the sending of a message v and the "last" delivery of the same message v, which would be very tricky. Therefore, in this paper, we prefer to assume that messages can be uniquely distinguished.

The transition rules of the forward reversible semantics can be found in Figure 3.10. Processes now include a memory (or *history*) h that records the intermediate states of a process, and messages have an associated unique identifier. In the memory, we use terms headed by constructors τ, check, send, rec, spawn, and self to record the steps performed by the forward semantics. Note that we could optimize the information stored in these terms by following a strategy similar to that in [97, 108, 133] for the reversibility of functional expressions, but this is orthogonal to our purpose in this paper, so we focus mainly on the concurrent actions. Note also that the auxiliary function matchrec now deals with messages of the form $\{v, \lambda\}$, which is a trivial extension of the original function in the standard semantics by just ignoring λ when computing the first matching message.

Example 3.7. Let us consider the program shown in Figure 3.12 (a), together with the execution trace sketched in Figure 3.12 (b). Figure 3.13 shows a high level account of the corresponding derivation under the forward semantics. For clarity, we consider the following conventions:

- Processes client1, client2 and server are denoted with c1, c2 and s, respectively.
- In the processes, we do not show the current environment. Moreover, we use the notation $C[e]$ to denote that e is the redex to be reduced next and $C[\,]$ is

$$(Seq)\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, h, (\theta', e'), q\rangle \mid \Pi}$$

$$(Check)\quad \frac{\theta, e \xrightarrow{\mathsf{check(t)}} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{check}(\theta, e, \mathtt{t}) : h, (\theta', e'), q\rangle \mid \Pi}$$

$$(Send)\quad \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e' \quad \lambda \text{ is a fresh identifier}}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma \cup (p'', \{v, \lambda\}); \langle p, \mathsf{send}(\theta, e, p'', \{v, \lambda\}) : h, (\theta', e'), q\rangle \mid \Pi}$$

$$(Receive)\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q) = (\theta_i, e_i, \{v, \lambda\})}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{rec}(\theta, e, \{v, \lambda\}, q) : h, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\backslash \{v, \lambda\}\rangle \mid \Pi}$$

$$(Spawn)\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\begin{array}{l}\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \quad \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \\ \qquad\qquad \mid \langle p', [\,], (id, \mathsf{apply}\ a/n\ (\overline{v_n})), [\,]\rangle \mid \Pi\end{array}}$$

$$(Self)\quad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{self}(\theta, e) : h, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi}$$

$$(Sched)\quad \frac{}{\Gamma \cup \{(p, \{v, \lambda\})\}; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, h, (\theta, e), \{v, \lambda\} : q\rangle \mid \Pi}$$

Figure 3.10: Forward reversible semantics

$$(\overline{Seq})\quad \Gamma; \langle p, \tau(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$

$$(\overline{Check})\quad \Gamma; \langle p, \mathsf{check}(\theta, e, \mathtt{t}) : h, (\theta', e'), q\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$

$$(\overline{Send})\quad \Gamma \cup \{(p'', \{v, \lambda\})\}; \langle p, \mathsf{send}(\theta, e, p'', \{v, \lambda\}) : h, (\theta', e'), q\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$

$$(\overline{Receive})\quad \Gamma; \langle p, \mathsf{rec}(\theta, e, \{v, \lambda\}, q) : h, (\theta', e'), q \backslash\backslash \{v, \lambda\}\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$

$$(\overline{Spawn})\quad \begin{array}{l}\Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, (\theta', e'), q\rangle \mid \langle p', [\,], (id, e''), [\,]\rangle \mid \Pi \\ \qquad \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi\end{array}$$

$$(\overline{Self})\quad \Gamma; \langle p, \mathsf{self}(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi$$

$$(\overline{Sched})\quad \begin{array}{l}\Gamma; \langle p, h, (\theta, e), \{v, \lambda\} : q\rangle \mid \Pi \leftharpoondown \Gamma \cup (p, \{v, \lambda\}); \langle p, h, (\theta, e), q\rangle \mid \Pi \\ \qquad \text{if the topmost } \mathsf{rec}(\ldots) \text{ item in } h \text{ (if any) has the} \\ \qquad \text{form } \mathsf{rec}(\theta', e', \{v', \lambda'\}, q') \text{ with } q' \backslash\backslash \{v', \lambda'\} \neq \{v, \lambda\} : q\end{array}$$

Figure 3.11: Backward reversible semantics

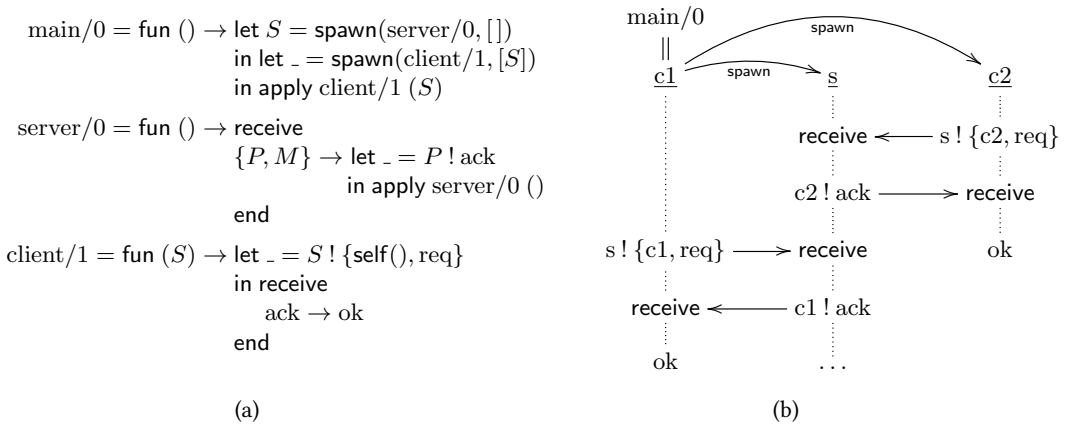


Figure 3.12: A simple client-server

an arbitrary (possibly empty) context. We also underline the selected redex when there are more than one (e.g., a redex in each process).

- In the histories, some arguments are denoted by "_" since they are not relevant in the current derivation.
- Finally, we only show the steps performed with rules *Spawn*, *Send*, *Receive* and *Sched*; the transition relation is labeled with the applied rule.

We now prove that the forward semantics $\rightharpoonup$ is a conservative extension of the standard semantics $\hookrightarrow$.

In order to state the result, we let $\mathsf{del}(s)$ denote the system that results from s by removing the histories of the processes; formally, $\mathsf{del}(\Gamma; \Pi) = \Gamma; \mathsf{del}'(\Pi)$, where

$$\begin{array}{lcl} \mathsf{del}'(\langle p, h, (\theta, e), q\rangle) & = & \langle p, (\theta, e), q\rangle \\ \mathsf{del}'(\langle p, h, (\theta, e), q\rangle \mid \Pi) & = & \langle p, (\theta, e), q\rangle \mid \mathsf{del}'(\Pi) \end{array}$$

where we assume that Π is not empty.

We can now state the conservative extension result.

Theorem 3.8. *Let s_1 be a system of the reversible semantics without occurrences of "*check*" and $s_1' = \mathsf{del}(s_1)$ a system of the standard semantics. Then, $s_1' \hookrightarrow^* s_2'$ iff $s_1 \rightharpoonup^* s_2$ and $\mathsf{del}(s_2) = s_2'$.*

Proof. The proof is straightforward since the transition rules of the forward semantics in Figure 3.10 are just annotated versions of the corresponding rules in Figure 3.6. The only tricky point is noticing that the introduction of unique identifiers

$$
\begin{array}{lll}
 & \{\,\}; & \langle \mathrm{c1}, [\,], (id, C[\mathsf{apply}\ \mathrm{main/0}\ ()]), [\,]\rangle \\
\rightharpoonup^{*} & \{\,\}; & \langle \mathrm{c1}, [\,], (\_, C[\mathsf{spawn}(\mathrm{server/0}, [\,])]), [\,]\rangle \\
\rightharpoonup_{Spawn} & \{\,\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{spawn}(\mathrm{client/1}, \underline{[\mathrm{s}]})]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\,], (\_, C[\mathsf{receive}\ \{P, M\} \to \ldots]), [\,]\rangle \\
\rightharpoonup_{Spawn} & \{\,\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\,], (\_, C[\mathsf{receive}\ \{P, M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\,], (\_, C[\underline{\mathrm{s} \,!\, \{\mathrm{c2}, \mathrm{req}\}}]), [\,]\rangle \\
\rightharpoonup_{Send} & \{(\mathrm{s}, m_1)\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\,], (\_, C[\mathsf{receive}\ \{P, M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle \\
\rightharpoonup_{Sched} & \{\,\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\,], (\_, C[\underline{\mathsf{receive}\ \{P, M\} \to \ldots}]), [m_1]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle \\
\rightharpoonup_{Receive} & \{\,\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{rec}(\_,\_, m_1, [m_1])], (\_, C[\underline{\mathrm{c2} \,!\, \mathrm{ack}}]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle \\
\rightharpoonup_{Send} & \{(\mathrm{c2}, m_2)\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c2}, m_2), \mathsf{rec}(\_,\_, m_1, [m_1])], (\_, C[\mathsf{receive}\ \{P, M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle \\
\rightharpoonup_{Sched} & \{\,\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c2}, m_2), \mathsf{rec}(\_,\_, m_1, [m_1])], (\_, C[\mathsf{receive}\ \{P, M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, C[\underline{\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}}]), [m_2]\rangle \\
\rightharpoonup_{Receive} & \{\,\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\underline{\mathrm{s} \,!\, \{\mathrm{c1}, \mathrm{req}\}}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c2}, m_2), \mathsf{rec}(\_,\_, m_1, [m_1])], (\_, C[\mathsf{receive}\ \{P, M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_, m_2, [m_2]), \mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightharpoonup_{Send} & \{(\mathrm{s}, m_3)\}; & \langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s}, m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c2}, m_2), \mathsf{rec}(\_,\_, m_1, [m_1])], (\_, C[\mathsf{receive}\ \{P, M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_, m_2, [m_2]), \mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightharpoonup_{Sched} & \{\,\}; & \langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s}, m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c2}, m_2), \mathsf{rec}(\_,\_, m_1, [m_1])], (\_, C[\underline{\mathsf{receive}\ \{P, M\} \to \ldots}]), [m_3]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_, m_2, [m_2]), \mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightharpoonup_{Receive} & \{\,\}; & \langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s}, m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{rec}(\_,\_, m_3, [m_3]), \mathsf{send}(\_,\_,\mathrm{c2}, m_2), \mathsf{rec}(\_,\_, m_1, [m_1])], (\_, C[\underline{\mathrm{c1} \,!\, \mathrm{ack}}]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_, m_2, [m_2]), \mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightharpoonup_{Send} & \{(\mathrm{c1}, m_4)\}; & \langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s}, m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1}, m_4), \mathsf{rec}(\_,\_, m_3, [m_3]), \mathsf{send}(\_,\_,\mathrm{c2}, m_2), \mathsf{rec}(\_,\_, m_1, [m_1])], \\
 & & \ (\_, C[\mathsf{receive}\ \{P, M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_, m_2, [m_2]), \mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightharpoonup_{Sched} & \{\,\}; & \langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s}, m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\underline{\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}}]), [m_4]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1}, m_4), \mathsf{rec}(\_,\_, m_3, [m_3]), \mathsf{send}(\_,\_,\mathrm{c2}, m_2), \mathsf{rec}(\_,\_, m_1, [m_1])], \\
 & & \ (\_, C[\mathsf{receive}\ \{P, M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_, m_2, [m_2]), \mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightharpoonup_{Receive} & \{\,\}; & \langle \mathrm{c1}, [\mathsf{rec}(\_,\_, m_4, [m_4]), \mathsf{send}(\_,\_,\mathrm{s}, m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, \mathrm{ok}), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1}, m_4), \mathsf{rec}(\_,\_, m_3, [m_3]), \mathsf{send}(\_,\_,\mathrm{c2}, m_2), \mathsf{rec}(\_,\_, m_1, [m_1])], \\
 & & \ (\_, C[\mathsf{receive}\ \{P, M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_, m_2, [m_2]), \mathsf{send}(\_,\_,\mathrm{s}, m_1)], (\_, \mathrm{ok}), [\,]\rangle
\end{array}
$$

Figure 3.13: A derivation under the forward semantics, with $m_1 = \{\{\mathrm{c2}, \mathrm{req}\}, 1\}$, $m_2 = \{\mathrm{ack}, 2\}$, $m_3 = \{\{\mathrm{c1}, \mathrm{req}\}, 3\}$, and $m_4 = \{\mathrm{ack}, 4\}$.

for messages does not change the behavior of rule $Receive$ since function matchrec always returns the oldest occurrence (in terms of position in the queue) of the selected message. □

The transition rules of the backward semantics are shown in Figure 3.11. In general, all rules restore the control (and, if it applies, also the queue) of the process. Nevertheless, let us briefly discuss a few particular situations:

- First, observe that rule $\overline{Send}$ can only be applied when the message sent is in the global mailbox. If this is not the case (i.e., the message has been delivered using rule $Sched$), then we should first apply backward steps to the receiver process until, eventually, the application of rule $\overline{Sched}$ puts the message back into the global mailbox and rule $\overline{Send}$ becomes applicable. This is required to ensure causal consistency. In the next section, we will introduce a particular strategy that achieves this effect in a controlled manner.

- A similar situation occurs with rule $\overline{Spawn}$. Given a process p with a history item $\mathsf{spawn}(\theta, e, p')$, rule $\overline{Spawn}$ cannot be applied until the history and the queue of process p' are both empty. Therefore, one should first apply a number of backward steps to process p' in order to be able to undo the spawn item. We note that there is no need to require that no message targeting the process p' (which would become an *orphan* message) is in the global mailbox: in order to send such a message the pid p' is needed, hence the sending of the message depends on the spawn and, thus, it must be undone beforehand.

- Observe too that rule $\overline{Receive}$ can only be applied when the queue of the process is exactly the same queue that was obtained after applying the corresponding (forward) $Receive$ step. This is necessary in order to ensure that the restored queue is indeed the right one (note that adding the message to an arbitrary queue would not work since we do not know the "right" position for the message).

- In principle, there is some degree of freedom in the application of rule $\overline{Sched}$ since it does not interfere with the remaining rules, except for $\overline{Receive}$ and other applications of $\overline{Sched}$. Therefore, the application of rule $\overline{Sched}$ can be switched with the application of any other backward rule except for $\overline{Receive}$ or another $\overline{Sched}$. The fact that two $\overline{Sched}$ (involving the same process) do not commute is ensured since $\overline{Sched}$ always applies to the most recent message of a queue. The fact that a $\overline{Sched}$ and a $\overline{Receive}$ do not commute is ensured since the side condition of $\overline{Sched}$ checks that there is no $\mathsf{rec}(\ldots)$ item in the

history of the process that can be used to apply rule $\overline{Receive}$ with the current queue. Hence, their applicability conditions do not overlap.

Example 3.9. Consider again the program shown in Figure 3.12. By starting from the last system in the forward derivation shown in Figure 3.13, we may construct the backward derivation shown in Figure 3.14. Observe that it does not strictly follow the inverse order of the derivation shown in Figure 3.13. Actually, a derivation that undoes the steps in the precise inverse order exists, but it is not the only possibility. We will characterize later on (see Corollary 3.22) which orders are allowed and which are not. In Figure 3.14, besides following the same conventions of Example 3.7, for clarity, we underline the selected history item to be undone or the element in the queue to be removed (when the applied rule is $\overline{Sched}$).

3.4.1 Properties of the Uncontrolled Reversible Semantics

In the following, we prove several properties of our reversible semantics, including its *causal consistency*, an essential property for reversible concurrent calculi [35].

Given systems s_1, s_2, we call $s_1 \rightharpoonup^* s_2$ a *forward* derivation and $s_2 \leftharpoondown^* s_1$ a *backward* derivation. A derivation potentially including both forward and backward steps is denoted by $s_1 \rightleftharpoons^* s_2$. We label transitions as follows: $s_1 \rightleftharpoons_{p,r,k} s_2$ where

- p, r are the pid of the selected process and the label of the applied rule, respectively, as in Section 3.3.1,
- k is a history item if the applied rule was different from $Sched$ and $\overline{Sched}$, and
- $k = \mathsf{sched}(\{v, \lambda\})$ when the applied rule was $Sched$ or $\overline{Sched}$, where $\{v, \lambda\}$ is the message delivered or put back into Γ. Note that this information is available when applying the rule.

We ignore some labels when they are clear from the context.

We extend the definitions of functions init and final from Section 3.3.1 to reversible derivations in the natural way. The notions of composable, coinitial and cofinal derivations are extended also in a straightforward manner.

Given a rule label r, we let $\overline{r}$ denote its reverse version, i.e., if $r = Send$ then $\overline{r} = \overline{Send}$ and vice versa (if $r = \overline{Send}$ then $\overline{r} = Send$). Also, given a transition t, we let $\overline{t} = (s' \leftharpoondown_{p,\overline{r},k} s)$ if $t = (s \rightharpoonup_{p,r,k} s')$ and $\overline{t} = (s' \rightharpoonup_{p,r,k} s)$ if $t = (s \leftharpoondown_{p,\overline{r},k} s')$. We say that $\overline{t}$ is the *inverse* of t. This notation is naturally extended to derivations. We let ϵ_s denote the zero-step derivation $s \rightleftharpoons^* s$.

	$\{\ \}$;	$\langle \mathrm{c1}, [\mathsf{rec}(\_,\_,m_4,[m_4]), \mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, \mathrm{ok}), [\,]\rangle$ $\mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])],$ $(\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle$ $\mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, \mathrm{ok}), [\,]\rangle$
$\leftharpoondown_{\overline{Receive}}$	$\{\ \}$;	$\langle \mathrm{c1}, [\mathsf{rec}(\_,\_,m_4,[m_4]), \mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, \mathrm{ok}), [\,]\rangle$ $\mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])],$ $(\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle$ $\mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [m_2]\rangle$
$\leftharpoondown_{\overline{Receive}}$	$\{\ \}$;	$\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [m_4]\rangle$ $\mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])],$ $(\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle$ $\mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [m_2]\rangle$
$\leftharpoondown_{\overline{Sched}}$	$\{(\mathrm{c1}, m_4)\}$;	$\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$ $\mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])],$ $(\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle$ $\mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [m_2]\rangle$
$\leftharpoondown_{\overline{Sched}}$	$\{(\mathrm{c2}, m_2), (\mathrm{c1}, m_4)\}$;	$\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$ $\mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])],$ $(\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle$ $\mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$
$\leftharpoondown_{\overline{Send}}$	$\{(\mathrm{c2}, m_2)\}$;	$\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$ $\mid \langle \mathrm{s}, [\mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathrm{c1}\,!\,\mathrm{ack}]), [\,]\rangle$ $\mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$
$\leftharpoondown_{\overline{Receive}}$	$\{(\mathrm{c2}, m_2)\}$;	$\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$ $\mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [m_3]\rangle$ $\mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$
$\leftharpoondown_{\overline{Sched}}$	$\{(\mathrm{s}, m_3), (\mathrm{c2}, m_2)\}$;	$\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$ $\mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle$ $\mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$
$\leftharpoondown_{\overline{Send}}$	$\{(\mathrm{s}, m_3)\}$;	$\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$ $\mid \langle \mathrm{s}, [\mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathrm{c2}\,!\,\mathrm{ack}]), [\,]\rangle$ $\mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$
$\leftharpoondown_{\overline{Receive}}$	$\{(\mathrm{s}, m_3)\}$;	$\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$ $\mid \langle \mathrm{s}, [\,], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [m_1]\rangle$ $\mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$
$\leftharpoondown_{\overline{Sched}}$	$\{(\mathrm{s}, m_1), (\mathrm{s}, m_3)\}$;	$\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$ $\mid \langle \mathrm{s}, [\,], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle$ $\mid \langle \mathrm{c2}, [\mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$
$\leftharpoondown_{\overline{Send}}$	$\{(\mathrm{s}, m_3)\}$;	$\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{receive\ ack} \to \mathrm{ok}]), [\,]\rangle$ $\mid \langle \mathrm{s}, [\,], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle$ $\mid \langle \mathrm{c2}, [\,], (\_, C[\mathrm{s}\,!\,\{\mathrm{c2}, \mathrm{req}\}]), [\,]\rangle$
$\leftharpoondown_{\overline{Send}}$	$\{\ \}$;	$\langle \mathrm{c1}, [\mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathrm{s}\,!\,\{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle$ $\mid \langle \mathrm{s}, [\,], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle$ $\mid \langle \mathrm{c2}, [\,], (\_, C[\mathrm{s}\,!\,\{\mathrm{c2}, \mathrm{req}\}]), [\,]\rangle$
$\leftharpoondown_{\overline{Spawn}}$	$\{\ \}$;	$\langle \mathrm{c1}, [\mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{spawn}(\mathrm{client}/1, [\mathrm{s}])]), [\,]\rangle$ $\mid \langle \mathrm{s}, [\,], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle$
$\leftharpoondown_{\overline{Spawn}}$	$\{\ \}$;	$\langle \mathrm{c1}, [\,], (\_, C[\mathsf{spawn}(\mathrm{server}/0, [\,])]), [\,]\rangle$
$\leftharpoondown^*$	$\{\ \}$;	$\langle \mathrm{c1}, [\,], (\_, C[\mathsf{apply}\ \mathrm{main}/0\ ()]), [\,]\rangle$

Figure 3.14: A derivation under the backward semantics, with $m_1 = \{\{\mathrm{c2}, \mathrm{req}\}, 1\}$, $m_2 = \{\mathrm{ack}, 2\}$, $m_3 = \{\{\mathrm{c1}, \mathrm{req}\}, 3\}$, and $m_4 = \{\mathrm{ack}, 4\}$.

In the following we restrict the attention to systems reachable from the execution of a program:

Definition 3.10 (Reachable systems)**.** A system is *initial* if it is composed by a single process, and this process has an empty history and an empty queue; furthermore the global mailbox is empty. A system s is *reachable* if there exists an initial system s_0 and a derivation $s_0 \rightleftharpoons^* s$ using the rules corresponding to a given program.

Moreover, for simplicity, we also consider an implicit, fixed program in the technical results, that is we fix the function μ in the semantics of expressions.

The next lemma proves that every forward (resp. backward) transition can be undone by a backward (resp. forward) transition.

Lemma 3.11 (Loop lemma)**.** *For every pair of reachable systems, s_1 and s_2, we have $s_1 \rightharpoonup_{p,r,k} s_2$ iff $s_2 \rightharpoondown_{p,\overline{r},k} s_1$.*

Proof. The proof is by case analysis on the applied rule. We discuss below the most interesting cases.

- Rule $Sched$: notice that the queue of a process is changed only by rule $Receive$ (which removes messages) and $Sched$ (which adds messages). Since, after the last $Receive$ at least one message has been added, then the side condition of rule $\overline{Sched}$ is always verified.
- Rule $\overline{Seq}$: one has to check that the restored control (θ, e) can indeed perform a sequential step to (θ', e'). This always holds for reachable systems. An analogous check needs to be done for all backward rules.

□

The following notion of concurrent transitions allows us to characterize which actions can be switched without changing the semantics of a computation. It extends the same notion from the standard semantics (cf. Definition 3.5) to the reversible semantics.

Definition 3.12 (Concurrent transitions)**.** Given two coinitial transitions, $t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$, we say that they are *in conflict* if at least one of the following conditions holds:

- **both transitions are forward**, they consider the same process, i.e., $p_1 = p_2$, and either $r_1 = r_2 = Sched$ or one transition applies rule $Sched$ and the other transition applies rule $Receive$.

- one is a **forward** transition that applies to a process p, say $p_1 = \mathrm{p}$, and the other one is a **backward** transition that undoes the creation of p, i.e., $p_2 = \mathrm{p}' \neq \mathrm{p}$, $r_2 = \overline{Spawn}$ and $k_2 = \mathsf{spawn}(\theta, e, \mathrm{p})$ for some control (θ, e);

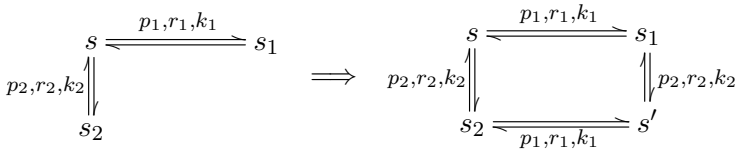
- one is a **forward** transition that delivers a message $\{v, \lambda\}$ to a process p, say $p_1 = \mathrm{p}$, $r_1 = Sched$ and $k_1 = \mathsf{sched}(\{v, \lambda\})$, and the other one is a **backward** transition that undoes the sending $\{v, \lambda\}$ to p, i.e., $p_2 = \mathrm{p}'$ (note that $\mathrm{p} = \mathrm{p}'$ if the message is sent to its own sender), $r_2 = \overline{Send}$ and $k_2 = \mathsf{send}(\theta, e, \mathrm{p}, \{v, \lambda\})$ for some control (θ, e);

- one is a **forward** transition and the other one is a **backward** transition such that $p_1 = p_2$ and either i) both applied rules are different from both $Sched$ and $\overline{Sched}$, i.e., $\{r_1, r_2\} \cap \{Sched, \overline{Sched}\} = \varnothing$; ii) one rule is $Sched$ and the other one is $\overline{Sched}$; iii) one rule is $Sched$ and the other one is $\overline{Receive}$; or iv) one rule is $\overline{Sched}$ and the other one is $Receive$.

Two coinitial transitions are *concurrent* if they are not in conflict. Note that two coinitial backward transitions are always concurrent.

The following lemma (the counterpart of Lemma 3.13 for the standard semantics) is a key result to prove the causal consistency of the semantics.

Lemma 3.13 (Square lemma). *Given two coinitial concurrent transitions* $t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ *and* $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$*, there exist two cofinal transitions* $t_2/t_1 = (s_1 \rightleftharpoons_{p_2,r_2,k_2} s')$ *and* $t_1/t_2 = (s_2 \rightleftharpoons_{p_1,r_1,k_1} s')$*. Graphically,*



Proof. We distinguish the following cases depending on the applied rules:

(1) Two forward transitions. Then, we have the following cases:

- Two transitions t_1 and t_2 where $r_1 \neq Sched$ and $r_2 \neq Sched$. Trivially, they apply to different processes, i.e., $p_1 \neq p_2$. Then, we can easily prove that by applying rule r_2 to p_1 in s_1 and rule r_1 to p_2 in s_2 we have two transitions t_1/t_2 and t_2/t_1 which produce the corresponding history items and are cofinal.

- One transition t_1 which applies rule $r_1 = \mathit{Sched}$ to deliver message $\{v_1, \lambda_1\}$ to process $p_1 = \mathrm{p}$, and another transition which applies a rule r_2 different from Sched. All cases but $r_2 = \mathit{Receive}$ with $p_2 = \mathrm{p}$ and $k_2 = \mathsf{rec}(\theta, e, \{v_2, \lambda_2\}, q)$ are straightforward. Note that $\lambda_1 \neq \lambda_2$ since these identifiers are unique. Here, by applying rule $\mathit{Receive}$ to s_1 and rule Sched to s_2 we will end up with the same mailbox in p (since it is a FIFO queue). However, the history item $\mathsf{rec}(\theta, e, \{v_2, \lambda_2\}, q')$ will be necessarily different since $q \neq q'$ by the application of rule Sched. This situation, though, cannot happen since transitions using rules Sched and $\mathit{Receive}$ are not concurrent.

- Two transitions t_1 and t_2 with rules $r_1 = r_2 = \mathit{Sched}$ delivering messages $\{v_1, \lambda_1\}$ and $\{v_2, \lambda_2\}$, respectively. Since the transitions are concurrent, they should deliver the messages to different processes, i.e., $p_1 \neq p_2$. Therefore, we can easily prove that delivering $\{v_2, \lambda_2\}$ from s_1 and $\{v_1, \lambda_1\}$ from s_2 we get two cofinal transitions.

(2) One forward transition and one backward transition. Then, we distinguish the following cases:

- If the two transitions apply to the same process, i.e., $p_1 = p_2$, then, since they are concurrent, we can only have $r_1 = \mathit{Sched}$ and a rule different from both $\overline{\mathit{Sched}}$ and $\overline{\mathit{Receive}}$, or $r_1 = \overline{\mathit{Sched}}$ and a rule different from both Sched and $\mathit{Receive}$. In these cases, the claim follows easily by a case distinction on the applied rules.

- Let us now consider that the transitions apply to different processes, i.e., $p_1 \neq p_2$, and the applied rules are different from $\mathit{Sched}, \overline{\mathit{Sched}}$. In this case, the claim follows easily except when one transition considers a process p and the other one undoes the spawning of the same process p. This case, however, is not allowed since the transitions are concurrent.

- Finally, let us consider that the transitions apply to different processes, i.e., $p_1 \neq p_2$, and that one transition applies rule Sched to deliver a message $\{v, \lambda\}$ from sender p to receiver p′, i.e., $p_1 = \mathrm{p}'$, $r_1 = \mathit{Sched}$ and $k_1 = \mathsf{sched}(\{v, \lambda\})$. In this case, the other transition should apply a rule r_2 different from $\overline{\mathit{Send}}$ with $k_2 = \mathsf{send}(\theta, e, \mathrm{p}', \{v, \lambda\})$ for some control (θ, e) since, otherwise, the transitions would not be concurrent. In any other case, one can easily prove that by applying r_2 to s_1 and Sched to s_2 we get two cofinal transitions.

(3) Two backward transitions. We distinguish the following cases:

- If the two transitions apply to different processes, the claim follows easily.
- Let us now consider that they apply to the same process, i.e., $p_1 = p_2$ and that the applied rules are different from $\overline{Sched}$. This case is not possible since, given a system, only one backward transition rule different from $\overline{Sched}$ is applicable (i.e., the one that corresponds to the last item in the history).
- Let us consider that both transitions apply to the same process and that both are applications of rule $\overline{Sched}$. This case is not possible since rule $\overline{Sched}$ can only take the newest message from the local queue of the process, and thus only one rule $\overline{Sched}$ can be applied to a given process.
- Finally, consider that both transitions apply to the same process and only one of them applies rule $\overline{Sched}$. In this case, the only non-trivial case is when the other applied rule is $\overline{Receive}$, since both change the local queue of the process. However, this case is not allowed by the backward semantics, since the conditions to apply rule $\overline{Sched}$ and rule $\overline{Receive}$ are non-overlapping.

□

Corollary 3.14 (Backward confluence)**.** *Given two backward derivations $s \leftharpoondown^* s_1$ and $s \leftharpoondown^* s_2$ there exist s_3 and two backward derivations $s_1 \leftharpoondown^* s_3$ and $s_2 \leftharpoondown^* s_3$.*

Proof. By iterating the square lemma (Lemma 3.13), noticing that backward transitions are always concurrent. This is a standard result for abstract relations (see, e.g., [9] and the original work by Rosen [120]), where confluence is implied by the *diamond property* (the square lemma in our work). □

The notion of concurrent transitions for the reversible semantics is a natural extension of the same notion for the standard semantics:

Lemma 3.15. *Let t_1 and t_2 be two forward coinitial transitions using the reversible semantics, and let t'_1 and t'_2 be their counterpart in the standard semantics obtained by removing the histories and the unique identifiers for messages. Then, t_1 and t_2 are concurrent iff t'_1 and t'_2 are.*

Proof. The proof is straightforward since Definition 3.5 and the first case of Definition 3.12 are perfectly analogous. □

The next result is used to switch the successive application of two transition rules. Let us note that previous proof schemes of causal consistency (e.g., [35]) did not include such a result, directly applying the square lemma instead. In our case, this would not be correct.

Lemma 3.16 (Switching lemma). *Given two composable transitions of the form* $t_1 = (s_1 \rightleftharpoons_{p_1,r_1,k_1} s_2)$ *and* $t_2 = (s_2 \rightleftharpoons_{p_2,r_2,k_2} s_3)$ *such that* $\overline{t_1}$ *and* t_2 *are concurrent, there exist a system* s_4 *and two composable transitions* $t_1' = (s_1 \rightleftharpoons_{p_2,r_2,k_2} s_4)$ *and* $t_2' = (s_4 \rightleftharpoons_{p_1,r_1,k_1} s_3)$.

Proof. First, using the loop lemma (Lemma 3.11), we have $\overline{t_1} = (s_2 \rightleftharpoons_{p_1,\overline{r_1},k_1} s_1)$. Now, since $\overline{t_1}$ and t_2 are concurrent, by applying the square lemma (Lemma 3.13) to $\overline{t_1} = (s_2 \rightleftharpoons_{p_1,\overline{r_1},k_1} s_1)$ and $t_2 = (s_2 \rightleftharpoons_{p_2,r_2,k_2} s_3)$, there exists a system s_4 such that $\overline{t_1'} = \overline{t_1}/t_2 = (s_3 \rightleftharpoons_{p_1,\overline{r_1},k_1} s_4)$ and $t_2' = t_2/\overline{t_1} = (s_1 \rightleftharpoons_{p_2,r_2,k_2} s_4)$. Using the loop lemma (Lemma 3.11) again, we have $t_1' = t_1/t_2 = (s_4 \rightleftharpoons_{p_1,r_1,k_1} s_3)$, which concludes the proof. □

Corollary 3.17. *Given two composable transitions* $t_1 = (s_1 \rightharpoonup_{p_1,r_1,k_1} s_2)$ *and* $t_2 = (s_2 \rightharpoonup_{p_2,r_2,k_2} s_3)$, *there exist a system* s_4 *and two composable transitions* $t_1' = (s_1 \rightharpoondown_{p_2,r_2,k_2} s_4)$ *and* $t_2' = (s_4 \rightharpoonup_{p_1,r_1,k_1} s_3)$. *Graphically,*

$$\begin{array}{ccc} s_1 & \xrightarrow{p_1,r_1,k_1} & s_2 \\ & & \downarrow{\scriptstyle p_2,r_2,k_2} \\ & & s_3 \end{array} \quad \Longrightarrow \quad \begin{array}{ccc} s_1 & \xrightarrow{p_1,r_1,k_1} & s_2 \\ \downarrow{\scriptstyle p_2,r_2,k_2} & & \downarrow{\scriptstyle p_2,r_2,k_2} \\ s_4 & \xrightarrow[p_1,r_1,k_1]{} & s_3 \end{array}$$

Proof. The corollary follows by applying the switching lemma (Lemma 3.16), noticing that two backward transitions are always concurrent. □

We now formally define the notion of causal equivalence between derivations, in symbols $\approx$, as the least equivalence relation between transitions closed under composition that obeys the following rules:

$$t_1; t_2/t_1 \approx t_2; t_1/t_2 \qquad t; \overline{t} \approx \epsilon_{\mathsf{init}(t)}$$

Causal equivalence amounts to say that those derivations that only differ for swaps of concurrent actions or the removal of successive inverse actions are equivalent. Observe that any of the notations $t_1; t_2/t_1$ and $t_2; t_1/t_2$ requires t_1 and t_2 to be concurrent.

Lemma 3.18 (Rearranging lemma). *Given systems* s, s', *if* $d = (s \rightleftharpoons^* s')$, *then there exists a system* s'' *such that* $d' = (s \rightharpoondown^* s'' \rightharpoonup^* s')$ *and* $d \approx d'$. *Furthermore,* d' *is not longer than* d.

Proof. The proof is by lexicographic induction on the length of d and on the number of steps from the earliest pair of transitions in d of the form $s_1 \rightharpoonup s_2 \rightharpoondown s_3$ to s'.

If there is no such pair we are done. If $s_1 = s_3$, then $s_1 \rightharpoonup s_2 = \overline{(s_2 \leftharpoonup s_3)}$. Indeed, if $s_1 \rightharpoonup s_2$ adds an item to the history of some process then $s_2 \leftharpoonup s_3$ should remove the same item. Otherwise, $s_1 \rightharpoonup s_2$ is an application of rule $Sched$ and $s_2 \leftharpoonup s_3$ should undo the scheduling of the same message. Then, we can remove these two transitions and the claim follows by induction since the resulting derivation is shorter and $(s_1 \rightharpoonup s_2 \leftharpoonup s_3) \approx \epsilon_{s_1}$. Otherwise, we apply Corollary 3.17 commuting $s_2 \leftharpoonup s_3$ with all forward transitions preceding it in d. If one such transition is its inverse, then we reason as above. Otherwise, we obtain a new derivation $d' \approx d$ which has the same length of d, and where the distance between the earliest pair of transitions in d' of the form $s'_1 \rightharpoonup s'_2 \leftharpoonup s'_3$ and s' has decreased. The claim follows then by the inductive hypothesis. □

An interesting consequence of the rearranging lemma is the following result, which states that every system obtained by both forward and backward steps from an initial system, is also reachable by a forward-only derivation:

Corollary 3.19. *Let s be an initial system. For each derivation $s \rightleftharpoons^* s'$, there exists a forward derivation of the form $s \rightharpoonup^* s'$.*

The following auxiliary result is also needed for proving causal consistency.

Lemma 3.20 (Shortening lemma)**.** *Let d_1 and d_2 be coinitial and cofinal derivations, such that d_2 is a forward derivation while d_1 contains at least one backward transition. Then, there exists a forward derivation d'_1 of length strictly less than that of d_1 such that $d'_1 \approx d_1$.*

Proof. We prove this lemma by induction on the length of d_1. By the rearranging lemma (Lemma 3.18) there exist a backward derivation d and a forward derivation d' such that $d_1 \approx d; d'$. Furthermore, $d; d'$ is not longer than d_1. Let $s_1 \leftharpoonup_{p_1,\overline{r_1},k_1} s_2 \rightharpoonup_{p_2,r_2,k_2} s_3$ be the only two successive transitions in $d; d'$ with opposite direction. We will show below that there is in d' a transition t which is the inverse of $s_1 \leftharpoonup_{p_1,\overline{r_1},k_1} s_2$. Moreover, we can swap t with all the transitions between t and $s_1 \leftharpoonup_{p_1,\overline{r_1},k_1} s_2$, in order to obtain a derivation in which $s_1 \leftharpoonup_{p_1,\overline{r_1},k_1} s_2$ and t are adjacent.[7] To do so we use the switching lemma (Lemma 3.16), since for all transitions t' in between, we have that $\overline{t'}$ and t are concurrent (this is proved below too). When $s_1 \leftharpoonup_{p_1,\overline{r_1},k_1} s_2$ and t are adjacent we can remove both of them using $\approx$. The

[7] More precisely, the transition is not t, but a transition that applies the same rule to the same process and producing the same history item, but possibly applied to a different system.

resulting derivation is strictly shorter, thus the claim follows by the inductive hypothesis.

Let us now prove the results used above. Thanks to the loop lemma (Lemma 3.11) we have the derivations above iff we have two forward derivations which are coinitial (with s_2 as initial state) and cofinal: $\overline{d}; d_2$ and d'. We first consider the case where $\overline{r_1} \neq \overline{Sched}$. Since the first transition of $\overline{d}; d_2$, $\overline{(s_1 \leftharpoondown_{p_1,\overline{r_1},k_1} s_2)}$, adds item k_1 to the history of p_1 and such an item is never removed (since the derivation is forward), then the same item k_1 has to be added also by a transition in d', otherwise the two derivations cannot be cofinal. The earliest transition in d' adding item k_1 is exactly t.

Let us now justify that for each transition t' before t in d' we have that $\overline{t'}$ and t are concurrent. First, t' is a forward transition and it should be applied to a process which is different from p_1, otherwise the item k_1 would be added by transition t in the wrong position in the history of p_1. We consider the following cases:

- If t' applies rule $Spawn$ to create a process p, then t should not apply to process p since the process p_1 to which t applies already existed before t'. Therefore, $\overline{t'}$ and t are concurrent.
- If t' applies rule $Send$ to send a message to some process p, then t cannot deliver the same message since we know that t is not a $Sched$ since it adds item k_1 to the history. Thus $\overline{t'}$ and t are concurrent.
- If t' applies some other rule, then t' and t are clearly concurrent.

Now, we consider the case $\overline{r_1} = \overline{Sched}$ with $k_1 = \mathsf{sched}(\{v, \lambda\})$, so that $\overline{(s_1 \leftharpoondown_{p_1,\overline{Sched},k_1} s_2)}$ adds a message $\{v, \lambda\}$ to the queue of p_1. We now distinguish two cases according to whether there is in $\overline{d}; d_2$ an application of rule $Receive$ to p_1 or not:

- If the forward derivation $\overline{d}; d_2$ contains no application of rule $Receive$ to p_1 then, in the final state, the queue of process p_1 contains the message. Hence, d' needs to contain a $Sched$ for the same message. The earliest such $Sched$ transition in d' is exactly t.

  Let us now justify that for each transition t' before t in d' we have that $\overline{t'}$ and t are concurrent. Consider the case where t' applies rule $Sched$ to deliver a different message to the same process p_1. Since no $Receive$ would be performed on p_1 then the queue will stay different, and the two derivations could not be cofinal, hence this case can never happen. In all the other cases the two transitions are concurrent.

- If the forward derivation $\overline{d}; d_2$ contains at least an application of rule $Receive$ to p_1, let us consider the first such application. This creates a history item k_2. In order for the two derivations to be cofinal, the same history item needs to be created in d'. The queue stored in k_2 has a suffix $\{v, \lambda\} : q$, hence also in d' the first $Sched$ delivering a message to p_1 should deliver message $\{v, \lambda\}$. Since there are no other $Sched$ nor $Receive$ targeting p_1 then the $Sched$ delivering message $\{v, \lambda\}$ to p_1 is concurrent to all previous transitions as desired.

□

Finally, we can state and prove the causal consistency of our reversible semantics. Intuitively speaking, it states that two different derivations starting from the same initial state can reach the same final state if and only if they are causal consistent. On the one hand, it means that derivations which are causal consistent lead to the same final state, hence it is not possible to distinguish such derivations looking at their final states (as a consequence, also their possible evolutions coincide). In particular, swapping two concurrent transitions or doing and undoing a given transition has no impact on the final state. On the other hand, derivations differing in any other way are distinguishable by looking at their final state, e.g., the final state keeps track of any past nondeterministic choice. In other terms, causal consistency states that the amount of history information stored is precisely what is needed to distinguish computations which are not causal consistent, and no more.

Theorem 3.21 (Causal consistency)**.** *Let d_1 and d_2 be coinitial derivations. Then, $d_1 \approx d_2$ iff d_1 and d_2 are cofinal.*

Proof. By definition of $\approx$, if $d_1 \approx d_2$, then they are coinitial and cofinal, so this direction of the theorem is verified.

Now, we have to prove that, if d_1 and d_2 are coinitial and cofinal, then $d_1 \approx d_2$. By the rearranging lemma (Lemma 3.18), we know that the two derivations can be written as the composition of a backward derivation, followed by a forward derivation, so we assume that d_1 and d_2 have this form. The claim is proved by lexicographic induction on the sum of the lengths of d_1 and d_2, and on the distance between the end of d_1 and the earliest pair of transitions t_1 in d_1 and t_2 in d_2 which are not equal. If all such transitions are equal, we are done. Otherwise, we have to consider three cases depending on the directions of the two transitions:

1. Consider that t_1 is a forward transition and t_2 is a backward one. Let us assume that $d_1 = d; t_1; d'$ and $d_2 = d; t_2; d''$. Here, we know that $t_1; d'$ is a forward derivation, so we can apply the shortening lemma (Lemma 3.20) to

the derivations $t_1; d'$ and $t_2; d''$ (since d_1 and d_2 are coinitial and cofinal, so are $t_1; d'$ and $t_2; d''$), and we have that $t_2; d''$ has a strictly shorter forward derivation which is causally equivalent, and so the same is true for d_2. The claim then follows by induction.

2. Consider now that both t_1 and t_2 are forward transitions. By assumption, the two transitions must be different. Let us assume first that they are not concurrent. Therefore, they should be applied to the same process and either both rules are $Sched$, or one is $Sched$ and the other one is $Receive$. In the first case, we get a contradiction to the fact that d_1 and d_2 are cofinal since both derivations are forward and, thus, we would either have a different queue in the process or different items $\mathsf{rec}(\ldots)$ in the history. In the second case, where we have one rule $Sched$ and one $Receive$, the situation is similar. Therefore, we can assume that t_1 and t_2 are concurrent transitions.

   We have two cases, according to whether t_1 is an application of $Sched$ or not. If it is not, let t_1' be the transition in d_2 creating the same history item as t_1. Then, we have to prove that t_1' can be switched back with all previous forward transitions. This holds since no previous forward transition can add any history item to the same process, since otherwise the two derivations could not be cofinal. Hence the previous forward transitions are applied to different processes and thus we never have a conflict since the only possible sources of conflict would be rules $Spawn$ and $Sched$, but this could not happen since, in this case, t_1 could not happen neither.

   If t_1 is an application of $Sched$ then we can find the transition t_1' in d_2 scheduling the same message (otherwise the two derivations could not be cofinal), and show that it can be switched with all the previous transitions. If the previous transition targets a different process then the only possible conflicts are with rules $Send$ or $Spawn$, but in this case t_1 could not have been performed. If the previous transition targets the same process then the only possible conflicts are with rules $Sched$ or $Receive$, but in this case the derivations could not be cofinal.

   Then, in all the cases, we can repeatedly apply the switching lemma (Lemma 3.16) to have a derivation causally equivalent to d_2 where t_2 and t_1' are consecutive. The same reasoning can be applied in d_1, so we end up with consecutive transitions t_1 and t_2'. Finally, we can apply the switching lemma once more to $t_1; t_2'$ so that the first pair of different transitions is now closer to the end of the derivation. Hence the claim follows by the inductive hypothesis.

3. Finally, consider that both t_1 and t_2 are backward transitions. By definition, we have that t_1 and t_2 are concurrent. Let us consider first that the rules applied in the transitions are different from $\overline{Sched}$. Then, we have that t_1 and t_2 cannot remove the same history item. Let k_1 be the history item removed by t_1. Since d_1 and d_2 are cofinal, either there is another transition in d_1 that puts k_1 back in the history or there is a transition t_1' in d_2 removing the same history item k_1. In the first case, $\overline{t_1}$ should be concurrent to all the backward transitions following it but the ones that remove history items from the history of the same process. All the transitions of this kind have to be undone by corresponding forward transitions (since they are not possible in d_2). Consider the last such transition: we can use the switching lemma (Lemma 3.16) to make it the last backward transition. Similarly, the forward transition undoing it should be concurrent to all the previous forward transitions (the reason is the same as in the previous case). Thus, we can use the switching lemma again to make it the first forward transition. Finally, we can apply the simplification rule $t;\overline{t} \approx \epsilon_{\mathsf{init}(t)}$ to remove the two transitions, thus shortening the derivation. In the second case (there is a transition t_1' in d_2 removing the same history item k_1), one can argue as in case (2) above. The claim then follows by the inductive hypothesis.

   The case when at least one of the rules applied in the transitions is $\overline{Sched}$ follows by a similar reasoning by considering the respective queues instead of the histories.

□

We now show that, as a corollary of previous results, a transition can be undone if and only if each of its consequences, if any, has been undone. Formally, a *consequence* of a forward transition t is a forward transition t' that can only happen after t has been performed (assuming t has not been undone in between). Hence t' cannot be switched with t. E.g., consuming a message from the queue of a process (using rule $Receive$) is a consequence of delivering this message (using rule $Sched$). Similarly, every action performed by a process is a consequence of spawning this process.

Corollary 3.22. *Let $d = (s_1 \rightleftharpoons \cdots \rightleftharpoons s_n \rightharpoonup s_{n+1} \rightleftharpoons \cdots \rightleftharpoons s_m)$ be a derivation, with $t = (s_n \rightharpoonup_{p,r,k} s_{n+1})$ a forward transition. Then, transition $\overline{t}$ can be applied to s_m, i.e., $s_m \leftharpoondown_{p,\overline{r},k} s_{m+1}$ iff each consequence of t in d, if any, has been undone in d.*

Proof. If each consequence t' of t in d has been undone in d then we can find $d' \approx d$ with no consequence of t, by moving each consequence t' and its undoing $\overline{t'}$ close

to each other (they can be switched using the switching lemma (Lemma 3.16) with all the transitions in between, but for further consequences which can be removed beforehand) and then applying $t'; \overline{t'} \approx \epsilon_{\mathsf{init}(t')}$. Then we can find $d'' \approx d'$ where t is the last transition, since t is concurrent to all subsequent transitions, hence we can apply the switching lemma (Lemma 3.16) again. The thesis then follows by applying the loop lemma (Lemma 3.11).

Assume now that transition $\bar{t}$ can be applied to s_m. Thanks to the rearranging lemma (Lemma 3.18) there is a derivation $d_b; d_f \approx d; \bar{t}$ where d_b is a backward derivation and d_f is a forward derivation. In order to transform $d; \bar{t}$ into $d_b; d_f$ we need to move $\bar{t}$ backward using the switching lemma (Lemma 3.16) until we find t. However, neither t nor $\bar{t}$ can be switched with the consequences of t, hence the only possibility is that all the consequences t' of t can be removed using $t'; \overline{t'} \approx \epsilon_{\mathsf{init}(t')}$ as above. □

3.5 Rollback Semantics

In this section, we introduce a (nondeterministic) "undo" operation which has some similarities to, e.g., the rollback operator of [81, 51]. Here, processes in "rollback" mode are annotated using $\lfloor\ \rfloor_\Psi$, where Ψ is the set of requested rollbacks. A typical rollback refers to a checkpoint that the backward computation of the process has to go through before resuming its forward computation. To be precise, we distinguish the following types of rollbacks:

- $\#^{\mathtt{t}}_{\mathsf{ch}}$, where "ch" stands for "checkpoint": a rollback to undo the actions of a process until a checkpoint with identifier $\mathtt{t}$ is reached;
- $\#_{\mathsf{sp}}$, where "sp" stands for "spawn": a rollback to undo *all* the actions of a process, finally deleting it from the system;
- $\#^{\lambda}_{\mathsf{sch}}$, where "sch" stands for "sched": a rollback to undo the actions of a process until the delivery of a message $\{v, \lambda\}$ is undone.

In the following, in order to simplify the reduction rules, we consider that our semantics satisfies the following *structural equivalence*:

$$(SC) \quad \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_\varnothing \mid \Pi \equiv \Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi$$

Note that only the first of the rollback types above targets a checkpoint. This kind of checkpoint is introduced nondeterministically by the rule below, where we denote by $\rightsquigarrow$ the new reduction relation that models backward moves of the rollback

$$(\overline{Seq}) \quad \Gamma; \lfloor\langle p, \tau(\theta, e) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi \rightsquigarrow \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi$$

$$(\overline{Check}) \quad \Gamma; \lfloor\langle p, \mathsf{check}(\theta, e, \mathtt{t}) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi \rightsquigarrow \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_{\Psi \setminus \{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \mid \Pi$$

$$(\overline{Send1}) \quad \Gamma \cup \{(p', \{v, \lambda\})\}; \lfloor\langle p, \mathsf{send}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi \rightsquigarrow \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi$$

$$(\overline{Send2}) \quad \begin{array}{c} \Gamma; \lfloor\langle p, \mathsf{send}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \lfloor\langle p', h', (\theta'', e''), q'\rangle\rfloor_{\Psi'} \mid \Pi \\ \rightsquigarrow \Gamma; \lfloor\langle p, \mathsf{send}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \lfloor\langle p', h', (\theta'', e''), q'\rangle\rfloor_{\Psi' \cup \{\#^{\lambda}_{\mathsf{sch}}\}} \mid \Pi \\ \text{if } (p', \{v, \lambda\}) \text{ does not occur in } \Gamma \text{ and } \#^{\lambda}_{\mathsf{sch}} \notin \Psi' \end{array}$$

$$(\overline{Receive}) \quad \Gamma; \lfloor\langle p, \mathsf{rec}(\theta, e, \{v, \lambda\}, q) : h, (\theta', e'), q \setminus\!\setminus \{v, \lambda\}\rangle\rfloor_\Psi \mid \Pi \rightsquigarrow \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi$$

$$(\overline{Spawn1}) \quad \begin{array}{c} \Gamma; \lfloor\langle p, \mathsf{spawn}(\theta, e, p'') : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \lfloor\langle [], p'', (\theta'', e''), []\rangle\rfloor_{\Psi'} \mid \Pi \\ \rightsquigarrow \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi \end{array}$$

$$(\overline{Spawn2}) \quad \begin{array}{c} \Gamma; \lfloor\langle p, \mathsf{spawn}(\theta, e, p'') : h, (\theta, e), q\rangle\rfloor_\Psi \mid \lfloor\langle p'', h'', (\theta'', e''), q''\rangle\rfloor_{\Psi'} \mid \Pi \\ \rightsquigarrow \Gamma; \lfloor\langle p, \mathsf{spawn}(\theta, e, p'') : h, (\theta, e), q\rangle\rfloor_\Psi \mid \lfloor\langle p'', h'', (\theta'', e''), q''\rangle\rfloor_{\Psi' \cup \{\#_{\mathsf{sp}}\}} \mid \Pi \\ \text{if } h'' \neq [] \vee q'' \neq [] \text{ and } \#_{\mathsf{sp}} \notin \Psi' \end{array}$$

$$(\overline{Self}) \quad \Gamma; \lfloor\langle p, \mathsf{self}(\theta, e) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi \rightsquigarrow \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi$$

$$(\overline{Sched}) \quad \begin{array}{c} \Gamma; \lfloor\langle p, h, (\theta, e), \{v, \lambda\} : q\rangle\rfloor_\Psi \mid \Pi \rightsquigarrow \Gamma \cup (p, \{v, \lambda\}); \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_{\Psi \setminus \{\#^{\lambda}_{\mathsf{sch}}\}} \mid \Pi \\ \text{if the topmost } \mathsf{rec}(\ldots) \text{ item in } h \text{ (if any) has the} \\ \text{form } \mathsf{rec}(\theta', e', \{v', \lambda'\}, q') \text{ with } q' \setminus\!\setminus \{v', \lambda'\} \neq \{v, \lambda\} : q \end{array}$$

Figure 3.15: Rollback semantics: backward reduction rules

semantics:

$$(\overline{Undo}) \quad \begin{array}{l} \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi \rightsquigarrow \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_{\Psi \cup \{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \mid \Pi \\ \text{if } \mathsf{check}(\theta', e', \mathtt{t}) \text{ occurs in } h, \text{ for some } \theta' \text{ and } e' \end{array}$$

Only after this rule is applied steps can be undone, since default computation in the rollback semantics is forward.

The backward rules of the rollback semantics are shown in Figure 3.15. Here, we assume that $\Psi \neq \varnothing$ (but Ψ' might be empty).

Note that, while rollbacks to checkpoints are generated nondeterministically by rule $\overline{Undo}$, the two other kinds of checkpoints are generated by the backward reduction rules in order to ensure causal consistency (in the sense of Corollary 3.22). This is clarified by the discussion below, where we briefly explain the main differences w.r.t. the uncontrolled backward semantics:

- As in the uncontrolled semantics of Figure 3.11, the sending of a message can be undone when the message is still in the global mailbox (rule $\overline{Send1}$).

Otherwise, one may need to first apply rule $\overline{Send2}$ in order to "propagate" the rollback mode to the receiver of the message, so that rules $\overline{Sched}$ and $\overline{Send1}$ can be eventually applied.

- As for undoing the spawning of a process p'', rule $\overline{Spawn1}$ steadily applies when both the history and the queue of the spawned process p'' are empty, thus deleting both the history item in p and the process p''. Otherwise, we apply rule $\overline{Spawn2}$ to propagate the rollback mode to process p'' so that, eventually, rule $\overline{Spawn1}$ can be applied.
- Finally, observe that rule $\overline{Sched}$ requires the same side condition as in the uncontrolled semantics. This is needed in order to avoid the commutation of rules $\overline{Receive}$ and $\overline{Sched}$.

The rollback semantics is modeled by the relation $\rightleftarrows$, which is defined as the union of the forward reversible relation $\rightharpoonup$ (Figure 3.10) and the backward relation $\rightharpoondown$ defined in Figure 3.15. Note that, in contrast to the (uncontrolled) reversible semantics of Section 3.4, the rollback semantics given by the relation $\rightleftarrows$ has less nondeterministic choices: all computations run forward except when a rollback action demands some backward steps to recover a previous state of a process (which can be propagated to other processes in order to undo the spawning of a process or the sending of a message).

Note, however, that besides the introduction of rollbacks, there is still some nondeterminism in the backward rules of the rollback semantics: on the one hand, the selection of the process when there are several ongoing rollbacks is nondeterministic; also, in many cases, both rule $\overline{Sched}$ and another rule are applicable to the same process. The semantics could be made deterministic by using a particular strategy to select the processes (e.g., round robin) and applying rule $\overline{Sched}$ whenever possible (i.e., give to $\overline{Sched}$ a higher priority than to the remaining backward rules).

Example 3.23. Consider again the program shown in Figure 3.12. Let us assume that function $\mathrm{main}/0$ is now defined as follows:

$$\begin{array}{l}\mathrm{main}/0 = \mathsf{fun}\ () \to \mathsf{let}\ S = \mathsf{spawn}(\mathrm{server}/0, [\,])\\ \qquad\qquad\qquad\quad \mathsf{in\ let}\ \_ = \mathsf{spawn}(\mathrm{client}/1, [S])\\ \qquad\qquad\qquad\quad \mathsf{in\ let}\ X = \mathsf{check}(\mathtt{t})\\ \qquad\qquad\qquad\quad \mathsf{in\ apply}\ \mathrm{client}/1\ (S)\end{array}$$

so that a checkpoint has been introduced after spawning the two processes: the server (s) and one of the clients ($\mathrm{c2}$). Then, by repeating the same forward derivation

shown in Figure 3.13 (with the additional step to evaluate the checkpoint), we get the following final system:

$$\begin{array}{l}\{\ \};\ \langle \mathrm{c1}, [\mathsf{rec}(\_,\_,m_4,[m_4]), \mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{check}(\_,\_,\mathtt{t}), \mathsf{spawn}(\_,\_,\mathrm{c2}),\\ \qquad \mathsf{spawn}(\_,\_,\mathrm{s})], (\_,\mathrm{ok}), [\,]\rangle\\ \quad \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2),\\ \qquad \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle\\ \quad \mid \langle \mathrm{c2}, [\underline{\mathsf{rec}(\_,\_,m_2,[m_2])}, \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_,\mathrm{ok}), [\,]\rangle\end{array}$$

Figure 3.16 shows the steps performed by the rollback semantics in order to undo the steps of process c1 until the checkpoint is reached. In Figure 3.16 we follow the same conventions as in Examples 3.7 and 3.9. Observe that we could also use the relation "$\not\hookrightarrow$" here in order to also perform some forward steps on process c2, as it would happen in practice.

We state below the soundness of the rollback semantics. In order to do it, we let $\mathsf{rolldel}(s)$ denote the system that results from s by removing ongoing rollbacks; formally, $\mathsf{rolldel}(\Gamma;\Pi) = \Gamma;\mathsf{rolldel}'(\Pi)$, with

$$\begin{array}{lcl}\mathsf{rolldel}'(\langle p,h,(\theta,e),q\rangle) & = & \langle p,h,(\theta,e),q\rangle\\ \mathsf{rolldel}'(\lfloor\langle p,h,(\theta,e),q\rangle\rfloor_\Psi) & = & \langle p,h,(\theta,e),q\rangle\\ \mathsf{rolldel}'(\langle p,h,(\theta,e),q\rangle \mid \Pi) & = & \langle p,h,(\theta,e),q\rangle \mid \mathsf{rolldel}'(\Pi)\\ \mathsf{rolldel}'(\lfloor\langle p,h,(\theta,e),q\rangle\rfloor_\Psi \mid \Pi) & = & \langle p,h,(\theta,e),q\rangle \mid \mathsf{rolldel}'(\Pi)\end{array}$$

where we assume that Π is not empty. We also extend the definition of initial and reachable systems to the rollback semantics.

Definition 3.24 (Reachable systems under the rollback semantics)**.**
A system is *initial* under the rollback semantics if it is composed by a single process with an empty set Ψ of active rollbacks; furthermore, the history, the queue and the global mailbox are empty too. A system s is *reachable* under the rollback semantics if there exist an initial system s_0 and a derivation $s_0 \not\hookrightarrow^* s$ using the rules corresponding to a given program.

Theorem 3.25 (Soundness)**.** *Let s be a system reachable under the rollback semantics. If $s \not\hookrightarrow^* s'$, then* $\mathsf{rolldel}(s) \rightleftharpoons^* \mathsf{rolldel}(s')$.

Proof. For forward transitions the proof is trivial since the forward rules are the same in both semantics, and they apply only to processes which are not under rollback. For backward transitions the proof is by case analysis on the applied rule, noting that the effect of structural equivalence is removed by rolldel:

$$
\begin{array}{lll}
 & \{\,\}; & \lfloor\langle \mathrm{c1}, [\underline{\mathsf{rec}(\_,\_,m_4,[m_4])}, \mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{check}(\_,\_,\mathtt{t}), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], \\
 & & (\_, \mathrm{ok}), [\,]\rangle\rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{Receive}} & \{\,\}; & \lfloor\langle \mathrm{c1}, [\underline{\mathsf{send}(\_,\_,\mathrm{s},m_3)}, \mathsf{check}(\_,\_,\mathtt{t}), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], \\
 & & (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [m_4]\rangle\rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{Send2}} & \{\,\}; & \lfloor\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{check}(\_,\_,\mathtt{t}), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], \\
 & & (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [m_4]\rangle\rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \\
 & & \mid \lfloor\langle \mathrm{s}, [\underline{\mathsf{send}(\_,\_,\mathrm{c1},m_4)}, \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle\rfloor_{\{\#^{3}_{\mathsf{sch}}\}} \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{Send2}} & \{\,\}; & \lfloor\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{check}(\_,\_,\mathtt{t}), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], \\
 & & (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\underline{m_4}]\rangle\rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}},\#^{4}_{\mathsf{sch}}\}} \\
 & & \mid \lfloor\langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c1},m_4), \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle\rfloor_{\{\#^{3}_{\mathsf{sch}}\}} \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{Sched}} & \{(\mathrm{c1},m_4)\}; & \lfloor\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{check}(\_,\_,\mathtt{t}), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], \\
 & & (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle\rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \\
 & & \mid \lfloor\langle \mathrm{s}, [\underline{\mathsf{send}(\_,\_,\mathrm{c1},m_4)}, \mathsf{rec}(\_,\_,m_3,[m_3]), \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], \\
 & & (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle\rfloor_{\{\#^{3}_{\mathsf{sch}}\}} \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{Send1}} & \{\,\}; & \lfloor\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{check}(\_,\_,\mathtt{t}), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], \\
 & & (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle\rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \\
 & & \mid \lfloor\langle \mathrm{s}, [\underline{\mathsf{rec}(\_,\_,m_3,[m_3])}, \mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathrm{c1}\ !\ \mathrm{ack}]), [\,]\rangle\rfloor_{\{\#^{3}_{\mathsf{sch}}\}} \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{Receive}} & \{\,\}; & \lfloor\langle \mathrm{c1}, [\mathsf{send}(\_,\_,\mathrm{s},m_3), \mathsf{check}(\_,\_,\mathtt{t}), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], \\
 & & (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle\rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \\
 & & \mid \lfloor\langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\underline{m_3}]\rangle\rfloor_{\{\#^{3}_{\mathsf{sch}}\}} \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{Sched}} & \{(\mathrm{s},m_3)\}; & \lfloor\langle \mathrm{c1}, [\underline{\mathsf{send}(\_,\_,\mathrm{s},m_3)}, \mathsf{check}(\_,\_,\mathtt{t}), \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], \\
 & & (\_, C[\mathsf{receive}\ \mathrm{ack} \to \mathrm{ok}]), [\,]\rangle\rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{Send1}} & \{\,\}; & \lfloor\langle \mathrm{c1}, [\underline{\mathsf{check}(\_,\_,\mathtt{t})}, \mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathrm{s}\ !\ \{\mathrm{c1}, \mathrm{req}\}]), [\,]\rangle\rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, \mathrm{ok}), [\,]\rangle \\
\rightsquigarrow_{\overline{Check}} & \{\,\}; & \langle \mathrm{c1}, [\mathsf{spawn}(\_,\_,\mathrm{c2}), \mathsf{spawn}(\_,\_,\mathrm{s})], (\_, C[\mathsf{check}(\mathtt{t})]), [\,]\rangle \\
 & & \mid \langle \mathrm{s}, [\mathsf{send}(\_,\_,\mathrm{c2},m_2), \mathsf{rec}(\_,\_,m_1,[m_1])], (\_, C[\mathsf{receive}\ \{P,M\} \to \ldots]), [\,]\rangle \\
 & & \mid \langle \mathrm{c2}, [\mathsf{rec}(\_,\_,m_2,[m_2]), \mathsf{send}(\_,\_,\mathrm{s},m_1)], (\_, \mathrm{ok}), [\,]\rangle
\end{array}
$$

Figure 3.16: A derivation under the backward reduction rules of Figure 3.15, with $m_1 = \{\{\mathrm{c2}, \mathrm{req}\}, 1\}$, $m_2 = \{\mathrm{ack}, 2\}$, $m_3 = \{\{\mathrm{c1}, \mathrm{req}\}, 3\}$, and $m_4 = \{\mathrm{ack}, 4\}$.

- Rule $\overline{Undo}$: the effect is removed by rolldel, hence an application of this rule corresponds to a zero-step derivation under the uncontrolled semantics;
- Rules $\overline{Seq}$, $\overline{Check}$, $\overline{Send1}$, $\overline{Receive}$, $\overline{Spawn1}$, $\overline{Self}$ and $\overline{Sched}$: they are matched, respectively, by rules $\overline{Seq}$, $\overline{Check}$, $\overline{Send}$, $\overline{Receive}$, $\overline{Spawn}$, $\overline{Self}$ and $\overline{Sched}$ of the uncontrolled semantics;
- Rules $\overline{Send2}$ and $\overline{Spawn2}$: the effect is removed by rolldel, hence an application of any of these rules corresponds to a zero-step derivation under the uncontrolled semantics.

□

We can now show the completeness of the rollback semantics provided that the involved process is in rollback mode:

Lemma 3.26 (Completeness in rollback mode)**.** *Let s be a reachable system. If $s \rightharpoonup s'$ then take any system s_r such that* $\mathsf{rolldel}(s_r) = s$ *and where the process that performed the transition $s \rightharpoonup s'$ is in rollback mode for a non-empty set of rollbacks. There exists s'_r such that $s_r \rightharpoondown s'_r$ and* $\mathsf{rolldel}(s'_r) = s'$.

Proof. The proof is by case analysis on the applied rule. Each step is matched by the homonymous rule, but for $\overline{Send}$ and $\overline{Spawn}$ which are matched by rules $\overline{Send1}$ and $\overline{Spawn1}$. □

The following result illustrates the usefulness of the rollback semantics:

Lemma 3.27. *Let us consider a forward derivation d of the form:*

$$\begin{array}{l}\Gamma; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \mid \Pi \\ \rightharpoonup \Gamma; \langle p, \mathsf{check}(\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e, \mathtt{t}) : h, (\theta, \mathsf{let}\ X = \mathtt{t}\ \mathsf{in}\ e), q\rangle \mid \Pi \\ \rightharpoonup^* \Gamma'; \langle p, h', (\theta', e'), q'\rangle \mid \Pi'\end{array}$$

Then, there is a backward derivation d' under the rollback semantics restoring process p:

$$\begin{array}{l}\Gamma'; \lfloor \langle p, h', (\theta', e'), q'\rangle \rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \mid \Pi' \\ \leftharpoondown^* \Gamma''; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \mid \Pi''\end{array}$$

Proof. Trivially (by Theorem 3.25) the forward derivation d can also be performed under the uncontrolled reversible semantics. Now, by applying the loop lemma (Lemma 3.11) to each step of d, we have a backward derivation $\overline{d}$ of the form:

$$\begin{array}{l}\Gamma'; \langle p, h', (\theta', e'), q'\rangle \mid \Pi' \\ \leftharpoonup^* \Gamma; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \mid \Pi\end{array}$$

Consider the relation $\leq$ on transitions of $\overline{d}$ defined as the reflexive and transitive closure of the following clauses:

- $t_1 \leq t_2$ if both t_1 and t_2 undo actions in the same process p', and the transition undone by t_2 is a direct consequence of the one undone by t_1;
- $t_1 \leq t_2$ if t_1 undoes a spawn of process p_2 and t_2 undoes the first transition of p_2;
- $t_1 \leq t_2$ if t_1 undoes the send of a message λ and t_2 undoes the scheduling of the same message.

Let us show that $\leq$ is a partial order. We only need to show that there are no cycles, but this follows from the fact that the total order given by $\overline{d}$ is compatible with $\leq$.

We also notice that any two transitions which are not related by $\leq$ can be swapped using the switching lemma (Lemma 3.16).

Then, there exists a derivation $\overline{d_r}; \overline{d_u}$ such that $\overline{d_r}$ contains all transitions t such that $t_l \leq t$ where t_l is the last transition in $\overline{d}$, and only them. Since $\overline{d_u}$ contains no transition on p we have that $\overline{d_r}$ is of the form:

$$\Gamma'; \langle p, h', (\theta', e'), q' \rangle \mid \Pi' \\ \leftharpoonup^* \Gamma''; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q \rangle \mid \Pi''$$

Using again the switching lemma (Lemma 3.16) one can transform $\overline{d_r}$ into a derivation $\overline{d_r'}$ obtained using the following execution strategy, where initially the active process is p, the termination condition is "the checkpoint action `t` has been undone", and the stack is empty:

- transitions of the active process are undone if possible, until the termination condition holds; if there is an occurrence of the active process in the stack and the termination condition for this process is matched because of the current transition undo, remove such occurrence from the stack (this remove does not follow the usual FIFO strategy for stacks);
- if the termination condition holds, then pop a new active process from the stack, if there are no processes on the stack then terminate;
- if no transition is possible for the active process then one of the two following subconditions should hold:

1. the active process needs to undo a spawn of a process which is not in the initial state: push the active process on the stack, and set the spawned process as new active process with termination condition "all actions have been undone";
2. the active process needs to undo a send of a message λ which is not in the global mailbox: push the active process on the stack, and set the process to which message λ has been scheduled as new active process with termination condition "the scheduling of the message λ has been undone";

The switching lemma can be applied since this execution strategy is compatible with $\leq$. Now we show that the same execution strategy can be performed using the rollback semantics. We only need to show that the active process is in rollback mode, then the thesis will follow from the completeness in rollback mode (Lemma 3.26). This can be shown by inspection of the execution strategy, considering the following invariant: the active process and all the processes on the stack are in rollback mode, and they have one checkpoint for each occurrence in the stack, plus one for the occurrence as active process. The invariant holds at the beginning since p has one checkpoint corresponding to its termination condition. When the termination condition holds, a checkpoint is removed by rule $\overline{Check}$, $\overline{Spawn1}$, or $\overline{Sched}$. When a new active process is selected, a new checkpoint is added by rule $\overline{Spawn2}$ or $\overline{Send2}$. □

One can notice that in the lemma above only the process containing the checkpoint is restored. We can restore the whole system to the original configuration only if we restrict the forward derivation to be a causal derivation, following the terminology in [36].

Definition 3.28. A forward derivation d is causal iff all the transitions are consequences of the first one.

Hence, we have the following corollary:

Corollary 3.29. *Let us consider a causal derivation d of the form:*

$$\begin{aligned}&\Gamma; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \mid \Pi\\ &\rightharpoonup \Gamma; \langle p, \mathsf{check}(\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e, \mathtt{t}) : h, (\theta, \mathsf{let}\ X = \mathtt{t}\ \mathsf{in}\ e), q\rangle \mid \Pi\\ &\rightharpoonup^* \Gamma'; \langle p, h', (\theta', e'), q'\rangle \mid \Pi'\end{aligned}$$

Then, there is a backward derivation d' under the rollback semantics restoring the system to the original configuration:

$$\Gamma'; \lfloor \langle p, h', (\theta', e'), q' \rangle \rfloor_{\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \mid \Pi' \\ \rightharpoonup^* \Gamma; \langle p, h, (\theta, \mathsf{let}\ X = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q \rangle \mid \Pi$$

Proof. The proof follows the same strategy as the one of Lemma 3.27, noticing that $\overline{d_u}$ is empty hence $\Gamma = \Gamma''$ and $\Pi = \Pi''$. □

While a derivation restoring the whole system exists, not all derivations do so. More in general, given a set of rollbacks, it is not the case that there is a unique system that is obtained by executing backward transitions as far as possible (without executing any $\overline{Undo}$). Indeed, the only nondeterminism is due to the fact that $\overline{Sched}$ can commute with other transitions, e.g., with $\overline{Check}$, which ends the rollback. If we establish a policy for $Sched$ actions, and we use the dual policy for undoing them, then the result is unique. A sample policy could be that $Sched$ steps are performed as late as possible, and dually undone as soon as possible. In such a setting we have the following result:

Lemma 3.30. *Let s be a reachable system. If $s \rightharpoonup s_1$ and $s \rightharpoonup s_2$, both transitions use the same policy for $\overline{Sched}$, and the rules are different from $\overline{Undo}$, then there exists a system s' such that $s_1 \rightharpoonup^* s'$ and $s_2 \rightharpoonup^* s'$.*

Proof. Let us consider the case where both transitions are applied to the same process p. In this case, only one backward rule is applicable and the claim follows trivially. Note that the only case where more than one backward rule would be applicable is when one of the rules is $\overline{Sched}$ and the other one is a different rule, but this case is excluded by the fact that we consider a fixed policy for $\overline{Sched}$ as mentioned above.

Consider now the case where each transition is applied to a different process, say p_1 and p_2, so that we have $s \rightharpoonup s_1$ and $s \rightharpoonup s_2$. By the soundness of the backward reduction rules of the rollback semantics (Theorem 3.25), we have $\mathsf{rolldel}(s) \leftharpoonup^* \mathsf{rolldel}(s_1)$ and $\mathsf{rolldel}(s) \leftharpoonup^* \mathsf{rolldel}(s_2)$. Note that each of the derivations above has either length 1 or 0. We just consider the case where they have both length 1, since the others are simpler. By the square lemma (Lemma 3.13), there exists a system s'' such that $\mathsf{rolldel}(s_1) \leftharpoonup s''$ and $\mathsf{rolldel}(s_2) \leftharpoonup s''$. Now, we show that processes p_2 and p_1 are still in rollback mode in s_1 and s_2, respectively. Here, the only case where the application of a backward rule to a process removes a rollback from a different process is $\overline{Spawn}$. Consider, e.g., that the rule applied to process p_1 is $\overline{Spawn}$ and that the removed process is p_2. In this case, however, no backward rule

could be applied to process p_2, so this case is not possible. Therefore, by applying the completeness of the rollback semantics, we have $s_1 \rightharpoonup s_1'$ and $s_2 \rightharpoonup s_2'$ with $\mathsf{rollDel}(s_1') = \mathsf{rollDel}(s_2') = s''$. The thesis follows by noticing that the rollbacks in s_1' and s_2' coincide (in both the cases they are the rollbacks in s minus the ones removed by the performed transitions, which are the same in both the cases) hence $s_1' = s_2' = s'$. □

The following result is an easy corollary of the previous lemma:

Corollary 3.31. *Let s be a reachable system. If $s \rightharpoonup^* s_1 \not\rightharpoonup$ and $s \rightharpoonup^* s_2 \not\rightharpoonup$, both derivations use the same policy for $\overline{Sched}$, and never use rule $\overline{Undo}$, then $s_1 = s_2$.*

Proof. Analogously to the proof of Corollary 3.14, using standard results for confluence of abstract relations [9], we have that Lemma 3.30 implies that there exists a system s' such that $s_1 \rightharpoonup^* s'$ and $s_2 \rightharpoonup^* s'$. Moreover, since both s_1 and s_2 are irreducible, we have $s_1 = s_2$. □

3.6 Proof-of-concept Implementation of the Reversible Semantics

We have developed a proof-of-concept implementation of the uncontrolled reversible semantics for Erlang that we presented in Section 3.3. This implementation is conveniently bundled together with a graphical user interface (we refer to this as "the application") in order to facilitate the interaction of users with the reversible semantics. However, the application has been developed in a modular way, so that it is possible to include the implementation of the reversible semantics in other projects (e.g., it has been included in the reversible debugger CauDEr [85, 84]).

Let us recall that our semantics is defined for a language that is equivalent to Core Erlang [25], a much simpler language than Erlang. Not surprisingly, the implementation of our reversible semantics is defined for Core Erlang as well. Prior to its compilation, Erlang programs are translated to Core Erlang by the Erlang/OTP system, so that the resulting code is simplified. For instance, pattern matching can occur almost anywhere in an Erlang program, whereas in Core Erlang, pattern matching can only occur in case statements. Nevertheless, directly writing Core Erlang programs would not be comfortable for the user, since Core Erlang is only used as an intermediate language. Hence, our implementation considers the Core Erlang code translated from the Erlang program provided by the user.

The application works as follows: when it is started, the first step is to select an Erlang source file. The selected source file is then translated into Core Erlang, and

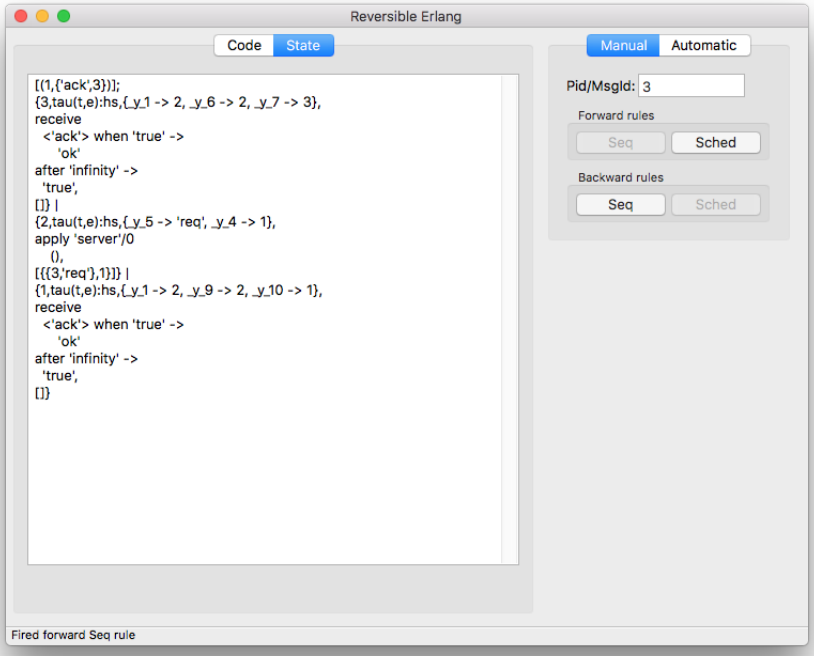


Figure 3.17: Screenshot of the application

the resulting code is shown in the code window. Then, the user can choose any of the functions from the module and write the arguments that she wants to evaluate the function with. An initial system state, with an empty global mailbox and a single process performing the specified function application, appears on the state window when the user presses the start button, as shown in Figure 3.17. Now, the user is able to control the system state by selecting the rules from the reversible semantics that she wants to fire.

We have defined two different modes for controlling the reversible semantics. The first mode is a *manual* mode, where the user selects the rule to be fired for a particular process or message. Here, the user is in charge of "controlling" the reversible semantics, although this approach can rapidly become exhausting. The second mode is the *automatic* mode. Here, the user specifies a number of steps and chooses a direction (forward or backward), and the rules to be applied are selected at random—for the chosen direction—until the specified number of steps is reached or

no more rules can be applied. Alternatively, the user can move the state forward up to a *normalized system*. To normalize a system, one must ignore the $Sched$ rule and apply only the other rules. A normalized system is reached when no rule other than $Sched$ can be fired. Hence, in a normalized system, either all processes are blocked (waiting for some message to arrive) or the system state is final. Normalizing a system allows the user to perform all the reductions that do not depend on the network. Reductions depending on the network can then be performed one by one to understand their impact on the derivation.

The release version (v1.0) of the application is fully written in Erlang, and it is publicly available from `https://github.com/mistupv/rev-erlang` under the MIT license. Hence, the only requirement to build the application is to have Erlang/OTP installed. Besides, we have included some documentation and a few examples to easily test the application.

3.7 Related Work

First, regarding the semantics of Erlang presented in Section 3.3, we have some similarities with both [19] and [130]. In contrast to [19], which presents a monolithic semantics, our relation is split into expression-level rules and system-level rules. This division eases the presentation of a reversible semantics, since it only affects the system-level rules. As for [130], we follow the idea of introducing a global mailbox (there called "ether") so that every message passing communication can be decomposed into two steps: sending and scheduling. Their semantics considers other features of Erlang (such as links or monitors) but does not present the semantics of expressions, as we do. Another difference lies in the fact that all *side effects* are asynchronous in [130] (e.g., the spawning of a process is asynchronous), a design decision that allows for a simpler semantics. In our case, spawning a process is dealt with in a synchronous manner, which is closer to the actual behavior of Erlang. Finally, as mentioned in Section 3.3, we deliberately ignore the restriction that guarantees the order of messages for any pair of given processes. This may increase the number of possible interleavings, but we consider that it models better the behavior of current Erlang implementations.

Regarding reversibility, the approach presented in this paper is in the line of work on causal-consistent reversibility [35, 117] (see [83] for a survey). In particular, our work is closer to [35], since we also consider adding a *memory* (a history in our terminology) in order to make a computation reversible. Moreover, our proof of causal consistency mostly follows the proof scheme in [35]. In contrast, we consider

a different concurrent language with asynchronous communication, while communication in [35] is synchronous. On the other hand, [117] does not introduce a memory but keeps the old actions marked with a "key". As pointed out in [117], process equivalence is easier to check than in [35] (where one would need to abstract away from the memories). Like [35], also [117] considers synchronous communication. Formalizing the Erlang semantics using a labeled transition relation as in [35, 117] (rather than a reduction semantics, as we do in this paper), and then defining a reversible extension would be an interesting and challenging approach for further research.

Nevertheless, as mentioned in the Introduction, the closest to our work is the debugging approach based on a rollback construct of [51, 52, 81, 82, 90], but it is defined in the context of a different language or formalism. Among the languages considered in the works above, the closest to ours is μOz [90, 51]. A main difference is that μOz is not distributed: messages move atomically from the sender to a chosen queue, and from the queue to the receiver. Each of the two actions is performed by a specific process, hence naturally part of its history. In our case, the scheduling action is not directly performed by a process, and it is only potentially observed when the target process performs the receive action (but not necessarily observed, e.g., if the message does not match the patterns in the receive). The definition of the notions of conflict and concurrency in this setting is, as a consequence, much trickier than in μOz. This difficulty carries over to the definition of the history information that needs to be tracked, and to how this information is exploited in the reversible semantics (actually, this was one of the main difficulties we encountered during our work). Furthermore, in the case of μOz only the uncontrolled semantics has been fully formalized [90], while the controlled semantics and the corresponding results are only sketched [51].

Also, we share some similarities with the checkpointing technique for fault-tolerant distributed computing of [43, 77], although the aim is different (they aim at defining a new language rather than extending an existing one).

On the other hand, [107] has very recently introduced a novel technique for recovery in Erlang based on session types. Although the approach is different, our rollback semantics could also be used for rollback recovery. In contrast to [107], that only considers recovery of processes as a whole, our approach could be helpful to design a more fine grained recovery strategy.

Finally, as mentioned in the Introduction, this paper extends and improves [109] in different ways. Firstly, [109] only presents a rollback semantics. Here, we have introduced an uncontrolled reversible semantics and have proved a number of fundamental theoretical properties, including its causal consistency (no proofs of tech-

nical results are provided in [109]). Secondly, the reversible semantics in [109] does not consider messages' unique identifiers (λ), so that the problems mentioned in Section 3.4 are not avoided. Moreover, the process' histories also include items for the applications of rule *Sched*, which makes the underlying notion of concurrency unnecessarily restrictive. As for the rollback semantics of [109], besides the points mentioned above, it only considered one rollback for each process, while sets of rollbacks are accepted in this work. Consequently, we have now reduced the number of rules required to undo the sending of a message or to undo the introduction of a checkpoint, so that the rollback semantics is simpler. Furthermore, we have designed and developed a proof-of-concept implementation in this paper that allowed us to check the viability of the reversible semantics in practice.

3.8 Conclusion and Future Work

We have defined a reversible semantics for a first-order subset of Erlang that undoes the actions of a process step by step in a sequential way. To the best of our knowledge, this is the first attempt to define a reversible semantics for Erlang. In this work, we have first introduced an uncontrolled, reversible semantics, and have proved that it enjoys the usual properties (loop lemma, square lemma, and causal consistency). Then, we have introduced a controlled version of the backward semantics that can be used to model a rollback operator that undoes the actions of a process up to a given checkpoint. A proof-of-concept implementation shows that our approach is indeed viable in practice.

As future work, we consider the definition of mechanisms to control reversibility so that history information is stored only when needed to perform a rollback. This could be essential to extend Erlang with a new construct for *safe sessions*, where all the actions in a session can be undone if the session aborts. Such a construct could have a great potential to automate the fault-tolerance capabilities of the language Erlang.

Chapter 4

CauDEr: A Causal-Consistent Reversible Debugger for Erlang

Ivan Lanese[1], Naoki Nishida[2], Adrián Palacios[3], Germán Vidal[3]

[1] Focus Team, University of Bologna/INRIA
ivan.lanese@gmail.com

[2] Graduate School of Informatics, Nagoya University
nishida@i.nagoya-u.ac.jp

[3] MiST, DSIC, Universitat Politècnica de València
{apalacios,gvidal}@dsic.upv.es

Abstract. Programming languages based on the actor model, such as Erlang, avoid some concurrency bugs by design. However, other concurrency bugs, such as message order violations and livelocks, can still show up in programs. These hard-to-find bugs can be more easily detected by using causal-consistent reversible debugging, a debugging technique that allows one to traverse a computation both forward and backward. Most notably, causal consistency implies that, when going backward, an action can only be undone provided that its consequences, if any, have been undone beforehand.

This work has been partially supported by MINECO/AEI/FEDER (EU) under grant TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic), by COST Action IC1405 on Reversible Computation - extending horizons of computing, and by JSPS KAKENHI Grant Number JP17H01722. Adrián Palacios was partially supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores* and *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D*, MINECO (SEIDI), under FPI grants BES-2014-069749 and EEBB-I-16-11469. This chapter is an adapted author version of the paper published in *"Ivan Lanese, Naoki Nishida, Adrián Palacios, Germán Vidal: CauDEr: A Causal-Consistent Reversible Debugger for Erlang. Proceedings of FLOPS 2018, Lecture Notes in Computer Science 10818: 247–263 (2018)"*. DOI: https://doi.org/10.1007/978-3-319-90686-7_16

To the best of our knowledge, we present the first causal-consistent reversible debugger for Erlang, which may help programmers to detect and fix various kinds of bugs, including message order violations and livelocks.

4.1 Introduction

Over the last years, concurrent programming has become a common practice. However, it is also a difficult and error-prone activity, since concurrency enables faulty behaviors, such as *deadlocks* and *livelocks*, which are hard to avoid, detect and fix. One of the reasons for these difficulties is that these behaviors may show up only in some extremely rare circumstances (e.g., for some unusual scheduling).

A recent analysis [92] reveals that most of the approaches to software validation and debugging in message-passing concurrent languages like Erlang are based on some form of static analysis (e.g., Dialyzer [91], McErlang [48], Soter [39]) or testing (e.g., QuickCheck [28], PropEr [113], Concuerror [61], CutEr [54]). However, these techniques are helpful only to find some specific categories of problems. On the other hand, traditional debuggers (like the one included in the OTP Erlang distribution) are sometimes not particularly useful when an unusual interleaving brings up an error, since recompiling the program for debugging may give rise to a completely different execution behavior. In this setting, *causal-consistent reversible debugging* [51] may be useful to complement the previous approaches. Here, one can run a program in the debugger in a controlled manner. If something (potentially) incorrect shows up, the user can stop the forward computation and go backwards—in a causal-consistent way—to look for the origin of the problem. In this context, we say that a backward step is *causal consistent* [35, 83] if an action cannot be undone until all the actions that depend on it have already been undone. Causal-consistent reversibility is particularly relevant for debugging because it allows us to undo the actions of a given process in a stepwise manner while ignoring the actions of the remaining processes, unless they are causally related. In a traditional reversible debugger, one can only go backwards in exactly the reverse order of the forward execution, which makes focusing on undoing the actions of a given process much more difficult, since they can be interleaved with completely unrelated actions from other processes.

The main contributions of this paper are the following. We have designed and implemented CauDEr, a publicly available software tool for causal-consistent reversible debugging of (a subset of) Erlang programs. The tool builds upon some recent developments on the causal-consistent reversible semantics of Erlang [109, 86],

$$
\begin{array}{rcl}
module & ::= & \mathsf{module}\ Atom = fun_1, \ldots, fun_n \\
fun & ::= & fname = \mathsf{fun}\ (X_1, \ldots, X_n) \to expr \\
fname & ::= & Atom/Integer \\
lit & ::= & Atom \mid Integer \mid Float \mid [\,] \\
expr & ::= & Var \mid lit \mid fname \mid [expr_1|expr_2] \mid \{expr_1, \ldots, expr_n\} \\
 & \mid & \mathsf{call}\ expr\ (expr_1, \ldots, expr_n) \mid \mathsf{apply}\ expr\ (expr_1, \ldots, expr_n) \\
 & \mid & \mathsf{case}\ expr\ \mathsf{of}\ clause_1; \ldots; clause_m\ \mathsf{end} \\
 & \mid & \mathsf{let}\ Var = expr_1\ \mathsf{in}\ expr_2 \mid \mathsf{receive}\ clause_1; \ldots; clause_n\ \mathsf{end} \\
 & \mid & \mathsf{spawn}(expr, [expr_1, \ldots, expr_n]) \mid expr_1\ !\ expr_2 \mid \mathsf{self}() \\
clause & ::= & pat\ \mathsf{when}\ expr_1 \to expr_2 \\
pat & ::= & Var \mid lit \mid [pat_1|pat_2] \mid \{pat_1, \ldots, pat_n\}
\end{array}
$$

Figure 4.1: Language syntax rules

though we also introduce (in Section 4.3) a new rollback semantics which is especially tailored for reversible debugging. In this semantics, one can for instance run a program backwards up to the sending of a particular message, the creation of a given process, or the introduction of a binding for some variable. We present our tool and illustrate its use for finding bugs that would be difficult to deal with using the previously available tools (Section 4.4). We use a concurrent implementation of the dining philosophers problem as a running example. CauDEr is publicly available from `https://github.com/mistupv/cauder`.

4.2 The Language

Erlang is a message passing concurrent and distributed functional programming language. We define our technique for (a subset of) Core Erlang [25], which is used as an intermediate representation during the compilation of Erlang programs. In this section, we describe the syntax and semantics of the subset of Core Erlang we are interested in.

The syntax of the language can be found in Figure 4.1. A module is a sequence of function definitions, where each function name f/n (atom/arity) has an associated definition of the form $\mathsf{fun}\ (X_1, \ldots, X_n) \to e$. We consider that a program consists of a single module for simplicity. The body of a function is an *expression*, which can include variables, literals, function names, lists, tuples, calls to built-in functions—mainly arithmetic and relational operators—, function applications, case expressions, let bindings, and receive expressions; furthermore, we also consider the

functions spawn, "!" (for sending a message), and self() that are usually considered built-ins in the Erlang language. As is common practice, we assume that X is a fresh variable in a let binding of the form let $X = expr_1$ in $expr_2$.

In this language, we distinguish expressions, patterns, and values. In contrast to expressions, *patterns* are built from variables, literals, lists, and tuples. Finally, *values* are built from literals, lists, and tuples, i.e., they are *ground* (without variables) patterns. Expressions are denoted by $e, e', e_1, e_2, \ldots$, patterns by pat, pat', pat_1, pat_2, ... and values by $v, v', v_1, v_2, \ldots$ Atoms are written in roman letters, while variables start with an uppercase letter. A *substitution* θ is a mapping from variables to expressions, and $\mathcal{D}\mathsf{om}(\theta) = \{X \in \mathit{Var} \mid X \neq \theta(X)\}$ is its domain. Substitutions are usually denoted by sets of bindings like, e.g., $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$. Substitutions are extended to morphisms from expressions to expressions in the natural way. The identity substitution is denoted by id. Composition of substitutions is denoted by juxtaposition, i.e., $\theta\theta'$ denotes a substitution θ'' such that $\theta''(X) = \theta'(\theta(X))$ for all $X \in \mathit{Var}$.

In a case expression "case e of pat_1 when $e_1 \to e'_1$; ...; pat_n when $e_n \to e'_n$ end", we first evaluate e to a value, say v; then, we find (if it exists) the first clause pat_i when $e_i \to e'_i$ such that v matches pat_i (i.e., there exists a substitution σ for the variables of pat_i such that $v = pat_i\sigma$) and $e_i\sigma$—the *guard*—reduces to *true*; then, the case expression reduces to $e'_i\sigma$. Note that guards can only contain calls to built-in functions (typically, arithmetic and relational operators).

Concurrent features.

In this work, we consider that a *system* is a pool of processes that can only interact through message sending and receiving (i.e., there is no shared memory). Each process has an associated *pid* (process identifier), which is unique in a system. Here, pids are ordinary values. Formally, a process is denoted by a tuple $\langle p, (\theta, e), q\rangle$ where p is the pid of the process, (θ, e) is the control—which consists of an environment (a substitution) and an expression to be evaluated—and q is the process' mailbox, a FIFO queue with the sequence of messages that have been sent to the process.

A running *system*, which we denote by $\Gamma; \Pi$, is composed by Γ, the *global mailbox*, which is a multiset of pairs of the form $(target\_process\_pid, message)$, and Π, which is a pool of processes. Π is denoted by an expression of the form

$$\langle p_1, (\theta_1, e_1), q_1\rangle \mid \cdots \mid \langle p_n, (\theta_n, e_n), q_n\rangle$$

Here, " | " denotes an associative and commutative operator. We typically denote a system by an expression of the form $\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi$ to point out that $\langle p, (\theta, e), q\rangle$

is an arbitrary process of the pool. Intuitively, Γ stores messages after they are sent, and before they are inserted in the target mailbox. Here, Γ (which is similar to the "ether" in [130]) is an artificial device used in our semantics to guarantee that all admissible message interleavings can be modeled.

In the following, we denote by $\overline{o_n}$ a sequence of syntactic objects $o_1, \ldots, o_n$ for some n.

The functions with side effects are self(), "!", spawn, and receive. The expression self() returns the pid of a process, while $p\,!\,v$ sends a message v to the process with pid p. New processes are spawned with a call of the form spawn$(a/n, [\overline{v_n}])$, so that the new process begins with the evaluation of apply a/n $(\overline{v_n})$. Finally, an expression "receive $\overline{pat_n}$ when $e_n \to e'_n$ end" traverses the messages in the process' queue until one of them matches a branch in the receive statement; i.e., it should find the *first* message v in the process' queue (if any) such that case v of pat_1 when $e_1 \to e'_1; \ldots; pat_n$ when $e_n \to e'_n$ end can be reduced; then, the receive expression evaluates to the same expression to which the above case expression would be evaluated, with the additional side effect of deleting the message v from the process' queue. If there is no matching message in the queue, the process *suspends* its execution until a matching message arrives.

Figure 4.2 shows an Erlang program implementing a simple client-server scheme with one server and two clients (a), as well as its translation into Core Erlang (b), where $\_C$, $\_X$ and $\_Y$ are anonymous variables introduced during the translation process to represent sequences of actions using let expressions. The execution starts with a call to function main/0. It first spawns two processes that execute functions server/0 and client/1, respectively, and then calls to function client/1 too. Client requests have the form $\{P, \mathrm{req}\}$, where P is the pid of the client. The server receives the message, returns a message ack to the client, and calls to function server/0 again in an endless loop. After processing the two requests, the server will suspend waiting for another request.

Following [86], the semantics of the language is defined in a modular way, so that the labeled transition relation $\xrightarrow{\ell}$ models the evaluation of *expressions* and $\hookrightarrow$ models the reduction of *systems*. Relation $\xrightarrow{\ell}$ follows a typical call-by-value semantics for side-effect free expressions;[1] in this case, reduction steps are labeled with τ. For the remaining functions, the expression rules cannot complete the reduction of an expression since some information is not *locally* available. In these cases, the steps are labeled with the information needed to complete the reduction within the system

[1]Because of lack of space, we are not presenting the rules of $\xrightarrow{\ell}$ here, but refer the interested reader to [86].

```
main() ->
  S = spawn(server/0, []),
  spawn(client/1, [S]),
  client(S).
server() ->
  receive
    {P, req} ->
      P ! ack,
      server()
  end.
client(S) ->
  S ! {self(), req},
  receive
    ack -> ok
  end.
```

(a) Erlang

$$\begin{aligned}
&\mathrm{main}/0 = \mathsf{fun}\ () \to \mathsf{let}\ S = \mathsf{spawn}(\mathrm{server}/0, [\,])\\
&\qquad\qquad\qquad\qquad\ \mathsf{in\ let}\ _C = \mathsf{spawn}(\mathrm{client}/1, [S])\\
&\qquad\qquad\qquad\qquad\ \mathsf{in\ apply}\ \mathrm{client}/1\ (S)\\
&\mathrm{server}/0 = \mathsf{fun}\ () \to \mathsf{receive}\\
&\qquad\qquad\qquad\qquad\quad \{P, \mathrm{req}\} \to\\
&\qquad\qquad\qquad\qquad\quad\ \mathsf{let}\ _X = P\,!\,\mathrm{ack}\\
&\qquad\qquad\qquad\qquad\quad\ \mathsf{in\ apply}\ \mathrm{server}/0\ ()\\
&\qquad\qquad\qquad\qquad \mathsf{end}\\
&\mathrm{client}/1 = \mathsf{fun}\ (S) \to \mathsf{let}\ _Y = S\,!\,\{\mathsf{self}(), \mathrm{req}\}\\
&\qquad\qquad\qquad\qquad \mathsf{in\ receive}\\
&\qquad\qquad\qquad\qquad\quad\ \mathrm{ack} \to \mathrm{ok}\\
&\qquad\qquad\qquad\qquad\ \ \mathsf{end}
\end{aligned}$$

(b) Core Erlang

Figure 4.2: A simple client server

rules of Figure 4.3. For sending a message, an expression $p'' \,!\, v$ is reduced to v with the side-effect of (eventually) storing the message v in the mailbox of process p''. The associated label is thus $\mathsf{send}(p'', v)$ so that rule *Send* can complete the step by adding the pair (p'', v) to the global mailbox Γ.

The remaining functions, receive, spawn and self, are reduced to a fresh distinguished symbol κ (a sort of *future*) in the expression rules, since the value cannot be determined locally. Therefore, in these cases, the labels also include κ. Then, the system rules of Figure 4.3 will bind κ to its correct value: the selected expression in rule *Receive* and a pid in rules *Spawn* and *Self*.

To be more precise, for a receive statement, the label has the form $\mathsf{rec}(\kappa, \overline{cl_n})$ where $\overline{cl_n}$ are the clauses of the receive statement. In rule *Receive*, the auxiliary function matchrec is used to find the first message in the queue that matches a clause, then returning a triple with the matching substitution θ_i, the selected branch e_i and the selected message v. Here, $q \backslash\backslash v$ denotes a new queue that results from q by removing the oldest occurrence of message v.

For a spawn, the label has the form $\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])$, where a/n and $[\overline{v_n}]$ are the arguments of spawn. Rule *Spawn* then adds a new process with a fresh pid p' initialized with the application $\mathsf{apply}\ a/n\ (v_1, \ldots, v_n)$ and an empty queue.

$$(Seq)\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'), q\rangle \mid \Pi}$$

$$(Send)\quad \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma \cup (p'', v); \langle p, (\theta', e'), q\rangle \mid \Pi}$$

$$(Receive)\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\overline{cl_n}, q) = (\theta_i, e_i, v)}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\backslash v\rangle \mid \Pi}$$

$$(Spawn)\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \mid \langle p', (id, \mathsf{apply}\ a/n\ (\overline{v_n})), [\,]\rangle \mid \Pi}$$

$$(Self)\quad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi}$$

$$(Sched)\quad \frac{}{\Gamma \cup \{(p, v)\}; \langle p, (\theta, e), q\rangle \mid \Pi \hookrightarrow \Gamma; \langle p, (\theta, e), v : q\rangle \mid \Pi}$$

Figure 4.3: Standard semantics: system rules

For a self, only κ is needed in the label. Rule $Self$ then proceeds in the obvious way by binding κ to the pid of the process.

The rules presented so far allow one to store messages in the global mailbox, but not to deliver them. This is the task of the scheduler, which is modeled by rule $Sched$. This rule nondeterministically chooses a pair (p, v) in the global mailbox Γ and delivers the message v to the target process p. Note also that Γ is a multiset, so we use "$\cup$" as multiset union.

4.3 Causal-Consistent Reversible Debugging

In this section, we present a causal-consistent reversible semantics for the considered language. The semantics is based on the reversible semantics for Erlang introduced in [109, 86]. In particular, [86] presents an *uncontrolled* reversible semantics, which is highly non-deterministic, and a *controlled* semantics that performs a backward computation up to a given *checkpoint* in a mostly deterministic way. Here, we build on the uncontrolled semantics, and define a new controlled semantics which is more appropriate as a basis for a causal-consistent reversible debugger than the one in [86].

First, following [86], we introduce an instrumented version of the standard semantics. For this purpose, we exploit a typical Landauer's embedding [80] and include a "history" h in the states. In contrast to the standard semantics, messages

$$(Seq)\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \tau(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi}$$

$$(Send)\quad \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e' \quad \lambda \text{ is a fresh identifier}}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma \cup (p'', \{v, \lambda\}); \langle p, \mathsf{send}(\theta, e, p'', \{v, \lambda\}) : h, (\theta', e'), q\rangle \mid \Pi}$$

$$(Receive)\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\overline{cl_n}, q) = (\theta_i, e_i, \{v, \lambda\})}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{rec}(\theta, e, \{v, \lambda\}, q) : h, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\backslash \{v, \lambda\}\rangle \mid \Pi}$$

$$(Spawn)\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\begin{array}{ll}\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup & \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \\ & \mid \langle p', [\,], (id, \mathsf{apply}\ a/n\ (\overline{v_n})), [\,]\rangle \mid \Pi\end{array}}$$

$$(Self)\quad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{self}(\theta, e) : h, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi}$$

$$(Sched)\quad \frac{}{\Gamma \cup \{(p, \{v, \lambda\})\}; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, h, (\theta, e), \{v, \lambda\} : q\rangle \mid \Pi}$$

Figure 4.4: Forward reversible semantics

now include a unique identifier (i.e., a timestamp λ). These identifiers are required to avoid mixing different messages with the same value (and possibly also with the same sender and/or receiver). More details can be found in [86].

The transition rules of the forward reversible semantics can be found in Figure 4.4. They are an easy—and conservative—extension of the semantics in Figure 4.3 by adding histories to processes. In the histories, we use terms headed by constructors τ, check, send, rec, spawn, and self to record the steps performed by the forward semantics. Note that the auxiliary function matchrec now deals with messages of the form $\{v, \lambda\}$, trivially extending the original function in the standard semantics by ignoring λ when computing the first matching message.

Rollback Debugging Semantics.

Now, we introduce a novel *rollback semantics* to undo the actions of a given process. Here, processes in "rollback" mode are annotated using $\lfloor\ \rfloor_\Psi$, where Ψ is a set with the requested rollbacks. In particular, we consider the following rollbacks to undo the actions of a given process in a causal-consistent way:

- s: one backward step;
- $\lambda^{\Uparrow}$: a backward derivation up to the sending of a message labeled with λ;

- $\lambda^{\Downarrow}$: a backward derivation up to the delivery of a message labeled with λ;
- λ^{rec}: a backward derivation up to the receive of a message labeled with λ;
- sp_p: a backward derivation up to the spawning of the process with pid p;
- sp: a backward derivation up to the creation of the annotated process;
- X: a backward derivation up to the introduction of variable X.

In the following, in order to simplify the reduction rules, we consider that our semantics satisfies the following *structural equivalence*:

$$\begin{array}{ll} (SC1) & \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\varnothing} \mid \Pi \equiv \Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi \\ (SC2) & \Gamma; \lfloor \langle p, [\,], (\theta, e), [\,] \rangle \rfloor_{\Psi} \mid \Pi \equiv \Gamma; \langle p, [\,], (\theta, e), [\,] \rangle \mid \Pi \end{array}$$

Therefore, when the set of rollbacks is empty or the process is back to its initial state, we consider that the required rollback has been completed.

Our rollback debugging semantics is modeled with the reduction relation $\leadarrow$, defined by the rules in Figure 4.5. Here, we assume that $\Psi \neq \varnothing$ (but Ψ' might be empty). Let us briefly explain the rules of the rollback semantics:

- Some actions can be directly undone. This is the case dealt with by rules $\overline{Seq}$, $\overline{Send1}$, $\overline{Receive}$, $\overline{Spawn1}$, $\overline{Self}$, and $\overline{Sched}$. In every rule, we remove the corresponding rollback request from Ψ. In particular, all of them remove s (since a causal-consistent step has been performed). Rule $\overline{Seq}$ additionally removes the variables whose bindings were introduced in the last step; rule $\overline{Send1}$ removes $\lambda^{\Uparrow}$ (representing the sending of the message with identifier λ); rule $\overline{Receive}$ removes λ^{rec} (representing the receiving of the message with identifier λ); rule $\overline{Spawn1}$ removes $\mathsf{sp}_{p''}$ (representing the spawning of the process with pid p''); and rule $\overline{Sched}$ removes $\lambda^{\Downarrow}$ (representing the delivery of the message with identifier λ). Note also that rule $\overline{Sched}$ requires a side condition to avoid the (incorrect) commutation of rules $\overline{Receive}$ and $\overline{Sched}$ (see [86] for more details on this issue).
- Other actions require some dependencies to be undone first. This is the case of rules $\overline{Send2}$ and $\overline{Spawn2}$. In the first case, rule $\overline{Send2}$ applies in order to "propagate" the rollback mode to the receiver of the message, so that rules $\overline{Sched}$ and $\overline{Send1}$ can be eventually applied. In the second case, rule $\overline{Spawn2}$ applies to propagate the rollback mode to process p'' so that, eventually, rule $\overline{Spawn1}$ can be applied. Observe that the rollback sp introduced by the rule

$$(\overline{Seq}) \quad \Gamma; \lfloor \langle p, \tau(\theta, e) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi \rightharpoonup \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\Psi \setminus (\{\mathsf{s}\} \cup \mathcal{V})} \mid \Pi \quad \text{where } \mathcal{V} = \mathcal{D}\mathsf{om}(\theta') \setminus \mathcal{D}\mathsf{om}(\theta)$$

$$(\overline{Send1}) \quad \Gamma \cup \{(p', \{v, \lambda\})\}; \lfloor \langle p, \mathsf{send}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi \rightharpoonup \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\Psi \setminus \{\mathsf{s}, \lambda^{\Uparrow}\}} \mid \Pi$$

$$(\overline{Send2}) \quad \Gamma; \lfloor \langle p, \mathsf{send}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \langle p', h', (\theta'', e''), q' \rangle \rfloor_{\Psi'} \mid \Pi \rightharpoonup \Gamma; \lfloor \langle p, \mathsf{send}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \langle p', h', (\theta'', e''), q' \rangle \rfloor_{\Psi' \cup \{\lambda^{\Downarrow}\}} \mid \Pi \quad \text{if } (p', \{v, \lambda\}) \text{ does not occur in } \Gamma \text{ and } \lambda^{\Downarrow} \notin \Psi'$$

$$(\overline{Receive}) \quad \Gamma; \lfloor \langle p, \mathsf{rec}(\theta, e, \{v, \lambda\}, q) : h, (\theta', e'), q \backslash\backslash \{v, \lambda\} \rangle \rfloor_\Psi \mid \Pi \rightharpoonup \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\Psi \setminus \{\mathsf{s}, \lambda^{\mathsf{rec}}\}} \mid \Pi$$

$$(\overline{Spawn1}) \quad \Gamma; \lfloor \langle p, \mathsf{spawn}(\theta, e, p'') : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \langle p'', [\,], (\theta'', e''), [\,] \rangle \rfloor_{\Psi'} \mid \Pi \rightharpoonup \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\Psi \setminus \{\mathsf{s}, \mathsf{sp}_{p''}\}} \mid \Pi$$

$$(\overline{Spawn2}) \quad \Gamma; \lfloor \langle p, \mathsf{spawn}(\theta, e, p'') : h, (\theta, e), q \rangle \rfloor_\Psi \mid \lfloor \langle p'', h'', (\theta'', e''), q'' \rangle \rfloor_{\Psi'} \mid \Pi \rightharpoonup \Gamma; \lfloor \langle p, \mathsf{spawn}(\theta, e, p'') : h, (\theta, e), q \rangle \rfloor_\Psi \mid \lfloor \langle p'', h'', (\theta'', e''), q'' \rangle \rfloor_{\Psi' \cup \{\mathsf{sp}\}} \mid \Pi \quad \text{if } h'' \neq [\,] \vee q'' \neq [\,] \text{ and } \mathsf{sp} \notin \Psi'$$

$$(\overline{Self}) \quad \Gamma; \lfloor \langle p, \mathsf{self}(\theta, e) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi \rightharpoonup \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\Psi \setminus \{\mathsf{s}\}} \mid \Pi$$

$$(\overline{Sched}) \quad \Gamma; \lfloor \langle p, h, (\theta, e), \{v, \lambda\} : q \rangle \rfloor_\Psi \mid \Pi \rightharpoonup \Gamma \cup (p, \{v, \lambda\}); \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\Psi \setminus \{\mathsf{s}, \lambda^{\Downarrow}\}} \mid \Pi \quad \text{if the topmost } \mathsf{rec}(\ldots) \text{ item in } h \text{ (if any) has the form } \mathsf{rec}(\theta', e', \{v', \lambda'\}, q') \text{ with } q' \backslash\backslash \{v', \lambda'\} \neq \{v, \lambda\} : q$$

Figure 4.5: Rollback debugging semantics

> $\overline{Spawn2}$ does not need to be removed from Ψ since the complete process is deleted from Π in rule $\overline{Spawn1}$.

The correctness of the new rollback semantics can be shown following a similar scheme as in [86] for proving the correctness of the rollback semantics for checkpoints.

We now introduce an operator that performs a causal-consistent backward derivation and is parameterized by a system, a pid and a set of rollback requests:

$$\mathsf{rb}(\Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi, p, \Psi) = \Gamma'; \Pi' \text{ if } \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_\Psi \mid \Pi \rightharpoonup^* \Gamma'; \Pi' \not\rightharpoonup$$

The operator adds a set of rollback requests to a given process[2] and then performs

[2] Actually, in this work, we only consider a single rollback request at a time, so Ψ is always a singleton. Nevertheless, our formalization considers that Ψ is a set for notational convenience and, also, in order to accept multiple rollbacks in the future.

as many steps as possible using the rollback debugging semantics.

By using the above parametric operator, we can easily define several rollback operators that are useful for debugging. Our first operator, $\mathsf{rollback}(\Gamma; \Pi, p)$, just performs a causal-consistent backward step for process p:

$$\mathsf{rollback}(\Gamma; \Pi, p) = \mathsf{rb}(\Gamma; \Pi, p, \{\mathsf{s}\})$$

Notice that this may trigger the execution of any number of backward steps in other processes in order to first undo the consequences, if any, of the step in p.

This operator can easily be extended to an arbitrary number of steps:

$$\mathsf{rollback}(\Gamma; \Pi, p, n) = \begin{cases} \Gamma; \Pi & \text{if } n = 0 \\ \mathsf{rollback}(\Gamma'; \Pi', p, n-1) & \text{if } n > 0 \text{ and} \\ & \mathsf{rollback}(\Gamma; \Pi, p) = \Gamma'; \Pi' \end{cases}$$

Also, we might be interested in going backward until a relevant action is undone. For instance, we introduce below operators that go backward up to, respectively, the sending of a message with a particular identifier λ, the receiving of a message with a particular identifier λ, and the spawning of a process with pid p':

$$\begin{array}{lcl} \mathsf{rollback}(\Gamma; \Pi, p, \lambda^{\Uparrow}) & = & \mathsf{rb}(\Gamma; \Pi, p, \{\lambda^{\Uparrow}\}) \\ \mathsf{rollback}(\Gamma; \Pi, p, \lambda^{\mathsf{rec}}) & = & \mathsf{rb}(\Gamma; \Pi, p, \{\lambda^{\mathsf{rec}}\}) \\ \mathsf{rollback}(\Gamma; \Pi, p, \mathsf{sp}_{p'}) & = & \mathsf{rb}(\Gamma; \Pi, p, \{\mathsf{sp}_{p'}\}) \end{array}$$

Note that p is a parameter of the three operators, but it could also be automatically computed (from λ in the first two rules, from p' in the last one) by inspecting the histories of the processes in Π. This is actually what CauDEr does.

Finally, we consider an operator that performs backward steps up to the introduction of a binding for a given variable:

$$\mathsf{rollback}(\Gamma; \Pi, p, X) = \mathsf{rb}(\Gamma; \Pi, p, \{X\})$$

Here, p cannot be computed automatically from X, since variables are local and, hence, variable X may occur in several processes; thus, p is needed to uniquely identify the process of interest.[3]

4.4 CauDEr: A Causal-Consistent Reversible Debugger

The CauDEr implementation is conveniently bundled together with a graphical user interface to facilitate the interaction of users with the reversible debugger.

[3] Actually, in CauDEr, uniqueness of variable names is enforced via renaming.

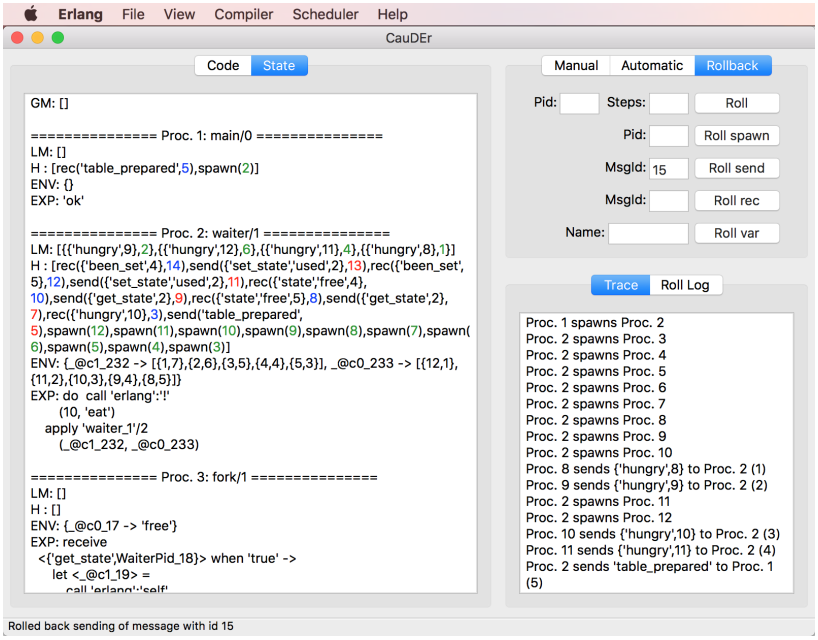


Figure 4.6: CauDEr screenshot

CauDEr works as follows: when it is started, the first step is to select an Erlang source file. The selected source file is then translated into Core Erlang, and the resulting code is shown in the Code tab. Then, the user can choose any of the functions from the module and write the arguments that she wants to evaluate the function with. An initial system state, with an empty global mailbox and a single process performing the specified function application, appears in the State tab when the user presses the START button. Now, the user can explore possible program executions both forward and backward, according to three different modes, corresponding to the three tabs on the top right of the window in Figure 4.6. In the Manual mode, the user selects a process or message identifier, and buttons corresponding to forward and backward enabled reductions for the chosen process/message are available. Note that a backward reduction is *enabled* only if the action has no causal dependencies that need to be undone (single backward reductions correspond to applications of rules $\overline{Seq}$, $\overline{Send1}$, $\overline{Receive}$, $\overline{Spawn1}$, $\overline{Self}$, and $\overline{Sched}$ in Figure 4.5, see the uncontrolled reversible semantics in [86] for more details). In the Automatic mode one can decide the direction (forward or backward) and the number of steps to be performed. Actual steps are selected by a suitable scheduler. Currently, two (random) schedulers are available, one of which gives priority to processes w.r.t.

the scheduling of messages (as in the "normalization" strategy described in [86]), while the other has a uniform distribution. None of these schedulers mimics the Erlang/OTP scheduler. Indeed, it would be very hard to replicate this behavior, as it depends on many parameters (threads, workload, etc). However, this is not necessary, since we are only interested in reproducing the errors that occur in actual executions, and we discuss in future work how to obtain this without the need of mimicking the Erlang/OTP scheduler. The Automatic tab also includes a Normalize button, that executes all enabled actions but message schedulings. The last tab, Rollback, implements the rollback operators described in Section 4.3.

While exploring the execution, two tabs are updated to provide information on the system and its execution. The State tab describes the current system, including the global mailbox GM, and, for each process, the following components: the local mailbox LM, the history H, the environment ENV, and the expression under evaluation EXP. Identifiers of messages are highlighted in color. This tab can be configured to hide any component of the process representation. Also, we consider two levels of abstraction for both histories and environments: for histories, we can either show all the actions or just the concurrent actions (send, receive and spawn); for environments, we can either show all variable bindings (called the *full* environment) or only the bindings for those variables occurring in the current expression (called the *relevant* environment).

The Trace tab gives a linearized description of the concurrent actions performed in the system, namely sends and receives of messages, and spawns of processes. This is aimed at giving a global picture of the system evolution, to highlight anomalies that might be caused by bugs.

A further tab is available, Roll Log, which is updated in case of rollbacks. It shows which actions have been actually undone upon a rollback request. This tab allows one to understand the causal dependencies of the target process of the rollback request, frequently highlighting undesired or missing dependencies directly caused by bugs.

The release version (v1.0) of CauDEr is fully written in Erlang, and it is publicly available from `https://github.com/mistupv/cauder` under the MIT license. The only requirement to build the application is to have Erlang/OTP installed and built with wxWidgets. The repository also includes some documentation and a few examples to easily test the application.

4.4.1 The CauDEr Workflow

A typical debugging session with CauDEr proceeds as follows. First, the user may run the program some steps forward using the Automatic mode in order to exercise the code. After each sequence of forward steps, she looks at the program output (which is not on the CauDEr window, but in the console where CauDEr has been launched) and possibly at the State and Trace tabs to check for abnormal behaviors. The State tab helps to identify these behaviors within a single process, while the Trace tab highlights anomalies in the global behavior.

If the user identifies an unexpected action, she can undo it by using any (or a combination) of the available rollback commands. The Roll Log tab provides information on the causal-consistent rollbacks performed (in some cases, this log is enough to highlight the bug). From there, the user typically switches to the Manual mode in order to precisely control the doing or undoing of actions in a specific state. This may involve performing other rollbacks to reach previous states. Our experience says that inspecting the full environment during the Manual exploration is quite helpful to locate bugs caused by sequential code.

4.4.2 Finding Concurrency Bugs with CauDEr

We use as a running example to illustrate the use of our debugger the well-known problem of dining philosophers. Here, we have a process for each philosopher and for each fork. We avoid implementations that are known to deadlock by using an arbitrator process, the waiter, that acts as an intermediary between philosophers and forks. In particular, if a philosopher wants to eat, he asks the waiter to get the forks. The waiter checks whether both forks are free or not. In the first case, he asks the forks to become used, and sends a message eat to the philosopher. Otherwise he sends a message think to the philosopher. When a philosopher is done eating, he sends a message eaten to the waiter, who in turn will release (i.e., set to free) the corresponding forks. The full Erlang code of the (correct) example, `dining.erl`, is available from `https://github.com/mistupv/dining-philos` .

Message order violation scenario.

Here, we consider the buggy version of the program that can be found in file `dining_simple_bug.erl` of the above repository. In this example, running the program forward using the Automatic mode for about 600 steps is enough to discern something wrong. In particular, the user notices in the output that some philosophers

are told to think when they should be told to eat, even at the beginning of the execution. Since the bug appears so early, it is probably a local bug, hence the user first focuses on the State tab. When the user considers the waiter process, she sees in the history an unexpected sequence of concurrent events of the following form (shown in reverse chronological order):

```
. . . ,send('think',10),rec('free',9),send({'get_state',2},8),
rec({'hungry',12},6),send({'get_state',2},7),rec({'hungry',9},2), . . .
```

Here, the waiter has requested the state of a fork with send({'get_state',2},7), where 2 is the process id of the waiter itself and 7 the message id. Unexpectedly, the waiter has received a message hungry as a reply, instead of a message free or used. To get more insight on this, the user decides to rollback the receive of {'hungry',12}, which has 6 as message id. As a result, the rollback gets the system back to a state where send({'get_state',2},7) is the last concurrent event for the waiter process. Finally, the user switches to the Manual mode and notices that the next available action for the waiter process is to receive the message {'hungry',12} in the receive construct from the ask_state function. Function ask_state is called by the waiter process when it receives a hungry request from a philosopher (to get the state of the two forks). Obviously, a further message hungry should not be received here. The user easily realizes then that the pattern in the receive is too general (in fact, it acts as a catch-all clause) and, as a result, the receive is matching also messages from other forks and even philosophers. Indeed, after sending the message get_state to a fork, the programmer assumed that the next incoming message will be the state of the fork. However, the function is being evaluated in the context of the waiter process, where many other messages could arrive, e.g., messages hungry or eaten from philosophers.

It would not be easy to find the same bug using a standard debugger. Indeed, one would need to find where the wrong message hungry is sent, and put there a breakpoint. However, in many cases, no scheduling error will occur, hence many attempts would be needed. With a standard reversible debugger (like Actoverse [126]) one could look for the point where the wrong message is received, but it would be difficult to stop the execution at the exact message. Watch points do not help much, since all such messages are equal, but only some of them are received in the wrong receive operation. Indeed, in this example, the CauDEr facility of rollbacking a specific message receiving, coupled with the addition of unique identifiers to messages, is a key in ensuring the success of the debugging session.

Livelock scenario.

Now, we consider the buggy version of the dining philosophers that can be found in file `dining_bug.erl` of our repository. In this case, the output of the program shows that, after executing some 2000 steps with the Automatic mode, some philosophers are always told to think, while others are always told to eat. In contrast to the previous example, this bug becomes visible only late in the execution, possibly only after some particular pattern of message exchanges has taken place (this is why it is harder to debug). In order to analyze the message exchanges the user should focus on the Trace tab first. By carefully examining it, the user realizes that, in some cases, after receiving a message eaten from a philosopher, the waiter sends the two messages {'set_state','free',2} to release the forks to the same fork:

Proc. 2 receives {'eaten',10} (28)
Proc. 2 sends {'set_state','free',2} to Proc. 5 (57)
Proc. 5 receives {'set_state','free',2} (57)
Proc. 5 sends {'been_set',5} to Proc. 2 (58)
Proc. 2 receives {'been_set',5} (58)
Proc. 2 sends {'set_state','free',2} to Proc. 5 (59)
Proc. 5 receives {'set_state','free',2} (59)
Proc. 5 sends {'been_set',5} to Proc. 2 (60)
Proc. 2 receives {'been_set',5} (60)

Then, the user rollbacks the sending of the last message from the waiter process (the one with message id 59) and chooses to show the full environment (a clever decision). Surprisingly, the computed values for LeftForkId and RightForkId are equal. She decides to rollback also the sending of message with id 57, but she cannot see anything wrong there, so the computed value for RightForkId must be wrong. Now the user focuses on the corresponding line on the code, and she notices that the operands of the modulo operator have been swapped, which is the source of the erroneous behavior.

This kind of livelocks are typically hard to find with other debugging tools. For instance, Concuerror [61] requires a finite computation, which is not the case in this scenario where the involved processes keep doing actions all the time but no global progress is achieved (i.e., some philosophers never eat).

4.5 Related Work

Causal-consistent debugging has been introduced by CaReDeb [51], in the context of language μOz. The present paper improves on CaReDeb in many directions. First, μOz is only a toy language where no realistic programs can be written (e.g., it supports only integers and a few arithmetic operations). Second, μOz is not distributed, since messages are atomically moved from the sender to a message queue, and from the queue to the target process. This makes its causality model, hence the definition of a causal-consistent reversible semantics, much simpler. Third, in [51] the precise semantics of debugging operators is not fully specified. Finally, the implementation described in [51] is just a proof-of-concept.

More in general, our work is in the research thread of causal-consistent reversibility (see [83] for a survey), first introduced in [35] in the context of process calculus CCS. Most of the works in this area are indeed on process calculi, but for the work on μOz already discussed (the theory was introduced in [90]) and a line of work on the coordination language μklaim [53]. However, μklaim is a toy language too. Hence, we are the first ones to consider a mainstream programming language. A first approach to the definition of a causal-consistent semantics of Erlang was presented in [109], and extended in [86]. While we based CauDEr on the uncontrolled semantics therein (and on its proof-of-concept implementation), we provided in the present paper an updated controlled semantics more suitable for debugging, and a mature implementation with a complete interface and many facilities for debugging. Moreover, our tool is able to deal with a larger subset of the language, mainly in terms of built-in functions and data structures.

While CaReDeb is the only other causal-consistent debugger we are aware of, two other reversible debuggers for actor systems exist. Actoverse [126] deals with Akka-based applications. It provides many relevant features which are complementary to ours. These include a partial-order graphical representation of message exchanges that would nicely match our causal-consistent approach, message-oriented breakpoints that allow one to force specific interleavings in message schedulings, and facilities for session replay to ensure bugs reappear when executing forward again. In contrast, Actoverse provides less facilities for state inspection and management than us (e.g., it has nothing similar to our Roll var command). Also, the paper does not include any theoretical framework defining the behavior of the debugger. EDD is a declarative debugger for Erlang (see [20] for a version dealing with sequential Erlang). EDD tracks the concurrent actions of an execution and allows the user to select any of them to start the questions. Declarative debugging is essentially orthogonal to our approach.

Causeway [127] is not a full-fledged debugger but a post-mortem trace analyzer, i.e., it performs no execution, but just explores a trace of a run. It concentrates on message passing aspects, e.g., it does not allow one to explore the state of single processes (states are not in the logs analyzed by Causeway). On the contrary it provides nice mechanisms to abstract and filter different kinds of communications, allowing the user to decide at each stage of the debugging process which messages are of interest. These mechanisms would be an interesting addition for CauDEr.

4.6 Discussion

In this work, we have presented the design of CauDEr, a causal-consistent reversible debugger for Erlang. It is based on the reversible semantics introduced in [109, 86], though we have introduced in this paper a new rollback semantics which is especially appropriate for debugging Erlang programs. We have shown in the paper that some bugs can be more easily located using our new tool, thus filling a gap in the collection of debugging tools for Erlang.

Currently, our debugger may run a program either forward or backward (in the latter case, in a causal-consistent way). After a backward computation that undoes some steps, we can resume the forward computation, though there are no guarantees that we will reproduce the previous forward steps. Some debuggers (so-called omniscient or back-in-time debuggers) allow us to move both forward and backward along a *particular* execution. As a future work, we plan to define a similar approach but ensuring that once we resume a forward computation, we can follow the same previous forward steps *or some other causal-consistent steps*. Such an approach might be useful, e.g., to determine which processes depend on a particular computation step and, thus, ease the location of a bug.

Another interesting line of future work involves the possibility of capturing a faulty behavior during execution in the standard environment, and then replaying it in the debugger. For instance, we could instrument source programs so that their execution in a standard environment writes a log in a file. Then, when the program ends up with an error, we could use this log as an input to the debugger in order to explore this particular faulty behavior (as postmortem debuggers do). This approach can be applied even if the standard environment is distributed and there is no common notion of time, since causal-consistent reversibility relies only on a notion of causality.

For the same reason we could also develop a fully distributed debugger, where each process is equipped with debugging facilities, and a central console allows us

to coordinate them. This would strongly improve scalability, since most of the computational effort (running and backtracking programs) would be distributed. However, this step requires a semantics without any synchronous interaction (e.g., rules $\overline{Send2}$ and $\overline{Spawn2}$ would need to be replaced by a more complex asynchronous protocol).

Acknowledgments

The authors gratefully acknowledge the anonymous referees for their useful comments and suggestions.

Chapter 5

Causal-Consistent Replay Debugging for Message Passing Programs

Ivan Lanese[1], Adrián Palacios[2], Germán Vidal[2]

[1] Focus Team, University of Bologna/INRIA
ivan.lanese@gmail.com

[2] MiST, DSIC, Universitat Politècnica de València
{apalacios,gvidal}@dsic.upv.es

Abstract. Debugging of concurrent systems is a tedious and error-prone activity. A main issue is that there is no guarantee that a bug that appears in the original computation is replayed inside the debugger. This problem is usually tackled by so-called replay debugging, which allows the user to record a program execution and replay it inside the debugger. In this paper, we present a novel technique for replay debugging that we call *controlled causal-consistent*

This work has been partially supported by the EU (FEDER) and the *Spanish Ministerio de Ciencia, Innovación y Universidades*/AEI (MICINN) under grant TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grants PROMETEO-II/2015/013 (SmartLogic) and Prometeo/2019/098 (DeepTrust), and by the COST Action IC1405 on Reversible Computation - extending horizons of computing. The first author has been also partially supported by French ANR project DCore ANR-18-CE25-0007. The second author has been also supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores* (MICINN) under FPI grant BES-2014-069749. This chapter is an adapted author version of the paper published in *"Ivan Lanese, Adrián Palacios, Germán Vidal: Causal-Consistent Replay Debugging for Message Passing Programs. Proceedings of FORTE 2019, Lecture Notes in Computer Science 11535: 167–184 (2019)"*. DOI: https://doi.org/10.1007/978-3-030-21759-4_10

replay. Controlled causal-consistent replay allows the user to record a program execution and, in contrast to traditional replay debuggers, to reproduce a visible misbehavior inside the debugger including all *and only* its causes. In this way, the user is not distracted by the actions of other, unrelated processes.

5.1 Introduction

Debugging is a main activity in software development. According to a 2014 study [136], the cost of debugging is $312 billions annually. Another recent study [17] estimates that the time spent in debugging is 49.9% of the total programming time. The situation is not likely to improve in the near future, given the increasing demand of concurrent and distributed software. Indeed, distribution is inherent in current computing platforms, such as the Internet or the Cloud, and concurrency is a must to overcome the advent of the power wall [129]. Debugging concurrent and distributed software is clearly more difficult than debugging sequential code [66]. Furthermore, misbehaviors may depend, e.g., on the execution speed of the different processes, showing up only in some (sometimes rare) cases.

A particularly unfortunate situation is when a program exhibits a misbehavior in its usual execution environment, but it runs smoothly when re-executed in the debugger. This problem is usually tackled by so-called replay debugging, which allows the user to record a program execution and replay it inside the debugger. However, in concurrent programs, part of the execution may not be relevant: some processes may not have interacted with the one showing a misbehavior, or may have interacted with it only at the very beginning of their execution, hence most of their execution is not relevant for the debugging session. Having to replay all these behaviors is both time and resource consuming as well as distracting for the user.

Our main contribution in this paper is a novel technique for replay debugging that we call *controlled causal-consistent replay*. It extends the techniques in the literature as follows: given a log of a (typically faulty) concurrent execution, we do not replay exactly the same execution step by step (as traditional replay debuggers), but we allow the user to select any action in the log (e.g., one showing a misbehavior) and to replay the execution up to this action, including all *and only* its causes. This allows one to focus on those processes where (s)he thinks the bug(s) might be, disregarding the actual interleaving of processes. To the best of our knowledge, the notion of controlled causal-consistent replay is new.

We fully formalize causal-consistent replay for (a subset of) a realistic functional and concurrent programming language based on message-passing: Erlang. Moreover, we prove relevant properties, e,g., that misbehaviors in the original compu-

$$
\begin{array}{rcl}
program & ::= & fun_1 \ \ldots \ fun_n \\
fun & ::= & fname = \mathsf{fun}\ (X_1, \ldots, X_n) \rightarrow expr \\
fname & ::= & Atom/Integer \\
lit & ::= & Atom \mid Integer \mid Float \mid [\,] \\
expr & ::= & Var \mid lit \mid fname \mid [expr_1|expr_2] \mid \{expr_1, \ldots, expr_n\} \\
 & \mid & \mathsf{call}\ expr\ (expr_1, \ldots, expr_n) \mid \mathsf{apply}\ expr\ (expr_1, \ldots, expr_n) \\
 & \mid & \mathsf{case}\ expr\ \mathsf{of}\ clause_1; \ldots; clause_m\ \mathsf{end} \\
 & \mid & \mathsf{let}\ Var = expr_1\ \mathsf{in}\ expr_2 \mid \mathsf{receive}\ clause_1; \ldots; clause_n\ \mathsf{end} \\
 & \mid & \mathsf{spawn}(expr, [expr_1, \ldots, expr_n]) \mid expr_1\ !\ expr_2 \mid \mathsf{self}() \\
clause & ::= & pat\ \mathsf{when}\ expr_1 \rightarrow expr_2 \\
pat & ::= & Var \mid lit \mid [pat_1|pat_2] \mid \{pat_1, \ldots, pat_n\}
\end{array}
$$

Figure 5.1: Language syntax rules

tation are always replayed, and that we guarantee minimal replay of observable behaviors. This is in contrast with most approaches to replay in the literature, that, beyond considering different languages, are either fully experimental (like, e.g., [89, 96, 137, 8]), or present limited theoretical results, as in [106, 65, 70].

Causal-consistent replay can be seen as the dual of causal-consistent rollback, a technique for reversible computing which allows one to select an action in a computation and undo it, including all *and only* its consequences. Indeed, the two techniques integrate well, giving rise to a framework to explore back and forward a given concurrent computation, always concentrating on the actions of interest and avoiding unrelated actions. By lack of space, we will only present causal-consistent replay in this paper. More details, including the integration with causal-consistent rollback, proofs of technical results, and a description of an implemented reversible replay debugger for Erlang [87] that follows the ideas in this paper, can be found in an accompanying technical report [88]. While not technically needed, printing the paper in color may help the understanding.

5.2 The Language

We present below the considered language: a first-order functional and concurrent language based on message passing that mainly follows the actor model.

Language Syntax.

The syntax of the language is in Figure 5.1. A program is a sequence of function definitions, where each function name f/n (atom/arity) has an associated definition fun $(X_1, \ldots, X_n) \to e$, where $X_1, \ldots, X_n$ are (distinct) fresh variables and are the only variables that may occur free in e. The body of a function is an *expression*, which can include variables, literals, function names, lists (using Prolog-like notation: $[\,]$ is the empty list and $[e_1|e_2]$ is a list with head e_1 and tail e_2), tuples (denoted by $\{e_1, \ldots, e_n\}$),[1] calls to built-in functions (mainly arithmetic and relational operators), function applications, case expressions, let bindings, receive expressions, spawn (for creating new processes), "!" (for sending a message), and self. As is common practice, we assume that X is a fresh variable in let $X = expr_1$ in $expr_2$.

In this language, we distinguish expressions, patterns, and values, ranged over respectively by $e, e', e_1, \ldots$, by $pat, pat', pat_1, \ldots$ and by $v, v', v_1, \ldots$. In contrast to expressions, *patterns* are built from variables, literals, lists, and tuples. Patterns can only contain fresh variables. Finally, *values* are built from literals, lists, and tuples. Atoms (i.e., constants with a name) are written in roman letters, while variables start with an uppercase letter. A *substitution* θ is a mapping from variables to expressions, and $\mathcal{D}\mathsf{om}(\theta)$ is its domain. Substitutions are usually denoted by (finite) sets of bindings like, e.g., $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$. The identity substitution is denoted by id. Composition of substitutions is denoted by juxtaposition, i.e., $\theta\theta'$ denotes a substitution θ'' such that $\theta''(X) = \theta'(\theta(X))$ for all $X \in Var$. Substitution application $\sigma(e)$ is also denoted by $e\sigma$.

In a case expression "case e of pat_1 when $e_1 \to e'_1$; $\ldots$; pat_n when $e_n \to e'_n$ end," we first evaluate e to a value, say v; then, we find (if it exists) the first clause pat_i when $e_i \to e'_i$ such that v matches pat_i, i.e., such that there exists a substitution σ for the variables of pat_i with $v = pat_i\sigma$, and $e_i\sigma$ (the *guard*) reduces to *true*; then, the case expression reduces to $e'_i\sigma$.

In our language, a running system is a pool of processes that can only interact through message sending and receiving (i.e., there is no shared memory). Received messages are stored in the queues of processes until they are consumed; namely, each process has one associated local (FIFO) queue. Each process is uniquely identified by its *pid* (process identifier). Message sending is asynchronous, while receive instructions block the execution of a process until an appropriate message reaches its local queue (see below).

In the paper, $\overline{o_n}$ denotes a sequence of syntactic objects $o_1, \ldots, o_n$.

[1] As in Erlang, the only data constructors in the language (besides literals) are the predefined functions for lists and tuples.

$$
\begin{array}{rl}
\mathrm{main}/0 = \mathsf{fun}\ () \to & \mathsf{let}\ S = \mathsf{spawn}(\mathrm{server}/0, [\,]) \\
& \mathsf{in\ let}\ P = \mathsf{spawn}(\mathrm{proxy}/0, [\,])\ \mathsf{in\ apply}\ \mathrm{client}/2\ (P, S) \\
\mathrm{server}/0 = \mathsf{fun}\ () \to & \mathsf{receive} \\
& \quad \{C, N\} \to \mathsf{receive} \\
& \qquad\qquad M \to \mathsf{let}\ X = C\ !\ \mathsf{call} + (N, M)\ \mathsf{in\ apply}\ \mathrm{server}/0\ () \\
& \qquad \mathsf{end}; \\
& \quad E \to \mathrm{error} \\
& \mathsf{end} \\
\mathrm{proxy}/0 = \mathsf{fun}\ () \to & \mathsf{receive}\ \{T, M\} \to \mathsf{let}\ W = T\ !\ M\ \mathsf{in\ apply}\ \mathrm{proxy}/0\ ()\ \mathsf{end} \\
\mathrm{client}/2 = \mathsf{fun}\ (P, S) \to & \mathsf{let}\ X = P\ !\ \{S, \{\mathsf{self}(), 40\}\}\ \mathsf{in\ let}\ Y = S\ !\ 2\ \mathsf{in\ receive}\ N \to N\ \mathsf{end}
\end{array}
$$

Figure 5.2: A simple client/server program

We consider the following functions with side-effects: self, "!", spawn, and receive. The expression self() returns the pid of a process, while $p\ !\ v$ sends a message v to the process with pid p, which will be eventually stored in p's local queue. New processes are spawned with a call of the form $\mathsf{spawn}(a/n, [\overline{v_n}])$, so that the new process begins with the evaluation of $\mathsf{apply}\ a/n\ (\overline{v_n})$. Finally, an expression "$\mathsf{receive}\ \overline{pat_n\ \mathsf{when}\ e_n \to e'_n}\ \mathsf{end}$" should find the *first* message v in the process' queue (if any) such that $\mathsf{case}\ v\ \mathsf{of}\ \overline{pat_n\ \mathsf{when}\ e_n \to e'_n}\ \mathsf{end}$ can be reduced to some expression e''; then, the receive expression evaluates to e'', with the side effect of deleting the message v from the process' queue. If there is no matching message, the process *suspends* until a matching message arrives.

Our language models a significant subset of Core Erlang [25], the intermediate representation used during the compilation of Erlang programs. Therefore, our developments can be directly applied to Erlang (as can be seen in the technical report [88], where the development of a practical debugger is described).

Example 5.1. The program in Figure 5.2 implements a simple client/server scheme with one server, one client and a proxy. The execution starts with a call to function main/0. It spawns the server and the proxy and finally calls function client/2. Both the server and the proxy then suspend waiting for messages. The client makes two requests $\{C, 40\}$ and 2, where C is the pid of client (obtained using self()). The second request goes directly to the server, but the first one is sent through the proxy (which simply resends the received messages), so the client actually sends $\{S, \{C, 40\}\}$, where S is the pid of the server. Here, we expect that the server first receives the message $\{C, 40\}$ and, then, 2, thus sending back 42 to the client C (and calling function $\mathrm{server}/0$ again in an endless recursion). If the first message does

not have the right structure, the catch-all clause "$E \rightarrow \mathrm{error}$" returns error and stops.

A High-Level Semantics.

Now, we present an (asynchronous) operational semantics for our language. Following [130], we introduce a *global mailbox* (there called "ether") to guarantee that our semantics generates all admissible message interleavings. In contrast to previous semantics [86, 109, 130], our semantics abstracts away from processes' queues. We will see in Section 5.2 that this decision simplifies both the semantics and the notion of independence, while still modeling the same potential computations (see the technical report [88]).

Definition 5.2 (process)**.** A process is a configuration $\langle p, \theta, e\rangle$, where p is its pid, θ an environment (a substitution of values for variables), and e an expression.

In order to define a *system* (roughly, a pool of processes interacting through message exchange), we first need the notion of global mailbox.

Definition 5.3 (global mailbox)**.** We define a global mailbox, Γ, as a multiset of triples of the form $(sender\_pid, target\_pid, message)$. Given a global mailbox Γ, we let $\Gamma \cup \{(p, p', v)\}$ denote a new mailbox also including the triple (p, p', v), where we use "$\cup$" as multiset union.

In Erlang, the order of two messages sent directly from process p to process p' is kept if both are delivered; see [41, Section 10.8].[2] To enforce such a constraint, we could define a global mailbox as a collection of FIFO queues, one for each sender-receiver pair. In this work, however, we keep Γ a multiset. This solution is both simpler and more general since FIFO queues serve only to select those computations satisfying the constraint. Nevertheless, if our logging approach is applied to a computation satisfying the above constraint, then our replay computation will also satisfy it, thus replay does not introduce spurious computations.

Definition 5.4 (system)**.** A system is a pair $\Gamma; \Pi$, where Γ is a global mailbox and Π is a pool of processes, denoted as $\langle p_1, \theta_1, e_1\rangle \mid \cdots \mid \langle p_n, \theta_n, e_n\rangle$; here "$\mid$" represents an associative and commutative operator. We often denote a system as $\Gamma; \langle p, \theta, e\rangle \,|\, \Pi$ to point out that $\langle p, \theta, e\rangle$ is an arbitrary process of the pool.

A system is *initial* if it has the form $\{\,\}; \langle p, id, e\rangle$, where $\{\,\}$ is an empty global mailbox, p is a pid, id is the identity substitution, and e is an expression.

[2] Current implementations only guarantee this restriction within the same node.

$$(Seq) \quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{seq}} \Gamma; \langle p, \theta', e'\rangle \mid \Pi}$$

$$(Send) \quad \frac{\theta, e \xrightarrow{\mathsf{send}(p',v)} \theta', e' \text{ and } \ell \text{ is a fresh symbol}}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{send}(\ell)} \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \theta', e'\rangle \mid \Pi}$$

$$(Receive) \quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \text{ and } \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{rec}(\ell)} \Gamma; \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid \Pi}$$

$$(Spawn) \quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \text{ and } p' \text{ is a fresh pid}}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{spawn}(p')} \Gamma; \langle p, \theta', e'\{\kappa \mapsto p'\}\rangle \mid \langle p', id, \mathsf{apply}\ a/n\ (\overline{v_n})\rangle \mid \Pi}$$

$$(Self) \quad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{self}} \Gamma; \langle p, \theta', e'\{\kappa \mapsto p\}\rangle \mid \Pi}$$

Figure 5.3: Logging semantics

Following the style in [109], the semantics of the language is defined in a modular way, so that the labeled transition relations $\rightarrow$ and $\hookrightarrow$ model the evaluation of *expressions* and the reduction of *systems*, respectively. Given an environment θ and an expression e, we denote by $\theta, e \xrightarrow{l} \theta', e'$ a one-step reduction labeled with l. The relation $\xrightarrow{l}$ follows a typical call-by-value semantics for side-effect free expressions; for expressions with side-effects, we label the reduction with the information needed to perform the side-effects within the system rules of Figure 5.3. We refer to the rules of Figure 5.3 as the *logging* semantics, since the relation is labeled with some basic information used to log the steps of a computation (see Section 5.3). For now, the reader can safely ignore these labels (actually, labels will be omitted when irrelevant). The topics of this work are orthogonal to the evaluation of expressions, thus we refer the reader to [88] for the formalization of the rules of $\xrightarrow{l}$. Let us now briefly describe the interaction between the reduction of expressions and the rules of the logging semantics:

- A one-step reduction of an expression without side-effects is labeled with τ. In this case, rule *Seq* in Fig. 5.3 is applied to update correspondingly the environment and expression of the considered process.
- An expression $p' \,!\, v$ is reduced to v, with label $\mathsf{send}(p', v)$, so that rule *Send* in Fig. 5.3 can add the triple $(p, p', \{v, \ell\})$ to Γ (p is the process performing the send). The message is *tagged* with some fresh (unique) identifier ℓ. These tags

allow us to track messages and avoid confusion when several messages have the same value (these tags are similar to the timestamps used in [106]).

- The remaining functions, receive, spawn and self, pose an additional problem: their value cannot be computed locally. Therefore, they are reduced to a fresh distinguished symbol κ, which is then replaced by the appropriate value in the system rules. In particular, a receive statement receive $\overline{cl_n}$ end is reduced to κ with label $\mathsf{rec}(\kappa, \overline{cl_n})$. Then, rule *Receive* in Fig. 5.3 nondeterministically checks if there exists a triple $(p', p, \{v, \ell\})$ in the global mailbox that matches some clause in $\overline{cl_n}$; pattern matching is performed by the auxiliary function matchrec. If the matching succeeds, it returns the pair (θ_i, e_i) with the matching substitution θ_i and the expression in the selected branch e_i. Finally, κ is bound to the expression e_i within the derived expression e'.

- For a spawn, an expression $\mathsf{spawn}(a/n, [\overline{v_n}])$ is also reduced to κ with label $\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])$. Rule *Spawn* in Fig. 5.3 then adds a new process with a fresh pid p' initialized with an empty environment id and the application apply $a/n\ (v_1, \ldots, v_n)$. Here, κ is bound to p', the pid of the spawned process.

- Finally, the expression $\mathsf{self}()$ is reduced to κ with label $\mathsf{self}(\kappa)$ so that rule *Self* in Fig. 5.3 can bind κ to the pid of the given process.

We often refer to reduction steps derived by the system rules as *actions* taken by the chosen process.

Example 5.5. Let us consider the program of Example 5.1 and the initial system $\{\ \};\ \langle \mathrm{c}, id, \text{apply } \mathrm{main}/0\ ()\rangle$, where c is the pid of the process. A possible (faulty) computation from this system is shown in Fig. 5.4 (the selected expression at each step is underlined).[3] Here, we ignore the labels of the relation $\hookrightarrow$. Moreover, we skip the steps that just bind variables and we do not show the bindings of variables but substitute them for their values for clarity.

Independence.

In order to define a causal-consistent replay semantics we need not only an interleaving semantics such as the one we just presented, but also a notion of causality or, equivalently, the opposite notion of independence. To this end, we use the labels

[3] Roughly speaking, the problem comes from the fact that the messages reach the server in the wrong order. Note that this faulty derivation is possible even by considering Erlang's policy on the order of messages, since they follow a different path.

$$
\begin{aligned}
&\{\,\};\ \langle \mathrm{c}, \_, \underline{\mathsf{apply}\ \mathrm{main}/0\ ()}\rangle \\
\hookrightarrow\ &\{\,\};\ \langle \mathrm{c}, \_, \underline{\mathsf{let}\ S = \mathsf{spawn}(\mathrm{server}/0, [\,])}\ \mathsf{in}\ \ldots\rangle \\
\hookrightarrow\ &\{\,\};\ \langle \mathrm{c}, \_, \mathsf{let}\ P = \underline{\mathsf{spawn}(\mathrm{proxy}/0, [\,])}\ \mathsf{in}\ \mathsf{apply}\ \mathrm{client}/2\ (P, \mathrm{s})\rangle \mid \langle \mathrm{s}, \_, \mathsf{apply}\ \mathrm{server}/0\ ()\rangle \\
\hookrightarrow\ &\{\,\};\ \langle \mathrm{c}, \_, \underline{\mathsf{apply}\ \mathrm{client}/2\ (\mathrm{p}, \mathrm{s})}\rangle \mid \langle \mathrm{s}, \_, \mathsf{apply}\ \mathrm{server}/0\ ()\rangle \mid \langle \mathrm{p}, \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\hookrightarrow\ &\{\,\};\ \langle \mathrm{c}, \_, \mathsf{let}\ X = \mathrm{p}\,!\,\{\mathrm{s}, \{\underline{\mathsf{self}()}, 40\}\}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{apply}\ \mathrm{server}/0\ ()\rangle \mid \langle \mathrm{p}, \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\hookrightarrow\ &\{\,\};\ \langle \mathrm{c}, \_, \mathsf{let}\ X = \mathrm{p}\,!\,\{\mathrm{s}, \{\mathrm{c}, 40\}\}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \underline{\mathsf{apply}\ \mathrm{server}/0\ ()}\rangle \mid \langle \mathrm{p}, \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\hookrightarrow\ &\{\,\};\ \langle \mathrm{c}, \_, \mathsf{let}\ X = \mathrm{p}\,!\,\{\mathrm{s}, \{\mathrm{c}, 40\}\}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, \_, \underline{\mathsf{apply}\ \mathrm{proxy}/0\ ()}\rangle \\
\hookrightarrow\ &\{\,\};\ \langle \mathrm{c}, \_, \mathsf{let}\ X = \underline{\mathrm{p}\,!\,\{\mathrm{s}, \{\mathrm{c}, 40\}\}}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, \_, \mathsf{receive}\ \ldots\rangle \\
\hookrightarrow\ &\{(\mathrm{c}, \mathrm{p}, \{\{\mathrm{s}, \{\mathrm{c}, 40\}\}, \ell_1\})\};\ \langle \mathrm{c}, \_, \mathsf{let}\ Y = \underline{\mathrm{s}\,!\,2}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, \_, \mathsf{receive}\ \ldots\rangle \\
\hookrightarrow\ &\{(\mathrm{c}, \mathrm{p}, \{\{\mathrm{s}, \{\mathrm{c}, 40\}\}, \ell_1\}), (\mathrm{c}, \mathrm{s}, \{2, \ell_2\})\};\ \langle \mathrm{c}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, \_, \underline{\mathsf{receive}\ \ldots}\rangle \\
\hookrightarrow\ &\{(\mathrm{c}, \mathrm{s}, \{2, \ell_2\})\};\ \langle \mathrm{c}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, \_, \underline{\mathsf{let}\ W = \mathrm{s}\,!\,\{\mathrm{c}, 40\}\ \mathsf{in}\ \ldots}\rangle \\
\hookrightarrow\ &\{(\mathrm{c}, \mathrm{s}, \{2, \ell_2\}), (\mathrm{p}, \mathrm{s}, \{\{\mathrm{c}, 40\}, \ell_3\})\};\ \langle \mathrm{c}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \underline{\mathsf{receive}\ \ldots}\rangle \mid \langle \mathrm{p}, \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\hookrightarrow\ &\{(\mathrm{p}, \mathrm{s}, \{\{\mathrm{c}, 40\}, \ell_3\})\};\ \langle \mathrm{c}, \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, \_, \mathrm{error}\rangle \mid \langle \mathrm{p}, \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle
\end{aligned}
$$

Figure 5.4: Faulty derivation with the client/server of Example 5.1

of the logging semantics (see Figure 5.3). These labels include the pid p of the process that performs the transition, the rule used to derive it and, in some cases, some additional information: a message tag ℓ in rules *Send* and *Receive*, and the pid p' of the spawned process in rule *Spawn*.

Before formalizing the notion of independence, we need to introduce some notation and terminology. Given systems s_0, s_n, we call $s_0 \hookrightarrow^* s_n$, which is a shorthand for $s_0 \hookrightarrow_{p_1,r_1} \ldots \hookrightarrow_{p_n,r_n} s_n$, $n \geq 0$, a *derivation*. One-step derivations are simply called *transitions*. We use $d, d', d_1, \ldots$ to denote derivations and $t, t', t_1, \ldots$ for transitions. Given a derivation $d = (s_1 \hookrightarrow^* s_2)$, we define $\mathsf{init}(d) = s_1$. Two derivations, d_1 and d_2, are said *coinitial* if $\mathsf{init}(d_1) = \mathsf{init}(d_2)$.

For simplicity, in the following, we consider derivations up to renaming of bound variables. Under this assumption, the semantics is *almost* deterministic, i.e., the main sources of non-determinism are the selection of a process p and of the message to be retrieved by p in rule $Receive$. Choices of the fresh identifier ℓ for messages and of the pid p' of new processes are also non-deterministic. Note that each process can perform at most one transition for each label, i.e., $s \hookrightarrow_{p,r} s_1$ and $s \hookrightarrow_{p,r} s_2$ trivially implies $s_1 = s_2$.

We now instantiate to our setting the well-known *happened-before* relation [78], and the related notion of *independent* transitions:[4]

[4]Here, we use the term *independent*, instead of *concurrent* as in [78], since the latter has a slightly different meaning in the literature of causal-consistency.

Definition 5.6 (happened-before, independence)**.** Given transitions $t_1 = (s_1 \hookrightarrow_{p_1,r_1} s_1')$ and $t_2 = (s_2 \hookrightarrow_{p_2,r_2} s_2')$, we say that t_1 happened before t_2, in symbols $t_1 \rightsquigarrow t_2$, if one of the following conditions holds:

- they consider the same process, i.e., $p_1 = p_2$, and t_1 comes before t_2;
- t_1 spawns a process p, i.e., $r_1 = \mathsf{spawn}(p)$, and t_2 is performed by process p, i.e., $p_2 = p$;
- t_1 sends a message ℓ, i.e., $r_1 = \mathsf{send}(\ell)$, and t_2 receives the same message ℓ, i.e., $r_2 = \mathsf{rec}(\ell)$.

Furthermore, if $t_1 \rightsquigarrow t_2$ and $t_2 \rightsquigarrow t_3$, then $t_1 \rightsquigarrow t_3$ (transitivity). Two transitions t_1 and t_2 are *independent* if $t_1 \not\rightsquigarrow t_2$ and $t_2 \not\rightsquigarrow t_1$.

Switching consecutive independent transitions does not change the final state:

Lemma 5.7 (switching lemma)**.** *Let $t_1 = (s_1 \hookrightarrow_{p_1,r_1} s_2)$ and $t_2 = (s_2 \hookrightarrow_{p_2,r_2} s_3)$ be consecutive independent transitions. Then, there are two consecutive transitions $t_{2\langle\langle t_1} = (s_1 \hookrightarrow_{p_2,r_2} s_4)$ and $t_{1\rangle\rangle t_2} = (s_4 \hookrightarrow_{p_1,r_1} s_3)$ for some system s_4.*

The happened-before relation gives rise to an equivalence relation equating all derivations that only differ in the switch of independent transitions. Formally,

Definition 5.8 (causally equivalent derivations)**.** Let d_1 and d_2 be derivations under the logging semantics. We say that d_1 and d_2 are *causally equivalent*, in symbols $d_1 \approx d_2$, if d_1 can be obtained from d_2 by a finite number of switches of pairs of consecutive independent transitions.

Causal equivalence is an instance of the *trace equivalence* in [99].

5.3 Logging Computations.

In this section, we introduce a notion of *log* for a computation. Basically, we aim to analyze in a debugger a faulty behavior that occurs in some execution of a program. To this end, we need to extract from an actual execution enough information to replay it inside the debugger. Actually, we do not want to replay necessarily the exact same execution, but a causally equivalent one. In this way, the programmer can focus on some actions of a particular process, and actions of other processes are only performed if needed (formally, if they happened-before these actions). As we

will see in the next section, this ensures that the considered misbehaviors will still be replayed.

In a practical implementation (see the technical report [88]), one should instrument the program so that its execution in the actual environment produces a collection of sequences of logged events (one sequence per process). In the following, though, we exploit the logging semantics and, in particular, part of the information provided by the labels. The two approaches are equivalent, but the chosen one allows us to formally prove a number of properties in a simpler way.

One could argue (as in, e.g., [106]) that logs should only store information about the receive events, since this is the only nondeterministic action (once a process is selected). However, this is not enough in our setting, where:

- We need to log the sending of a message since this is where messages are tagged, and we need to know its (unique) identifier to be able to relate the sending and receiving of each message.

- We also need to log the spawn events, since the generated pids are needed to relate an action to the process that performed it (spawn events are not considered in [106] and, thus, their set of processes is fixed).

We note that other nondeterministic events, such as input from the user or from external services, should also be logged in order to correctly replay executions involving them. One can deal with them by instrumenting the corresponding primitives to log the input values, and then use these values when replaying the execution. Essentially, they can be dealt with as the receive primitive. Hence, we do not present them in detail to keep the presentation as simple as possible.

In the following, (ordered) sequences are denoted by $w = (r_1, r_2, \ldots, r_n)$, $n \geq 1$, where $()$ denotes the empty sequence. Concatenation is denoted by $+$. We write $r{+}w$ instead of $(r){+}w$ for simplicity.

Definition 5.9 (log)**.** A *log* is a (finite) sequence of events $(r_1, r_2, \ldots)$ where each r_i is either $\mathsf{spawn}(p)$, $\mathsf{send}(\ell)$ or $\mathsf{rec}(\ell)$, with p a pid and ℓ a message identifier. Logs are ranged over by ω. Given a derivation $d = (s_0 \hookrightarrow_{p_1,r_1} s_1 \hookrightarrow_{p_2,r_2} \ldots \hookrightarrow_{p_n,r_n} s_n)$, $n \geq 0$, under the logging semantics, the *log of a pid p in d*, in symbols $\mathcal{L}(d, p)$, is inductively defined as follows:

$$\mathcal{L}(d,p) = \begin{cases} () & \text{if } n = 0 \text{ or } p \text{ does not occur in } d \\ r_1{+}\mathcal{L}(s_1 \hookrightarrow^* s_n, p) & \text{if } n > 0,\ p_1 = p, \text{ and } r_1 \notin \{\mathsf{seq}, \mathsf{self}\} \\ \mathcal{L}(s_1 \hookrightarrow^* s_n, p) & \text{otherwise} \end{cases}$$

The *log of* d, written $\mathcal{L}(d)$, is defined as: $\mathcal{L}(d) = \{(p, \mathcal{L}(d,p)) \mid p \text{ occurs in } d\}$. We sometimes call $\mathcal{L}(d)$ the *global* log of d to avoid confusion with $\mathcal{L}(d,p)$. Note that $\mathcal{L}(d,p) = \omega$ if $(p,\omega) \in \mathcal{L}(d)$ and $\mathcal{L}(d,p) = ()$ otherwise.

Example 5.10. Consider the derivation shown in Example 5.5, here referred to as d. If we run it under the logging semantics, we get the following logs:

$$\begin{array}{lcllcl} \mathcal{L}(d,\mathrm{c}) & = & (\mathsf{spawn}(\mathrm{s}), \mathsf{spawn}(\mathrm{p}), \mathsf{send}(\ell_1), \mathsf{send}(\ell_2)) & & & \\ \mathcal{L}(d,\mathrm{s}) & = & (\mathsf{rec}(\ell_2)) & \mathcal{L}(d,\mathrm{p}) & = & (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)) \end{array}$$

In the following we only consider finite derivations under the logging semantics. This is reasonable in our context where the programmer wants to analyze in the debugger a finite (possibly incomplete) execution showing a faulty behavior.

An essential property of our semantics is that causally equivalent derivations have the same log, i.e., the log depends only on the equivalence class, not on the selection of the representative inside the class. The reverse implication, namely that (coinitial) derivations with the same global log are causally equivalent, holds provided that we establish the following convention on when to stop a derivation:

Definition 5.11 (fully-logged derivation)**.** A derivation d is *fully-logged* if, for each process p, its last transition $s_1 \hookrightarrow_{p,r} s_2$ in d (if any) is a *logged* transition, i.e., $r \notin \{\mathsf{seq}, \mathsf{self}\}$. In particular, if a process performs no logged transition, then it performs no transition at all.

Restricting to fully-logged derivations is needed since only logged transitions contribute to logs. Otherwise, two derivations d_1 and d_2 could produce the same log, but differ simply because, e.g., d_1 performs more non-logged transitions than d_2. Restricting to fully-logged derivations, we include the minimal amount of transitions needed to produce the observed log.

Finally, we present a key result of our logging semantics. It states that two derivations are causally equivalent iff they produce the same log.

Theorem 5.12. *Let d_1, d_2 be coinitial fully-logged derivations. $\mathcal{L}(d_1) = \mathcal{L}(d_2)$ iff $d_1 \approx d_2$.*

5.4 A Causal-Consistent Replay Semantics

In this section, we introduce an *uncontrolled* replay semantics. It takes a program and the log of a given derivation, and allows us to replay any causally equivalent derivation. This semantics constitutes the kernel of our replay framework. The

$$(Seq)\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, \omega, \theta, e\rangle \mid \Pi \rightharpoonup_{p,\mathsf{seq},\{\mathsf{s}\}} \Gamma; \langle p, \omega, \theta', e'\rangle \mid \Pi}$$

$$(Send)\quad \frac{\theta, e \xrightarrow{\mathsf{send}(p',v)} \theta', e'}{\Gamma; \langle p, \mathsf{send}(\ell){+}\omega, \theta, e\rangle \mid \Pi \rightharpoonup_{p,\mathsf{send}(\ell),\{\mathsf{s},\ell^{\Uparrow}\}} \Gamma \cup \{(p,p',\{v,\ell\})\}; \langle p, \omega, \theta', e'\rangle \mid \Pi}$$

$$(Receive)\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta', e' \;\text{ and }\; \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\begin{array}{l}\Gamma \cup \{(p',p,\{v,\ell\})\}\langle p, \mathsf{rec}(\ell){+}\omega, \theta, e\rangle \mid \Pi \\ \qquad \rightharpoonup_{p,\mathsf{rec}(\ell),\{\mathsf{s},\ell^{\Downarrow}\}} \Gamma; \langle p, \omega, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid \Pi\end{array}}$$

$$(Spawn)\quad \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa,a/n,[\overline{v_n}])} \theta', e' \quad \text{and} \quad \omega' = \mathcal{L}(d,p')}{\begin{array}{l}\Gamma; \langle p, \mathsf{spawn}(p'){+}\omega, \theta, e\rangle \mid \Pi \rightharpoonup_{p,\mathsf{spawn}(p'),\{\mathsf{s},\mathsf{sp}_{p'}\}} \Gamma; \langle p, \omega, \theta', e'\{\kappa \mapsto p'\}\rangle \\ \qquad \mid \langle p', \omega', id, \mathsf{apply}\ a/n\ (\overline{v_n})\rangle \mid \Pi\end{array}}$$

$$(Self)\quad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, \omega, \theta, e\rangle \mid \Pi \rightharpoonup_{p,\mathsf{self},\{\mathsf{s}\}} \Gamma; \langle p, \omega, \theta', e'\{\kappa \mapsto p\}\rangle \mid \Pi}$$

Figure 5.5: Uncontrolled replay semantics

term uncontrolled indicates that the semantics specifies how to perform replay, but there is no policy to select the applicable rule when more than one is enabled. The uncontrolled semantics is suitable to set the basis of our replay mechanism, but does not allow one to focus on the causes of a given action. For this reason, in Section 5.5, we build on top of this semantics a *controlled* one, where the selection of actions is driven by the queries from the user.

In the following, we introduce a transition relation $\rightharpoonup$ to specify replay. Transition $\rightharpoonup$ is similar to the logging semantics $\hookrightarrow$ (Figure 5.3) but it is now driven by the considered log. Thus, processes have the form $\langle p, \omega, \theta, e\rangle$, with ω a log.

The uncontrolled causal-consistent replay semantics is shown in Figure 5.5. For technical reasons, labels of the replay semantics contain the same information as the labels of the logging semantics. Moreover, the labels now also include a set of replay *requests*. The reader can ignore these elements until the next section. For simplicity, we also consider that the log $\mathcal{L}(d,p)$ of each process p in the original derivation d is a fixed global parameter of the transition rules (see rule *Spawn*).

The rules for expressions are the same as in the logging semantics (an advantage of the modular design). The replay semantics is similar to the logging semantics, except that logs fix some parameters: the fresh message identifier in rule *Send*, the message received in rule *Receive*, and the fresh pid in rule *Spawn*.

Example 5.13. Consider the logs of Example 5.10. Then, we have, e.g., the replay

$$
\begin{array}{l}
\quad \{\,\};\ \langle \mathrm{c}, (\mathsf{spawn}(\mathrm{s}), \mathsf{spawn}(\mathrm{p}), \mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), \_, \underline{\mathsf{apply}\ \mathrm{main}/0\ ()}\rangle \\
\rightharpoonup \{\,\};\ \langle \mathrm{c}, (\mathsf{spawn}(\mathrm{s}), \mathsf{spawn}(\mathrm{p}), \mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), \_, \mathsf{let}\ S = \underline{\mathsf{spawn}(\mathrm{server}/0, [\,])}\ \mathsf{in}\ \ldots\rangle \\
\rightharpoonup \{\,\};\ \langle \mathrm{c}, (\mathsf{spawn}(\mathrm{p}), \mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), \_, \mathsf{let}\ P = \mathsf{spawn}(\mathrm{proxy}/0, [\,])\ \mathsf{in} \\
\qquad \mathsf{apply}\ \mathrm{client}/2\ (P, \mathrm{s})\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), \_, \underline{\mathsf{apply}\ \mathrm{server}/0\ ()}\rangle \\
\rightharpoonup \{\,\};\ \langle \mathrm{c}, (\mathsf{spawn}(\mathrm{p}), \mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), \_, \mathsf{let}\ P = \underline{\mathsf{spawn}(\mathrm{proxy}/0, [\,])}\ \mathsf{in} \\
\qquad \mathsf{apply}\ \mathrm{client}/2\ (P, \mathrm{s})\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), \_, \mathsf{receive}\ \ldots\rangle \\
\rightharpoonup \{\,\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), \_, \underline{\mathsf{apply}\ \mathrm{client}/2\ (\mathrm{p}, \mathrm{s})}\rangle \\
\qquad \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)), \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\rightharpoonup \{\,\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), \_, \mathsf{let}\ X = \mathrm{p}\ !\ \{\mathrm{s}, \{\underline{\mathsf{self}()}, 40\}\}\ \mathsf{in}\ \ldots\rangle \\
\qquad \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)), \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\rightharpoonup \{\,\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_1), \mathsf{send}(\ell_2)), \_, \mathsf{let}\ X = \mathrm{p}\ !\ \{\mathrm{s}, \{\mathrm{c}, 40\}\}\ \mathsf{in}\ \ldots\rangle \\
\qquad \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{p}, (\underline{\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)}), \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\rightharpoonup \{(\mathrm{c}, \mathrm{p}, \{\{\mathrm{s}, \{\mathrm{c}, 40\}\}, \ell_1\})\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_2)), \_, \mathsf{let}\ Y = \mathrm{s}\ !\ 2\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), \_, \mathsf{receive}\ \ldots\rangle \\
\qquad \mid \langle \mathrm{p}, (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)), \_, \underline{\mathsf{apply}\ \mathrm{proxy}/0\ ()}\rangle \\
\rightharpoonup \{(\mathrm{c}, \mathrm{p}, \{\{\mathrm{s}, \{\mathrm{c}, 40\}\}, \ell_1\})\};\ \langle \mathrm{c}, (\underline{\mathsf{send}(\ell_2)}), \_, \mathsf{let}\ Y = \mathrm{s}\ !\ 2\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), \_, \mathsf{receive}\ \ldots\rangle \\
\qquad \mid \langle \mathrm{p}, (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)), \_, \underline{\mathsf{receive}\ \ldots}\rangle \\
\rightharpoonup \{\,\};\ \langle \mathrm{c}, (\mathsf{send}(\ell_2)), \_, \mathsf{let}\ Y = \mathrm{s}\ !\ 2\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), \_, \mathsf{receive}\ \ldots\rangle \\
\qquad \mid \langle \mathrm{p}, (\mathsf{send}(\ell_3)), \_, \mathsf{let}\ \mathrm{s}\ !\ \{\mathrm{c}, 40\}\ \mathsf{in}\ \ldots\rangle \\
\rightharpoonup \{(\mathrm{p}, \mathrm{s}, \{\{\mathrm{c}, 40\}, \ell_3\})\};\ \langle \mathrm{c}, (\underline{\mathsf{send}(\ell_2)}), \_, \mathsf{let}\ Y = \underline{\mathrm{s}\ !\ 2}\ \mathsf{in}\ \ldots\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), \_, \mathsf{receive}\ \ldots\rangle \\
\qquad \mid \langle \mathrm{p}, (), \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\rightharpoonup \{(\mathrm{p}, \mathrm{s}, \{\{\mathrm{c}, 40\}, \ell_3\}), (\mathrm{c}, \mathrm{s}, \{2, \ell_2\})\};\ \langle \mathrm{c}, (), \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, (\mathsf{rec}(\ell_2)), \_, \underline{\mathsf{receive}\ \ldots}\rangle \\
\qquad \mid \langle \mathrm{p}, (), \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle \\
\rightharpoonup \{(\mathrm{p}, \mathrm{s}, \{\{\mathrm{c}, 40\}, \ell_3\})\};\ \langle \mathrm{c}, (), \_, \mathsf{receive}\ \ldots\rangle \mid \langle \mathrm{s}, (), \_, \mathrm{error}\rangle \mid \langle \mathrm{p}, (), \_, \mathsf{apply}\ \mathrm{proxy}/0\ ()\rangle
\end{array}
$$

Figure 5.6: Uncontrolled replay derivation with the traces of Example 5.10

derivation in Fig. 5.6. The actions performed by each process are the same as in the original derivation in Example 5.5, but the interleavings are slightly different. Moreover, after ten steps, the server is waiting for a message, the global mailbox contains a matching message but, in contrast to the logging semantics, receive cannot proceed since the message identifier in the log does not match (ℓ_2 vs ℓ_3).

Basic Properties of the Replay Semantics.

Here, we show that the uncontrolled replay semantics is consistent and we relate it with the logging semantics. We need the following auxiliary functions:

Definition 5.14. Let $d = (s_1 \hookrightarrow^* s_2)$ be a derivation under the logging semantics, with $s_1 = \Gamma; \langle p_1, \theta_1, e_1\rangle \mid \ldots \mid \langle p_n, \theta_n, e_n\rangle$. The system corresponding to s_1 in the

replay semantics is defined as follows:

$$addLog(\mathcal{L}(d), s_1) = \Gamma; \langle p_1, \mathcal{L}(d, p_1), \theta_1, e_1\rangle \mid \ldots \mid \langle p_n, \mathcal{L}(d, p_n), \theta_n, e_n\rangle$$

Conversely, given a system $s = \Gamma; \langle p_1, \omega_1, \theta_1, e_1\rangle \mid \ldots \mid \langle p_n, \omega_n, \theta_n, e_n\rangle$ in the replay semantics, we let $del(s)$ be the system obtained from s by removing logs, i.e., $del(s) = \Gamma; \langle p_1, \theta_1, e_1\rangle \mid \ldots \mid \langle p_n, \theta_n, e_n\rangle$, and similarly for derivations.

In the following, we extend the notions of log and coinitial derivations, as well as function init, to replay derivations in the obvious way. Furthermore, we now call a system s' *initial* under the replay semantics if there exists a derivation d under the logging semantics, and $s' = addLog(\mathcal{L}(d), \mathsf{init}(d))$.

We extend the notion of fully-logged derivations to our replay semantics:

Definition 5.15 (fully-logged replay derivation)**.** A derivation d under the replay semantics is *fully-logged* if, for each process p, the log is empty and its last transition (if any) is a logged transition.

Note that, in addition to Definition 5.11, we now require that processes *consume* all their logs.

We will only consider systems reachable from the execution of a program:

Definition 5.16 (reachable systems)**.** A system s is *reachable* if there exists an initial system s_0 such that $s_0 \rightharpoonup^* s$.

Since only reachable systems are of interest (non-reachable systems are ill-formed), in the following we assume that all systems are reachable.

Now, we can tackle the problem of proving that our replay semantics preserves causal equivalence, i.e., that the original and the replay derivations are always causally equivalent.

Theorem 5.17. *Let d be a fully-logged derivation under the logging semantics. Let d' be any finite fully-logged derivation under the replay semantics such that* $\mathsf{init}(d') = addLog(\mathcal{L}(d), \mathsf{init}(d))$. *Then $d \approx del(d')$.*

Usefulness for Debugging.

Now, we show that our replay semantics is indeed useful as a basis for designing a debugging tool. In particular, we prove that a (faulty) behavior occurs in the logged derivation iff any replay derivation also exhibits the same *faulty* behavior, hence replay is correct and complete.

In order to formalize such a result we need to fix the notion of faulty behavior we are interested in. For us, a misbehavior is a wrong system, but since the system is possibly distributed, we concentrate on misbehaviors visible from a "local" observer. Given that our systems are composed of processes and messages in the global mailbox, we consider that a (local) misbehavior is either a wrong message in the global mailbox or a process with a wrong configuration.

Theorem 5.18 (Correctness and completeness)**.** *Let d be a fully-logged derivation under the logging semantics. Let d' be any fully-logged derivation under the uncontrolled replay semantics such that* $\mathsf{init}(d') = addLog(\mathcal{L}(d), \mathsf{init}(d))$*. Then:*

1. *there is a system $\Gamma; \Pi$ in d with a configuration $\langle p, \theta, e\rangle$ in Π iff there is a system $\Gamma'; \Pi'$ in d' with a configuration $\langle p, \theta, e\rangle$ in $del(\Gamma'; \Pi')$;*

2. *there is a system $\Gamma; \Pi$ in d with a message $(p, p', \{v, \ell\})$ in Γ iff there is a system $\Gamma'; \Pi'$ in d' with a message $(p, p', \{v, \ell\})$ in Γ'.*

The result above is very strong: it ensures that a misbehavior occurring in a logged execution is replayed in *any* possible fully-logged derivation. This means that any scheduling policy is fine for replay. Furthermore, this remains true whatever actions the user takes: either the misbehavior is reached, or it remains in any possible forward computation.

One may wonder whether more general notions of misbehavior make sense. Above, we consider just "local" observations. One could ask for more than one local observation to be replayed. By applying the result above to multiple observations we get that all of them will be replayed, but, if they concern different processes or messages, we cannot ensure that they are replayed *at the same time or in the same order*. For instance, in the derivation of Figure 5.4, process c sends the message with identifier ℓ_2 before process p receives the message with identifier ℓ_1, while in the replay derivation of Figure 5.6 the two actions are executed in the opposite order. Only a *super user* able to see the whole system at once could see such a (mis)behavior, which are thus not relevant in our context.

5.5 Controlled Replay Semantics

In this section, we introduce a controlled version of the replay semantics. The semantics in the previous section allows one to replay a given derivation and be guaranteed to replay, sooner or later, any local misbehavior. In practice, though, one normally knows in which process p the misbehavior appears, and thus (s)he wants

$$\frac{\Gamma;\Pi \rightharpoonup_{p,r,\Psi'} \Gamma';\Pi' \quad \wedge \quad \psi \in \Psi'}{[\![\Gamma;\Pi]\!]_{\{p,\psi\}+\Psi} \rightsquigarrow [\![\Gamma';\Pi']\!]_{\Psi}} \qquad \frac{\Gamma;\Pi \rightharpoonup_{p,r,\Psi'} \Gamma';\Pi' \quad \wedge \quad \psi \notin \Psi'}{[\![\Gamma;\Pi]\!]_{\{p,\psi\}+\Psi} \rightsquigarrow [\![\Gamma';\Pi']\!]_{\{p,\psi\}+\Psi}}$$

$$\frac{\Gamma;\langle p,\mathsf{rec}(\ell)+\omega,\theta,e\rangle \mid \Pi \not\rightharpoonup_{p,r,\Psi'} \quad \wedge \quad sender(\ell)=p'}{[\![\Gamma;\langle p,\mathsf{rec}(\ell)+\omega,\theta,e\rangle \mid \Pi]\!]_{\{p,\psi\}+\Psi} \rightsquigarrow [\![\Gamma;\langle p,\mathsf{rec}(\ell)+\omega,\theta,e\rangle \mid \Pi]\!]_{(\{p',\ell^{\Uparrow}\},\{p,\psi\})+\Psi}}$$

$$\frac{\nexists p \text{ in } \Pi \quad \wedge \quad parent(p)=p'}{[\![\Gamma;\Pi]\!]_{\{p,\psi\}+\Psi} \rightsquigarrow [\![\Gamma;\Pi]\!]_{(\{p',\mathsf{sp}_p\},\{p,\psi\})+\Psi}}$$

Figure 5.7: Controlled replay semantics

to focus on a process p or even on some of its actions. However, to correctly replay these actions, one also needs to replay the actions that happened before them. We present in Figure 5.7 a semantics where the user can specify which actions (s)he wants to replay, and the semantics takes care of replaying them. Replaying an action requires to replay all *and only* its causes. Notably, the bug causing a misbehavior causes the action showing the misbehavior.

Here, given a system s, we want to start a replay until a particular action ψ is performed on a given process p. We denote such a replay request with $[\![s]\!]_{(\{p,\psi\})}$. In general, the subscript of $[\![\ ]\!]$ is a stack of requests, where the first element is the most recent one. In this paper, we consider the following replay requests:

- $\{p, \mathsf{s}\}$: one step of process p (the extension to n steps is straightforward);
- $\{p, \ell^{\Uparrow}\}$: request for process p to send the message tagged with ℓ;
- $\{p, \ell^{\Downarrow}\}$: request for process p to receive the message tagged with ℓ;
- $\{p, \mathsf{sp}_{p'}\}$: request for process p to spawn the process p'.

Variable creations as not valid targets for replay requests, since variable names are not known before their creation (variable creations are not logged). The requests above are *satisfied* when a corresponding uncontrolled transition is performed. Indeed, the third element labeling the relations of the replay semantics in Figure 5.5 is the set of requests satisfied in the corresponding step.

Let us explain the rules of the controlled replay semantics in Fig. 5.7. Here, we assume that the computation always starts with a single request.

- If the desired process p can perform a step satisfying the request ψ on top of the stack, we do it and remove the request from the stack (first rule).

- If the desired process p can perform a step, but it does not satisfy the request ψ, we update the system but keep the request in the stack (second rule).

- If a step on the desired process p is not possible, then we track the dependencies and add a new request on top of the stack. We have two rules: one for adding a request to a process to send a message we want to receive and another one to spawn the process we want to replay if it does not exist. Here, we use the auxiliary functions $sender$ and $parent$ to identify, respectively, the sender of a message and the parent of a process. Both functions $sender$ and $parent$ are easily computable from the logs in $\mathcal{L}(d)$.

The relation $\rightsquigarrow$ can be seen as a controlled version of the uncontrolled replay semantics in the sense that each derivation of the controlled semantics corresponds to a derivation of the uncontrolled one, while the opposite is not generally true. Notions for derivations and transitions are easily extended to controlled derivations. We also need a notion of projection from controlled systems to uncontrolled systems: $uctrl(\llbracket \Gamma; \Pi \rrbracket_\Psi) = \Gamma; \Pi$. The notion of projection trivially extends to derivations.

Theorem 5.19 (Soundness)**.** *For each controlled derivation d, $uctrl(d)$ is an uncontrolled derivation.*

While simple, this result allows one to recover many relevant properties from the uncontrolled semantics. For instance, by using the controlled semantics, if starting from a system $s = addLog(\mathcal{L}(d), \mathsf{init}(d))$ for some logging derivation d we find a wrong message $(p, p', \{v, \ell\})$, then we know that the same message exists also in d (from Theorem 5.18).

Our controlled semantics is not only sound but also minimal: causal-consistent replay redoes the minimal amount of actions needed to satisfy the replay request.

Here, we need to restrict the attention to requests that ask to replay transitions which are in the future of the process.

Definition 5.20. A controlled system $c = \llbracket s \rrbracket_{(\{p,\psi\})}$ is well initialized iff there are a derivation d under the logging semantics, a system $s_0 = addLog(\mathcal{L}(d), \mathsf{init}(d))$, an uncontrolled derivation $s_0 \rightharpoonup^* s$, and an uncontrolled derivation from s satisfying $\{p, \psi\}$.

The existence of a derivation satisfying the request can be efficiently checked. For replay requests $\{p, \mathsf{s}\}$ it is enough to check that process p can perform a step, for other replay requests it is enough to check the process log.

Theorem 5.21 (Minimality). *Let d be a controlled derivation such as* $\mathsf{init}(d) = [\![s]\!]_{(\{p,\psi\})}$ *is well-initialized. Derivation $uctrl(d)$ has minimal length among all uncontrolled derivations d' with* $\mathsf{init}(d') = s$ *including at least one transition satisfying the request $\{p, \psi\}$.*

5.6 Related Work and Conclusion

In this work, we have introduced (controlled) causal-consistent replay. It is strongly related (indeed dual) to the notion of causal-consistent reversibility, and its instance on debugging, causal-consistent reversible debugging, introduced in [51] for the *toy* language μOz. Beyond this, it has only been used so far in the CauDEr [85, 84] debugger for Erlang, which we took as a starting point for our prototype implementation (see [88]). Causal-consistent rollback has also been studied in the context of the process calculus HOπ [81] and the coordination language Klaim [53]. We refer to [51] for a description of the relations between causal-consistent debugging and other forms of reversible debugging.

The basic ideas in this paper are also applicable to other message-passing languages and calculi. In principle, the approach could also be applied to shared memory languages, yet it would require to log all interactions with shared memory (which may give rise, in principle, to an inefficient scheme).

An approach to record and replay for actor languages is introduced in [8]. While we concentrate on the theory, they focus on low-level issues: dealing with I/O, producing compact logs, etc. Actually, we could consider some of the ideas in [8] to produce more compact logs and thus reduce our instrumentation overhead.

At the semantic level, the work closest to ours is the reversible semantics for Erlang in [86]. However, all our semantics abstract away local queues in processes and their management. This makes the notion of independence much more natural, and it avoids some spurious conflicts between deliveries of different messages present in [86]. Moreover, our replay semantics is driven by the log of an actual execution, while the one in [86] is not. Finally, our controlled semantics, built on top of the uncontrolled reversible semantics, is much simpler than the low-level controlled semantics in [86] which, anyway, is based on undoing the actions of an execution up to a given checkpoint (rollback requests appeared later, in [85]).

None of the works above treats causal-consistent replay and, as far as we know, such notion has never been explored. For instance, no reference to it appears in a recent survey [26]. The survey classifies our approach as a message-passing multiprocessor scheme (the approach is studied in a single-processor multi-process set-

ting, but it makes no use of the single-processor assumption). It is in between content-based schemes (that record the content of the messages) and ordering-based schemes (that record the source of the messages), since it registers just unique identifiers for messages. This reduces the size of the log (content of long messages is not stored) w.r.t. content-based schemes, yet differently from ordering-based schemes it does not necessarily require to replay the system from a global checkpoint (but we do not yet consider checkpoints).

A related ordering-based scheme is [106]: it uses race detection to avoid logging all message exchanges, and we may try to integrate it in our approach in the future (though it considers only systems with a fixed number of processes). A content-based work is [96] for MPI programs, which does not replay calls to MPI functions, but just takes the values from the log. By applying this approach in our case, the state of Γ would not be replayed, and causal-consistent replay would not be possible since no relation between send and receive is kept.

Our work is also related to slicing, and in particular to [116], since it also deals with concurrent systems. Both approaches are based on causal consistency, but slicing considers the whole computation and extracts the fragment of it needed to explain a visible behavior, while we instrument the computation so to be able to go back and forward. Other differences include the considered languages—pi calculus vs Erlang—, the style of the semantics—labeled transitions vs reductions—, etc.

Chapter 6

Concolic Execution in Functional Programming by Program Instrumentation

Adrián Palacios[1], Germán Vidal[1]

[1] MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
{apalacios,gvidal}@dsic.upv.es

Abstract. Concolic execution, a combination of concrete and symbolic execution, has become increasingly popular in recent approaches to model checking and test case generation. In general, an interpreter of the language is augmented in order to also deal with symbolic values. In this paper, in contrast, we present an alternative approach that is based on a program instrumentation. Basically, the execution of the instrumented program in a standard environment produces a sequence of events that can be used to reconstruct the associated symbolic execution.

This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEOII/2015/013. Adrián Palacios was partially supported by the the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores de la Secretaría de Estado de Investigación, Desarrollo e Innovación del Ministerio de Economía y Competitividad* under FPI grant BES-2014-069749. This chapter is an adapted author version of the paper published in *"Adrián Palacios, Germán Vidal: Concolic Execution in Functional Programming by Program Instrumentation. Proceedings of LOPSTR 2015, Lecture Notes in Computer Science 9527: 277–292 (2015)"*. DOI: https://doi.org/10.1007/978-3-319-27436-2_17

6.1 Introduction

Software testing is one of the most widely used approaches for program validation. In this context, symbolic execution [74] was introduced as an alternative to random testing —which usually achieves a poor code coverage— or the complex and time-consuming design of test-cases by the programmer or software tester. In symbolic execution, one replaces the input data by symbolic values. Then, at each branching point of the execution, all feasible paths are explored and the associated constraints on symbolic values are stored. Symbolic states thus include a so called *path condition* with the constraints stored so far. Test cases are finally produced by solving the constraints in the leaves of the symbolic execution tree, which is typically incomplete since the number of states is often infinite.

Unfortunately, both the huge search space and the complexity of the constraints make test case generation based on symbolic execution difficult to scale. For instance, as soon as the path condition cannot be proved satisfiable, the execution of this branch is terminated in order to ensure soundness, giving rise to a poor coverage in many cases.

Concolic execution [56, 125] is a recent proposal that combines *conc*rete and symb*olic* execution, and overcomes some of the drawbacks of previous approaches. Essentially, concolic execution takes a program and some (initially random) concrete input data, and performs both a concrete and a symbolic execution that mimics the steps of the concrete execution. In this context, symbolic execution is simpler since we know the execution path that must be followed (the same of the concrete execution). Moreover, if the path condition becomes too complex and the constraint solver cannot prove its satisfiability, we can still push some concrete data from the concrete execution, thus simplifying it and often allowing the symbolic execution to continue. This technique forms the basis of some model checking and test-case generation tools (see, e.g., SAGE [57] and Java Pathfinder [115]). Test cases produced with this technique usually achieve a better code coverage than previous approaches based solely on symbolic execution. Moreover, it scales up better to complex or large programs.

Despite its popularity in the imperative and object-oriented programming paradigms, we can only find a few preliminary approaches to concolic execution in the context of functional and logic programming. To the best of our knowledge, the first approach for a high-level declarative programming language is [139], which presented a concolic execution scheme for logic programs, which was only aimed at a simple form of *statement* coverage. This approach was later extended and improved in [100]. In the context of functional programming, [138] introduced a formalization

of both concrete and symbolic execution for a simple subset of the functional and concurrent language Erlang [6], but the concolic execution procedure was barely sketched. More recently, [54] presented the design and implementation of a concolic testing tool for a complete functional subset of Erlang (i.e., the concurrency features are not considered in the paper). The tool, called CutEr, is publicly available from `https://github.com/aggelgian/cuter`.

However, the essential component of all these approaches is an interpreter augmented to also deal with symbolic values. In contrast, in this paper, we consider whether concolic execution can be performed by *program instrumentation*. We answer positively this question by introducing an stepwise approach based on *flattening* the initial program so that the return value of every expression is a pattern, and then instrumenting the resulting program so that its execution outputs a stream of events which suffice to reconstruct the associated symbolic execution. The main advantage w.r.t. the traditional approach to concolic execution is that the instrumented program can be run in any environment, even non-standard ones. For instance, one could run the instrumented program in a model checking environment like Concuerror [61] so that its execution would produce the sequences of events for all relevant interleavings, which might be useful for combining concolic testing and model checking.

The paper is organized as follows. Section 6.2 presents the considered language. Then, in Section 6.3, we present the instrumented semantics that outputs a sequence of events for each concrete execution. Section 6.4 introduces a program instrumentation that produces the same sequence of events but using the standard semantics. Section 6.5 presents a Prolog procedure for reconstructing the associated symbolic execution from the sequence of events. Finally, Section 6.6 concludes and points out some directions for further research.

6.2 The Language

In this section, we introduce the language considered in this paper. Our language is inspired in the concurrent functional language Erlang [6], which has a number of distinguishing features, like dynamic typing, concurrency via asynchronous message passing or hot code loading, that make it especially appropriate for distributed, fault-tolerant, soft real-time applications. Erlang's popularity is growing today due to the demand for concurrent services. But this popularity will also demand the development of powerful testing and verification techniques, thus the opportunity of our research.

$$
\begin{array}{rcl}
pgm & ::= & \mathrm{a}/n = \mathsf{fun}\ (X_1, \ldots, X_n) \to e. \mid pgm\ pgm \\
\mathsf{Exp} \ni e & ::= & \mathrm{a} \mid X \mid [\,] \mid [e_1|e_2] \mid \{e_1, \ldots, e_n\} \mid \mathsf{apply}\ e_0\ (e_1, \ldots, e_n) \\
& \mid & \mathsf{case}\ e\ \mathsf{of}\ clauses\ \mathsf{end} \mid \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \mid \mathsf{do}\ e_1\ e_2 \\
clauses & ::= & p_1 \to e_1; \ldots; p_n \to e_n \\
\mathsf{Pat} \ni p & ::= & [p_1|p_2] \mid [\,] \mid \{p_1, \ldots, p_n\} \mid \mathrm{a} \mid X \\
\mathsf{Value} \ni v & ::= & [v_1|v_2] \mid [\,] \mid \{v_1, \ldots, v_n\} \mid \mathrm{a}
\end{array}
$$

Figure 6.1: Core Erlang Syntax

Despite the fact that we plan to deal with full Erlang in the future, in this paper we only consider a functional subset of *Core Erlang* [24], an intermediate language used internally by the compiler.

The basic objects of the language are variables (denoted by $X, Y, \ldots \in \mathsf{Var}$), atoms (denoted by a, b, ...) and constructors (which are fixed in Erlang to lists, tuples and atoms); defined functions are named using atoms too (we will use, e.g., $\mathrm{f}/n, \mathrm{g}/m, \ldots$). The syntax for Core Erlang programs and expressions obeys the rules shown in Figure 6.1. Programs are sequences of function definitions. Each function f/n is defined by a rule $\mathsf{fun}\ (X_1, \ldots, X_n) \to e.$ where $X_1, \ldots, X_n$ are distinct variables and the body of the function, e, can be an atom, a process identifier, a variable, a list, a tuple, a function application, a case distinction, a let expression or a do construct (i.e., $\mathsf{do}\ e_1\ e_2$ evaluates sequentially e_1 and, then, e_2, so the value of e_1 is lost). Patterns are made of lists, tuples, atoms, and variables. Values are similar to patterns but cannot contain variables.

Example 6.1. Consider the Erlang function (left) and its translation to Core Erlang (right) shown in Figure 6.2, where some minor simplifications have been applied. Observe that Erlang's sequence operator "," is translated to a do operator when no value should be passed (using pattern matching) to the next elements in the sequence, and to a let expression otherwise. Note also that, despite the fact that this is not required by the syntax, some function applications are *flattened* in order to avoid nested applications. For this purpose, some additional let expressions are introduced. Moreover, additional default alternatives are added to each case expression in order to catch pattern matching errors, so it is common to have overlapping patterns in the clauses of a case construct.

As we will see later, for our instrumentation to be correct, we require some additional constraints on the syntax of programs. Basically, we require the following:

```
                                        f/2 = fun (X,Y) → do  apply g/1 (X),
f(X,Y) → g(X),                                                case apply h/1 (X) of
         case h(X) of                                           a → let Z = apply h/1 (Y)
           a → A = h(Y),                                              in apply g/1 (Z);
               g(A);                                            b → let V = apply h/1 ([])
           b → g(h([]))                                               in apply g/1 (V);
         end.                                                   W → fail
                                                              end.
```

Figure 6.2: Erlang function and its translation to Core Erlang

$$
\begin{array}{rcl}
pgm & ::= & \mathrm{a}/n = \mathsf{fun}\ (X_1, \ldots, X_n) \to \mathsf{let}\ X = e\ \mathsf{in}\ X. \mid pgm\ pgm \\
\mathsf{Exp} \ni e & ::= & \mathrm{a} \mid X \mid [\,] \mid [p_1|p_2] \mid \{p_1, \ldots, p_n\} \mid \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \mid \mathsf{do}\ e_1\ e_2 \\
 & \mid & \mathsf{let}\ p = \mathsf{apply}\ p_0\ (p_1, \ldots, p_n)\ \mathsf{in}\ e \mid \mathsf{let}\ p_1 = \mathsf{case}\ p_2\ \mathsf{of}\ clauses\ \mathsf{end}\ \mathsf{in}\ e \\
clauses & ::= & p_1 \to e_1; \ldots; p_n \to e_n \\
\mathsf{Pat} \ni p & ::= & [p_1|p_2] \mid [\,] \mid \{p_1, \ldots, p_n\} \mid \mathrm{a} \mid X \\
\mathsf{Value} \ni v & ::= & [v_1|v_2] \mid [\,] \mid \{v_1, \ldots, v_n\} \mid \mathrm{a}
\end{array}
$$

Figure 6.3: Flat language syntax

- both the name and the arguments of a function application must be patterns,
- the return value of a function must be a pattern,
- the argument of a case expression must be a pattern, and
- both function applications and case expressions can only occur in the right-hand side of a let expression.

The new constraints are needed in order to keep track of the intermediate values returned by expressions. These values are stored in a pattern, which can then be used by other expressions or returned as the result of a function application.

The restricted syntax is shown in Figure 6.3. In the following, we call the programs fulfilling this syntax *flat programs*. In practice, one can transform (purely functional) Core Erlang programs to our flat syntax using a simple pre-processing transformation. Furthermore, in the flat language we also require the *bound* variables in the body of the functions to have unique, fresh names. This is not strictly

necessary, but it simplifies the presentation by avoiding the use of context scopes associated to every let expression, etc. (as in [73], where the *last* binding of a variable in the environment should be considered to ensure that the right scope is used). We denote with $\overline{o_n}$ a sequence of objects $o_1, \ldots, o_n$. $\mathcal{V}ar(e)$ denotes the set of variables appearing in an expression e, and we say that e is *ground* if $\mathcal{V}ar(e) = \varnothing$.

In the following, we use the function bv to gather the bound variables of an expression:

Definition 6.2 (bound variables, bv)**.** Let e be an expression. The function $\mathsf{bv}(e)$ returns the set of bound variables of e as follows:

$$\mathsf{bv}(e) = \begin{cases} \{\,\} & \text{if } e \in \mathsf{Pat} \\ \mathcal{V}\mathsf{ar}(p) \cup \mathsf{bv}(e') & \text{if } e \equiv \mathsf{let}\ p\ = \mathsf{apply}\ p_0\ (p_1, \ldots, p_n)\ \mathsf{in}\ e' \\ \mathcal{V}\mathsf{ar}(p_0) \cup \ldots \cup \mathcal{V}\mathsf{ar}(p_n) \cup \mathsf{bv}(e_1) \cup \ldots \cup \mathsf{bv}(e') & \text{if } e \equiv \mathsf{let}\ p_0 = \mathsf{case}\ p\ \mathsf{of}\ \overline{p_n \to e_n}\ \mathsf{end}\ \mathsf{in}\ e' \\ \mathcal{V}\mathsf{ar}(p) \cup \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) & \text{if } e \equiv \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \\ \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) & \text{if } e \equiv \mathsf{do}\ e_1\ e_2 \end{cases}$$

where, in the fourth case, we assume that e_1 is neither an application nor a case expression (i.e., it is a pattern or another let expression).

6.3 Instrumented Semantics

In this section, we present an instrumented semantics for flat programs that produces a sequence of events that will suffice to reconstruct the associated symbolic execution. Essentially, we need to keep track of function calls, returns, let bindings and case selections.

First, let us note that the produced events will not show the actual run time values of the program variables, since they will not help us to reconstruct the associated symbolic execution. Rather, the events always include the static variable names. Therefore, in order to avoid variable name clashes, we will consider that variable names are *local* to every event. As a consequence, the two first elements of all events are $params$ and $vars$ denoting the list of parameters and the list of bound variables in the current function, respectively. These elements will be matched with the current values in the symbolic execution built so far in order to set the right environment for the operation represented by the event. See Section 6.5 for more details.

We consider the following events, which will suffice to reconstruct the symbolic execution:

- The first event, $\mathsf{call}(params, vars, p, [p_1, \ldots, p_n])$, is associated to a function application $\mathsf{let}\ p = \mathsf{apply}\ p_0\ (p_1, \ldots, p_n)\ \mathsf{in}\ e$. Here, $[p_1, \ldots, p_n]$ are the arguments of the function call, and p will be used to store the *return value* of the function call.
- The second event is $\mathsf{exit}(params, vars, p)$, where p is the pattern used to store the return value of the function body. We will produce an *exit* event at the end of every function.
- The next event is $\mathsf{bind}(params, vars, p, p')$, which binds the pattern p from a generic let expression (i.e., a let expression whose argument is neither an application nor a case expression) to the return value p' of that expression (see function ret below).
- Finally, for each expression of the form

  $$\mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of}\ p_1 \to e_1; \ldots; p_n \to e_n\ \mathsf{end}\ \mathsf{in}\ e$$

  we have two associated events. The first one is

  $$\mathsf{case}(params, vars, i, p_0, p_i, [(1, p_0, p_1), \ldots, (n, p_0, p_n)])$$

  Here, we store the position of the selected branch, i, the case argument p_0, the selected pattern p_i, as well a list with all case branches, which will become useful for producing alternative input data in the context of concolic testing. The second event is $\mathsf{exitcase}(params, vars, p, p')$, where p' is the return value of the selected branch (see below).

Before presenting the instrumented semantics, we need the following auxiliary function that identifies the *return value* of an expression:

Definition 6.3 (return value, ret)**.** Let e be an expression. We let $\mathsf{ret}(e)$ denote the return value of e as follows:

$$\mathsf{ret}(e) = \begin{cases} e & \text{if } e \in \mathsf{Pat} \\ \mathsf{ret}(e') & \text{if } e \equiv \mathsf{let}\ p\ = \mathsf{apply}\ p_0\ (p_1, \ldots, p_n)\ \mathsf{in}\ e' \\ \mathsf{ret}(e') & \text{if } e \equiv \mathsf{let}\ p_0 = \mathsf{case}\ p\ \mathsf{of}\ \overline{p_n \to e_n}\ \mathsf{end}\ \mathsf{in}\ e' \\ \mathsf{ret}(e_2) & \text{if } e \equiv \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \\ \mathsf{ret}(e_2) & \text{if } e \equiv \mathsf{do}\ e_1\ e_2 \end{cases}$$

where, in the fourth case, we assume that e_1 is neither an application nor a case expression (i.e., it is a pattern or another let expression).

Note that function ret is not well defined for arbitrary programs, e.g., $\mathsf{ret}(\mathsf{let}\ p = e\ \mathsf{in}\ \mathsf{apply}\ e_0\ (e_1, \ldots, e_n))$ is undefined. Extending the definition to cover this case would not help too since returning an expression which is not a pattern —like $\mathsf{apply}\ e_0\ (e_1, \ldots, e_n)$— would not be useful to reconstruct the symbolic execution (where the program is not available, only the sequence of events). This is why we transform the original programs to the flat form. In this case, it is immediate to see from the syntax in Fig. 6.3 that ret would always return a pattern for all program expressions.

The instrumented semantics for flat programs is formalized in Figure 6.4 following the style of a natural (big-step) semantics [73]. Observe that we do not need *closures* (as it is common in the natural semantics) since we do not allow fun expressions in the body of a function in this paper. Here, we use an *environment* θ —i.e., a mapping from variables to patterns— because we need to know the static values of the variables for the instrumentation (e.g., we use the case argument that appears statically in the program, rather than the instantiated run time value). The main novelty is that, for the instrumentation, we also need to keep track of the function where an expression occurs. For this purpose, we also introduce a simple context π that stores this information, i.e., for a given function $\mathsf{fun}\ (X_1, \ldots, X_n) \rightarrow e$ we store a tuple $\langle [X_1, \ldots, X_n], [\mathsf{bv}(e)] \rangle$. The environment is only updated in function applications, where $[\mathsf{bv}(e)]$ denotes a list with the variables returned by $\mathsf{bv}(e)$.

Let us briefly explain the rules of the semantics. Statements have the form $\pi, \theta \vdash e \Downarrow_\tau p$, where π is the aforementioned context, θ is a substitution (the environment), e is an expression, τ is a sequence of events, and p is a pattern —the value of e.

The first rule deals with patterns (including variables, atoms, tuples and lists). Here, the evaluation just proceeds by applying the current environment θ to the pattern p to bind its variables (if any), which is denoted by $p\theta$. The associated sequence of events is ϵ denoting an empty sequence.

The next rule deals with function applications. In this case, the context is necessary for setting the first and second parameters of call and exit events. Note that since we only consider flat programs, both the function name and the arguments are patterns; thus, their evaluation amounts to binding their variables using the current environment, which explains why the associated sequences of events are ϵ. Note also that, when recursively evaluating the body of the function, we update the context with the information of the function called. The bound variables are collected using the function bv; and, as mentioned before, in the flat language we assume that

$$\frac{}{\pi, \theta \vdash p \Downarrow_\epsilon p\theta}$$

$$\frac{\begin{array}{c}\langle vs, ps\rangle, \theta \vdash p_0 \Downarrow_\epsilon \mathrm{f}/m \quad \ldots \quad \langle vs, ps\rangle, \theta \vdash p_m \Downarrow_\epsilon p'_m \\ \langle[\overline{Y_m}], [\mathsf{bv}(e_2)]\rangle, \theta \cup \sigma \vdash e_2 \Downarrow_{\tau_1} p' \qquad \langle vs, ps\rangle, \theta \cup \sigma' \vdash e \Downarrow_{\tau_2} p''\end{array}}{\langle vs, ps\rangle, \theta \vdash \mathsf{let}\ p = \mathsf{apply}\ p_0\ (\overline{p_m})\ \mathsf{in}\ e \Downarrow_{\mathsf{call}(vs,ps,p,[\overline{p_m}])+\tau_1+\mathsf{exit}([\overline{Y_m}],[\mathsf{bv}(e_2)],p''_2)+\tau_2} p''}$$

$$\text{if } \mathrm{f}/m = \mathsf{fun}\ (\overline{Y_m}) \to e_2 \in \mathsf{pgm},\ \mathsf{ret}(e_2) = p''_2,\ \mathsf{match}(\overline{Y_m}, \overline{p'_m}) = \sigma,\ \mathsf{match}(p, p') = \sigma'$$

$$\frac{\langle vs, ps\rangle, \theta \vdash p_0 \Downarrow_\epsilon p'_0 \qquad \langle vs, ps\rangle, \theta \cup \sigma \vdash e_i \Downarrow_{\tau_1} p'_i \qquad \langle vs, ps\rangle, \theta \cup \sigma' \vdash e \Downarrow_{\tau_2} p'}{\langle vs, ps\rangle, \theta \vdash \mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of}\ \mathit{clauses}\ \mathsf{end}\ \mathsf{in}\ e \Downarrow_{\mathsf{case}(vs,ps,i,p_0,p_i,\mathit{alts})+\tau_1+\mathsf{exitcase}(vs,ps,p,p'_i)+\tau_2} p'}$$

$$\text{if } \mathit{clauses} = p_1 \to e_1; \ldots; p_m \to e_m,\ \mathsf{cmatch}(p'_0, \mathit{clauses}) = (i, p_i, \sigma),\ \mathit{alts} = [(1, p_0, p_1), \ldots, (m, p_0, p_m)],\ \mathsf{ret}(e_i) = p'_i,\ \mathsf{match}(p, p'_i) = \sigma'$$

$$\frac{\pi, \theta \vdash e_1 \Downarrow_{\tau_1} p'_1 \qquad \pi, \theta \cup \sigma \vdash e_2 \Downarrow_{\tau_2} p}{\pi, \theta \vdash \mathsf{let}\ p_1 = e_1\ \mathsf{in}\ e_2 \Downarrow_{\tau_1+\mathsf{bind}(vs,ps,p_1,\mathsf{ret}(e_1))+\tau_2} p} \text{ if } \mathsf{match}(p_1, p'_1) = \sigma$$

$$\frac{\pi, \theta \vdash e_1 \Downarrow_{\tau_1} p_1 \qquad \pi, \theta \vdash e_2 \Downarrow_{\tau_2} p_2}{\pi, \theta \vdash \mathsf{do}\ e_1\ e_2 \Downarrow_{\tau_1+\tau_2} p_2}$$

Figure 6.4: Flat language instrumented semantics

they all have different, fresh names. Observe that the subcomputation for evaluating the body of the function called is preceded by the call event and followed by an exit event. Here, we use the auxiliary function match to compute the matching substitution (if any) between two patterns, i.e., $\mathsf{match}(p_1, p_2) = \sigma$ if $\mathcal{D}\mathsf{om}(\sigma) \subseteq \mathcal{V}ar(p_1)$ and $p_1\sigma = p_2$, and fail otherwise. In this rule, $\mathsf{match}(\overline{Y_m}, \overline{p'_m})$ just returns the substitution $\{Y_1 \mapsto p'_1, \ldots, Y_m \mapsto p'_m\}$. The *update* of an environment θ using σ is denoted by $\theta \cup \sigma$. Formally, $\theta \cup \sigma = \delta$ such that $X\delta = \sigma(X)$ if $X \in \mathcal{D}\mathsf{om}(\sigma)$ and $X\delta = X\theta$ otherwise (i.e., σ has higher priority than θ). Observe that we use the evaluated patterns $p'_1, \ldots, p'_m$ to update the environment, but the original, static patterns $p_1, \ldots, p_m$ in the call event.

The next rule is used to evaluate case expressions. Here, we produce case and exitcase events that also include the parameter variables of the function and the bound variables. For selecting the matching branch of the case expression, we use the auxiliary function cmatch that is defined as follows: $\mathsf{cmatch}(p, p_1 \to e_1; \ldots; p_n \to e_n) = (i, p_i, \sigma)$ if $\mathsf{match}(p, p_i) = \sigma$ for some $i \in \{1, \ldots, n\}$ and $\mathsf{match}(p, p_j) = \mathsf{fail}$

for all $j < i$. Informally speaking, cmatch selects the first matching branch of the case expression, which follows the usual semantics of Erlang. As in the previous rule, note that we use p'_0 in cmatch but the original, static pattern p_0 in the case event.

The following rule is is used to evaluate let expressions. It produces a single bind event which includes, as usual, the parameter variables of the function and the bound variables. Finally, the last rule deals with do expressions. Here, we proceed as expected and return the concatenation of the sequences of events produced when evaluating the subexpressions.

In the following, without loss of generality, we assume that the entry point to the program is always the distinguished function main/n.

Definition 6.4 (instrumented execution)**.** Given a flat program pgm and an initial expression, apply main/n $(p_1, \ldots, p_n)$, with $\mathrm{main}/n = \mathsf{fun}\ (X_1, \ldots, X_n) \to e \in \mathsf{pgm}$, its evaluation is denoted by $\langle [\overline{X_n}], [\mathsf{bv}(e)] \rangle, \theta \vdash e \Downarrow_\tau v$, where $\theta = \{X_1 \mapsto p_1, \ldots, X_n \mapsto p_n\}$ is a substitution, v is the computed value and $\tau + \mathsf{exit}([\overline{X_n}], [\mathsf{bv}(e)], \mathsf{ret}(e))$ is the associated sequence of events.

Example 6.5. Let us consider the flat program shown in Figure 6.5. An example computation for apply main/1 ([a]) with the instrumented semantics is shown in Figure 6.6. Therefore, the associated sequence of events[1] is the following:

$$
\begin{array}{l}
\mathsf{call}([X], [W], W, [X, X]) \\
\mathsf{case}([X, Y], [W_1, H, T, W_2], 2, X, [H|T], [(1, X, [\,]), (2, X, [H|T])]) \\
\mathsf{call}([X, Y], [W_1, H, T, W_2], W_2, [T, Y]) \\
\mathsf{case}([X, Y], [W_1, H, T, W_2], 1, X, [\,], [(1, X, [\,]), (2, X, [H|T])]) \\
\mathsf{exitcase}([X, Y], [W_1, H, T, W_2], W_1, Y) \\
\mathsf{exit}([X, Y], [W_1, H, T, W_2], W_1) \\
\mathsf{exitcase}([X, Y], [W_1, H, T, W_2], W_1, [H|W_2]) \\
\mathsf{exit}([X, Y], [W_1, H, T, W_2], W_1) \\
\mathsf{exit}([X], [W], W)
\end{array}
$$

Let us remind that variable names are *local* to each event. Also, observe that the events do not need to store the names of the invoked functions since we are only interested in the sequence of pattern matching operations, as we will see in Section 6.5.

[1]Note that the flat program is not syntactically correct according to Fig. 6.3 since the right-hand side of the functions do not have the form let $X = e$ in X with e a pattern, a let binding or a do expression. Here, we keep this simpler formulation for clarity, and it also simplifies the sequence of events by avoiding some redundant bind events.

$$
\begin{array}{rl}
\mathrm{main}/1 \;=\; \mathsf{fun}\ (X) \rightarrow & \mathsf{let}\ W = \mathsf{apply}\ \mathrm{app}/2\ (X, X)\ \mathsf{in}\ W \\
\mathrm{app}/2 \;=\; \mathsf{fun}\ (X, Y) \rightarrow & \mathsf{let}\ W_1 = \mathsf{case}\ X\ \mathsf{of} \\
 & \qquad [\,] \rightarrow Y \\
 & \qquad [H|T] \rightarrow \mathsf{let}\ W_2 = \mathsf{apply}\ \mathrm{app}/2\ (T, Y)\ \mathsf{in}\ [H|W_2] \\
 & \quad \mathsf{end} \\
 & \mathsf{in}\ W_1
\end{array}
$$

Figure 6.5: Example flat program

Note that the semantics is a conservative extension of the standard semantics in the sense that the generation of events does not affect the evaluation, i.e., if we remove the context information and the events labeling the arrows, we are back to the standard semantics of an eager functional language essentially equivalent to that in [73].

We will show a method for constructing the associated symbolic execution (as well as its potential alternatives) in Section 6.5.

6.4 Program Instrumentation

In this section, we present a program transformation that instruments a program so that its standard execution will return the same sequence of events produced with the original program and the instrumented semantics of Figure 6.4.

For this purpose, we introduce the predefined function out, which outputs its first argument (e.g., to a given file or to the standard output) and returns its second argument. This function is implemented as a function *call* (i.e., not as a function application) so that there is no conflict when performing the instrumentation.

Definition 6.6 (program instrumentation)**.** Let pgm be a flat program. We instrument pgm by replacing each function definition:

$$\mathrm{f}/k = \mathsf{fun}\ (X_1, \ldots, X_k) \rightarrow \mathsf{let}\ X = e\ \mathsf{in}\ X$$

with a new function definition of the form

$$\mathrm{f}/k = \mathsf{fun}\ (X_1, \ldots, X_k) \rightarrow [\![\mathsf{let}\ X = e\ \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exit}(vs, bs, X)\text{”}, X)]\!]_{\mathsf{F}}^{vs,ps}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\pi_2, \theta_4 \vdash Y \Downarrow_\epsilon [\mathrm{a}]} \quad \dfrac{}{\pi_2, \theta_5 \vdash W_1 \Downarrow_\epsilon [\mathrm{a}]}}{\pi_2, \theta_4 \vdash \mathsf{let}\ W_1 = \mathsf{case} \ldots \Downarrow_{\tau_1} [\mathrm{a}]} \quad \dfrac{}{\pi_2, \theta_6 \vdash [H|W_2] \Downarrow_\epsilon [\mathrm{a}, \mathrm{a}]}}{\pi_2, \theta_3 \vdash \mathsf{let}\ W_2 = \mathsf{apply} \ldots \Downarrow_{\tau_2} [\mathrm{a}, \mathrm{a}]} \quad \dfrac{}{\pi_2, \theta_7 \vdash W_1 \Downarrow_\epsilon [\mathrm{a}, \mathrm{a}]}}{\pi_2, \theta_2 \vdash \mathsf{let}\ W_1 = \mathsf{case} \ldots \Downarrow_{\tau_3} [\mathrm{a}, \mathrm{a}]} \quad \dfrac{}{\pi_1, \theta_8 \vdash W \Downarrow_\epsilon [\mathrm{a}, \mathrm{a}]}}{\pi_1, \theta_1 \vdash \mathsf{let}\ W = \mathsf{apply}\ \mathrm{app}/2\ (X, X)\ \mathsf{in}\ W \Downarrow_{\tau_4} [\mathrm{a}, \mathrm{a}]}$$

with

$$\pi_1 = \langle [X], [W] \rangle \ \text{ and } \ \pi_2 = \langle [X, Y], [W_1, W_2 H, T] \rangle$$

$$\begin{array}{lcllcl}
\theta_1 & = & \{X \mapsto [\mathrm{a}]\} & \theta_2 & = & \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}]\} \\
\theta_3 & = & \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}], H \mapsto \mathrm{a}, T \mapsto [\,]\} & \theta_4 & = & \{X \mapsto [\,], Y \mapsto [\mathrm{a}]\} \\
\theta_5 & = & \{X \mapsto [\,], Y \mapsto [\mathrm{a}], W_1 \mapsto [\mathrm{a}]\} & \theta_6 & = & \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}], H \mapsto \mathrm{a}, T \mapsto [\,], W_2 \mapsto [\mathrm{a}]\} \\
\theta_7 & = & \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}], W_1 \mapsto [\mathrm{a}, \mathrm{a}]\} & \theta_8 & = & \{X \mapsto [\mathrm{a}], W \mapsto [\mathrm{a}, \mathrm{a}]\}
\end{array}$$

$$\begin{array}{rcl}
\tau_1 & = & \mathsf{case}([X, Y], [W_1, W_2], 1, X, [\,], [(1, X, [\,]), (2, X, [H|T])]) \\
 & & +\mathsf{exitcase}([X, Y], [W_1, W_2], W_1, Y) \\
\tau_2 & = & \mathsf{call}([X, Y], [W_1, W_2], W_2, [T, Y]) + \tau_1 + \mathsf{exit}([X, Y], [W_1, W_2], W_1) \\
\tau_3 & = & \mathsf{case}([X, Y], [W_1, W_2], 2, X, [H|T], [(1, X, [\,]), (2, X, [H|T])]) + \tau_2 \\
 & & +\mathsf{exitcase}([X, Y], [W_1, W_2], W_1, [H|W_2]) \\
\tau_4 & = & \mathsf{call}([X], [W], W, [X, X]) + \tau_3 + \mathsf{exit}([X, Y], [W_1, W_2], W_1)
\end{array}$$

Figure 6.6: Example computation with the instrumented semantics

where $vs = [\overline{X_k}]$, $ps = [\mathsf{bv}(e)]$, F is a flag to determine if an exitcase event should be produced when a pattern is reached (see below), and the auxiliary function $[\![\ ]\!]$ is shown in Figure 6.7.

Let us briefly explain the rules of the instrumentation. First, we add an exit event at the end of each function. An additional bind event is also required when the expression e is neither a function application nor an case expression in order to explicitly bind X to the return expression of e (for function applications and case expressions this is already done in the exit and exitcase events, respectively). Then, we also add call and case events in each occurrence of a function application and a case expression, respectively. Finding the value returned by a case expression is a bit more subtle. For this purpose, we introduce a flag that is propagated through the different cases so that only when the expression is the last expression in a case branch (a pattern) we produce an exitcase event. For let expressions, we produce a bind event and continue evaluating both the expression in the right-hand side of the binding and the result. Finally, the *default* case —the last equation in Figure 6.7— is only used to ignore the call to the predefined function out/2.

$$
\begin{array}{rcl}
[\![e]\!]^{vs,ps}_{\mathsf{F}} & = & e \ \text{ if } e \in \mathsf{Pat} \\
[\![e]\!]^{vs,ps}_{\mathsf{T}(p)} & = & \mathsf{out}(\text{“}\mathsf{exitcase}(vs,ps,p,e)\text{”}, e) \ \text{ if } e \in \mathsf{Pat} \\
[\![\mathsf{let}\ p = \mathsf{apply}\ p_0\ (\overline{p_n})\ \mathsf{in}\ e]\!]^{vs,ps}_{b} & = & \mathsf{let}\ p = \mathsf{out}(\text{“}\mathsf{call}(vs,ps,p,[p_1,\ldots,p_n])\text{”}, \\
 & & \qquad \mathsf{apply}\ p/0\ (p_1,\ldots,p_n)\ ) \\
 & & \mathsf{in}\ [\![e]\!]^{vs,ps}_{b} \\
[\![\mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of} & = & \mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of} \\
\quad p_1 \to e_1; & & \quad p_1 \to \mathsf{out}(\text{“}\mathsf{case}(vs,ps,1,p_0,p_1,alts)\text{”}, \\
 & & \qquad [\![e_1]\!]^{vs,ps}_{\mathsf{T}(p)}\ ) \\
\quad \ldots & & \quad \ldots \\
\quad p_n \to e_n & & \quad p_n \to \mathsf{out}(\text{“}\mathsf{case}(vs,ps,n,p_0,p_n,alts)\text{”}, \\
 & & \qquad [\![e_n]\!]^{vs,ps}_{\mathsf{T}(p)}\ ) \\
\quad \mathsf{end} & & \quad \mathsf{end} \\
\quad \mathsf{in}\ e]\!]^{vs,ps}_{b} & & \quad \mathsf{in}\ [\![e]\!]^{vs,ps}_{b} \\
[\![\mathsf{let}\ p = e_1\ \mathsf{in}\ e_2]\!]^{vs,ps}_{b} & = & \mathsf{let}\ p = [\![e_1]\!]^{vs,ps}_{\mathsf{F}}\ \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{bind}(vs,ps,p,\mathsf{ret}(e_1))\text{”}, \\
 & & \qquad [\![e_2]\!]^{vs,ps}_{b}\ ) \\
[\![\mathsf{do}\ e_1\ e_2]\!]^{vs,ps}_{b} & = & \mathsf{do}\ [\![e_1]\!]^{vs,ps}_{\mathsf{F}}\ [\![e_2]\!]^{vs,ps}_{b} \\
[\![e]\!]^{vs,ps}_{b} & = & e \ \text{ otherwise}
\end{array}
$$

where $alts = [(p_0, 1, p_1), \ldots, (p_0, n, p_n)]$

Figure 6.7: Program instrumentation

Example 6.7. Consider again the flat program of Example 6.5. The instrumented program is shown in Figure 6.8.

It can easily be shown that the instrumented program produces the same sequence of events of Example 6.5, e.g., by executing the program in the standard environment of Erlang (together with an appropriate definition of out/2).

The correctness of the program instrumentation is stated in the next result:

Theorem 6.8. *Let pgm be a flat program and pgm^I its instrumented version according to Definition 6.6. Given an initial expression,* apply $\mathrm{main}/n\ (p_1, \ldots, p_n)$*, its execution using pgm and the instrumented semantics (according to Definition 6.4) produces the same sequence of events as its execution using pgm^I and the standard semantics.*

Proof. We prove that for all program expressions, e, we have that $\langle vs, ps\rangle, \theta \vdash e \Downarrow_\tau p$ implies $\theta \vdash [\![e]\!]^{vs,ps}_{\mathsf{F}} \Downarrow p$ with the standard semantics[2] and, moreover, it outputs

[2]Here, we consider that the *standard* semantics is that of Figure 6.4 without the events labeling the

$$
\begin{array}{l}
\mathrm{main}/2 \;=\; \mathsf{fun}\ (X) \rightarrow \quad \mathsf{let}\ W = \ \mathsf{out}(\text{“}\mathsf{call}([X],[W],W,[X,X])\text{”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{apply}\ \mathrm{app}/2\ (X,X)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exit}([X],[W],W)\text{”},W) \\
\mathrm{app}/2 \;=\; \mathsf{fun}\ (X,Y) \rightarrow \\
\quad \mathsf{let}\ W_1 = \mathsf{case}\ X\ \mathsf{of} \\
\qquad\qquad [\,] \rightarrow\ \mathsf{out}(\text{“}\mathsf{case}([X,Y],[W_1,W_2,H,T],1,X,[\,],alts)\text{”}, \\
\qquad\qquad\qquad\qquad\quad \mathsf{out}(\text{“}\mathsf{exitcase}([X,Y],[W_1,W_2,H,T],W_1,Y)\text{”},Y)) \\
\qquad\qquad [H|T] \rightarrow\ \mathsf{out}(\text{“}\mathsf{case}([X,Y],[W_1,W_2,H,T],2,X,[H|T],alts)\text{”}, \\
\qquad\qquad\qquad\qquad\quad \mathsf{let}\ W_2 = \ \mathsf{out}(\text{“}\mathsf{call}([X,Y],[W_1,W_2,H,T],W_2,[T,Y])\text{”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{apply}\ \mathrm{app}/2\ (T,Y))) \\
\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exitcase}([X,Y],[W_1,W_2,H,T],W_1,[H|W_2])\text{”}, \\
\qquad\qquad\qquad\qquad\qquad\quad [H|W_2]) \\
\quad \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exit}([X,Y],[W_1,W_2,H,T],W_1)\text{”},W_1)
\end{array}
$$

where $alts = [(1, X, []), (2, X, [H|T])]$.

Figure 6.8: Instrumented program

the same sequence of events τ. The claim of the theorem is an easy consequence of this property. We prove the claim by induction on the depth k of the proof tree with the instrumented semantics.

Since the base case $k = 0$ is trivial (the rule to evaluate a pattern is the same in both cases), we now consider the inductive case $k > 0$. We distinguish the following cases depending on the applied rule from the semantics of Fig. 6.4:

- The first rule of the semantics is not applicable since the depth of the proof is $k > 0$.
- If the applied rule is the second one (to evaluate a function call), then the considered transition has the form

  $$\langle vs, ps\rangle, \theta \vdash \mathsf{let}\ p = \mathsf{apply}\ p_0\ (\overline{p_m})\ \mathsf{in}\ e \Downarrow_\tau p''$$

  with $\tau = \mathsf{call}(vs, ps, p, [\overline{p_m}]) + \tau_1 + \mathsf{exit}([\overline{Y_m}], [\mathsf{bv}(e_2)], p_2'') + \tau_2$. The instrumented expression expression is thus

  $$[\![\mathsf{let}\ p = \mathsf{apply}\ p_0\ (\overline{p_m})\ \mathsf{in}\ e]\!]_b^{vs,ps}$$

  Following the rules of Fig. 6.7, this is transformed to

  $$\mathsf{let}\ p = \mathsf{out}(\text{“}\mathsf{call}(vs, ps, p, [\overline{p_m}])\text{”}, \mathsf{apply}\ p_0\ (\overline{p_m}))\ \mathsf{in}\ [\![e]\!]_b^{vs,ps}$$

transitions.

such that the execution of this instrumented code will first output the event $\mathsf{call}(vs, ps, p, [\overline{p_m}])$ similarly to the instrumented semantics. By the induction hypothesis, the evaluation of $p_0, \ldots, p_m$ and e with the instrumented semantics produces the same values and outputs the same events than with their instrumented versions with the standard semantics. Let us now consider that p_0 evaluates to function f/m, whose definition is as follows: $\mathrm{f}/m = \mathsf{fun}\ \overline{Y_m} \to \mathsf{let}\ X = e\ \mathsf{in}\ X$. In the instrumented program, the same function has the form

$$\mathrm{f}/m = \mathsf{fun}\ (X_1, \ldots, X_m) \to [\![\mathsf{let}\ X = e\ \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exit}(vs, bs, X)\text{”}, X)]\!]_{\mathsf{F}}^{vs', ps'}$$

$vs' = [\overline{Y_m}]$ and $ps' = [\mathsf{bv}(e)]$. By the induction hypothesis, we know that the sequence of events for $\mathsf{let}\ X = e\ \mathsf{in}\ X$ in the instrumented semantics, is the same as that of $[\![\mathsf{let}\ X = e\ \mathsf{in}\ X]\!]_{\mathsf{F}}^{vs', ps'}$, therefore the claim follows.

- If the applied rule is the second one (to evaluate a function call), then the considered transition has the form

$$\langle vs, ps\rangle, \theta \vdash \mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of}\ \mathit{clauses}\ \mathsf{end}\ \mathsf{in}\ e \Downarrow_\tau p_0'$$

with $\mathit{clauses} = \overline{p_l \to e_l}$ and

$$\tau = \mathsf{case}(vs, ps, i, p_0, p_i, \mathit{alts}) + \tau_1 + \mathsf{exitcase}(vs, ps, p, p_i') + \tau_2$$

The instrumented expression expression is thus

$$[\![\mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of}\ \mathit{clauses}\ \mathsf{end}\ \mathsf{in}\ e]\!]_b^{vs,ps}$$

which is transformed to

$$\mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of}\ \mathit{clauses}'\ \mathsf{end}\ \mathsf{in}\ [\![e]\!]_b^{vs,ps}$$

with $\mathit{clauses}' = \overline{\mathsf{out}(\text{“}\mathit{case}(vs, ps, l, p_0, p_l, \mathit{alts})\text{”}, [\![e_l]\!]_{\mathsf{T}(p)}^{vs,ps})}$. By the induction hypothesis, we have that $\langle vs, ps\rangle, \theta \cup \sigma \vdash e_i \Downarrow_{\tau_1} p_i'$ implies $[\![e_i]\!]_{\mathsf{F}}^{vs,ps} \Downarrow p_i'$ outputs the sequence of events τ_1. Therefore, $[\![e_i]\!]_{\mathsf{T}(p)}^{vs,ps} \Downarrow p_i'$ outputs and additional event exitcase, and the claim follows by induction.

- Proving the claim for the two remaining rules is straightforward by the induction hypothesis.

□

A prototype implementation of the program instrumentation can be found at `http://kaz.dsic.upv.es/instrument.html`. Here, one can introduce a (restricted) Erlang program that is first transformed to the flat syntax and, then, instrumented (several input examples are provided). Moreover, it is also possible to run the instrumented program and obtain the corresponding sequence of events.

6.5 Concolic Execution

The relevance of the computed sequences of events is that one can easily reconstruct a symbolic execution that mimics the steps of the concrete execution that produced the sequence of events, as well as to produce alternative bindings for the initial variables so that a different execution path will be followed.

Let us first formalize the reconstruction of the symbolic execution from a sequence of events using the Prolog program shown in Fig. 6.9. As mentioned before, we should ensure that the elements of τ are renamed apart. In our implementation, the sequence of events is written to a file, that is then consulted as a sequence of *facts* and, thus, their variables are always renamed apart. For simplicity, we do not show these low level details in Fig. 6.9 but just assume the events in τ have been renamed apart.

$sym(\tau, Res, Vars) \leftarrow eval(\tau, [(Res, Vars, BVars)]).$

$eval([\,], [\,]).$

$eval([call(Vars, BVars, NRes, NVars)|Tau], [(Res, Vars, BVars)|Env]) \leftarrow$
$\quad eval(Tau, [(NRes, NVars, NBVars), (Res, Vars)|Env]).$

$eval([case(Vars, BVars, N, Arg, Pat, Alts)|Tau], [(Res, Vars, BVars)|Env]) \leftarrow$
$\quad Arg = Pat,\ eval(Tau, [(Res, Vars, BVars)|Env]).$

$eval([exitcase(Vars, BVars, Arg, Pat)|Tau], [(Res, Vars, BVars)|Env]) \leftarrow$
$\quad Arg = Pat,\ eval(Tau, [(Res, Vars, BVars)|Env]).$

$eval([bind(Vars, BVars, Pat1, Pat2)|R], [(Res, Vars, BVars)|Env]) \leftarrow$
$\quad Pat1 = Pat2, eval(R, [(Res, Vars, BVars)|Env]).$

$eval([exit(Vars, BVars, Pat)|Tau], [(Res, Vars, BVars)|Env]) \leftarrow$
$\quad Res = Pat,\ eval(Tau, Env).$

Figure 6.9: Prolog procedure for symbolic execution

Let us briefly explain the rules of the procedure. The first clause just calls $eval$

and initializes an stack of function environments with $(Res,\ Vars, BVars)$, where Res is the result of the evaluation, $Vars$ are the variables of the main function, and $BVars$ are the bounded variables of the main function. When calling sym, all these three variables are unbound.

The first rule of $eval/2$ just finishes the computation when there are no events to be processed.

The next rule deals with $call$ events and just pushes a new environment $(NRes,\ NVars, NBVars)$ into the stack of environments. Observe that the names of variables $Vars$ and $BVars$ occurs twice in the head of the clause —as arguments of the event and as in the current environment— which makes them unify and thus set the right values for them in the current symbolic execution. This is done in all the clauses.

The next rule deals with $case$ events and it main purpose is to unify Arg and Pat, which represent the case argument and the selected pattern, respectively.

The next rule takes an $exitcase$ event and proceeds similarly to the previous one by matching Arg and Pat, now denoting the pattern of a let expression and the result of the evaluation of a case branch.

The next rule deals with a $bind$ event in the obvious way by unifying the given patterns Pat_1 and Pat_2.

Finally, the last rule matches Res in the current environment (used to store the output of the current function call) with the pattern Pat and, moreover, pops the environment $(Res,\ Vars, BVars)$ from the stack of environments.

For example, given the sequence of events of Example 6.5 and the initial call $sym(\tau, Res,\ Vars)$, the above program returns:

$$Res = [X, X],\ \ Vars = [X]$$

which obviously produces less instantiated values than the concrete execution (where we had $Res = [\mathrm{a}, \mathrm{a}],\ \ Vars = [a]$).

For concolic testing, though, one is not interested in computing the symbolic execution associated to the concrete execution, but in alternative symbolic executions so that the produced data will give rise to different concrete executions. Luckily, it is easy to extend the previous procedure in order to compute alternative symbolic executions by just replacing the clause for $case$ events as follows:

$$\begin{array}{l} eval([case(\ Vars, BVars, N, Arg, Pat, Alts)|Tau], [(Res,\ Vars, BVars)|Env]) \\ \quad \leftarrow member((M, Arg', Pat'), Alts), \\ \quad\quad N \neq M,\ Arg' = Pat', \\ \quad\quad eval(Tau, [(Res,\ Vars, BVars)|Env]). \end{array}$$

By using the call $member((M, Arg', Pat'), Alts)$, this rule nondeterministically chooses all the alternative selections in case expressions, thus producing alternative bindings for the initial call. For instance, for the sequence of events of Example 6.5, we get three (nondeterministic) answers:

$$Vars = [\,]\,; \quad Vars = [X]\,; \quad Vars = [X, Y|R]$$

An implementation of the concolic testing tool has been undertaken. The first stage, flattening and instrumenting the source program has been implemented in Erlang itself, and can be tested at `http://kaz.dsic.upv.es/instrument.html`. In contrast, the concolic testing algorithm is being implemented in Prolog, since the facilities of this language —unification and nondeterminism— make it very appropriate for dealing with symbolic executions.

6.6 Discussion

In this paper, we have introduced a transformational approach to concolic execution that is based on flattening and instrumenting the source program —a simple first order, eager functional language—. The execution of the instrumented program gives rise to a stream of events that can then be easily processed in order to compute the variable bindings of the associated symbolic executions, as well as possible alternatives. To the best of our knowledge, our paper proposes the first approach to concolic execution by program instrumentation in the context of functional (or logic) programming. In contrast to using an interpreter-based design, in our approach the instrumented program can be run in any environment, even non-standard ones, which opens the door, for instance, to run the instrumented program in a model checking environment like Concuerror [61] so that its execution would produce the sequences of events for all relevant interleavings.

As a future work, we plan to extend our approach in order to cover a larger subset of Erlang as well as to design a fully automatic procedure for concolic testing (currently, one should manually run the instrumented program and the Prolog procedure for generating alternative bindings). Here, we expect that our transformational approach will be useful to cope with concurrent programs, as mentioned above.

Acknowledgments

We thank the anonymous reviewers and the participants of LOPSTR 2015 for their useful comments to improve this paper.

Chapter 7

Property-Based Test Case Generators for Free

Emanuele De Angelis[1], Fabio Fioravanti[1], Adrián Palacios[2], Alberto Pettorossi[3], Maurizio Proietti[4]

[1] DEC, University "G. d'Annunzio" of Chieti-Pescara, Italy
{emanuele.deangelis,fabio.fioravanti}@unich.it

[2] MiST, DSIC, Universitat Politècnica de València
apalacios@dsic.upv.es

[3] University of Roma Tor Vergata, Italy
pettorossi@info.uniroma2.it

[4] CNR-IASI, Roma, Italy
maurizio.proietti@iasi.cnr.it

Abstract. Property-Based Testing requires the programmer to write suitable *generators*, i.e., programs that generate (possibly in a random way) input values for which the program under test should be run. However, the process of writing generators is quite a costly, error-prone activity. In the context of Property-Based Testing of Erlang programs, we propose an approach to relieve the programmer from the task of writing generators. Our approach

This work has been partially supported by the EU (FEDER) and the *Spanish Ministerio de Ciencia, Innovación y Universidades*/AEI, grant TIN2016-76843-C4-1-R and by the *Generalitat Valenciana*, grant PROMETEO-II/2015/013 (SmartLogic). E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti are members of the INdAM Research group GNCS. E. De Angelis, F. Fioravanti, and A. Pettorossi are research associates at CNR-IASI, Rome, Italy. A. Palacios was partially supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores* and *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D* (MICINN) under FPI grants BES-2014-069749 and EEBB-I-17-12101. This chapter is an adapted author version of the paper published in *"Emanuele De Angelis, Fabio Fioravanti, Adrián Palacios, Alberto Pettorossi, Maurizio Proietti: Property-Based Test Case Generators for Free. Proceedings of TAP 2019, Lecture Notes in Computer Science 11823: 186–206 (2019)"*. DOI: https://doi.org/10.1007/978-3-030-31157-5_12

allows the automatic, efficient generation of input test values that satisfy a given specification. In particular, we have considered the case when the input values are data structures satisfying complex constraints. That generation is performed via the symbolic execution of the specification using constraint logic programming.

7.1 Introduction

Over the years, Property-Based Testing (PBT), as proposed by Claessen and Hughes [29], has been established as one of the favorite methods for testing software. The idea behind PBT is as follows: instead of supplying specific inputs, i.e., test cases, to a program under test, the developer defines properties to be satisfied by the program inputs and outputs. Then, random inputs are generated and the program is run with those input values, thereby producing output values and checking whether or not the input-output pairs satisfy the desired property. If the output associated with a particular input does not satisfy the property, the counterexample to the property reveals an undesirable behavior. Then, the developer can modify the program under test so that the counterexample is no longer generated. The fact that input values are generated in a random way plays a key role in the PBT techniques, because randomicity enables the generation of valid inputs which originally could have escaped the attention of the developer.

QuickCheck [29] is the first tool that implemented Property-Based Testing and it works for the functional language Haskell. Then, a similar approach has been followed for various programming languages, and among many others, let us mention: (i) Erlang [7, 113], (ii) Java [72, 144], (iii) Scala [123], and (iv) Prolog [4].

In this paper we will focus on the dynamically typed functional programming language Erlang and, in particular, we will refer to the PropEr framework [113, 119]. Typically, in PropEr the set of valid input data is defined through: (i) a type specification, and (ii) a filter specification (that is, a constraint), which should be satisfied by the valid inputs. When working with user-defined types and filters, the developer must provide a *generator*, that is, a program that constructs input data of the given type satisfying the given filter. PropEr supports writing generators by providing a mechanism for turning type specifications into data generators, and also providing primitives for constraining data size and assigning frequencies to guide data generation. However, the task of writing a generator that satisfies all constraints defined by a filter is left to the developer. Unfortunately, writing and maintaining generators is a time-consuming and error-prone activity. In particular, hand-written generators may result in the generation of sets of non-valid inputs or, even worse, sets of inputs

which are too restricted.

In this paper we explore an approach that relieves the developer from the task of writing data generators of valid inputs. We assume that the data generation task is specified by providing: (i) a *data type specification*, using the Erlang language for that purpose, and (ii) a *filter specification* provided by any boolean-valued Erlang function. We have constructed a symbolic interpreter, written in the *Constraint Logic Programming* (CLP) language [69], which takes the data type and the filter specification, and automatically generates data of the given type satisfying the given filter. Our interpreter is *symbolic*, in the sense that it is able to run Erlang programs (in particular, the filter functions) on symbolic values, represented as CLP terms with possibly variable occurrences.

The symbolic interpreter works by exploiting various computational mechanisms which are specific to CLP, such as: (i) *unification*, instead of matching, which enables the use of predicate definitions for generating terms satisfying those predicates, (ii) *constraint solving*, which allows the symbolic computation of sets of data satisfying given constraints, and (iii) *coroutining* between the process of generating the data structures and the process of applying the filter. By using the above mechanisms we realize an efficient, automated data generation process following a *constrain-and-generate* computation pattern, which first generates data structure skeletons with constraints on its elements, and then generates random concrete values satisfying those constraints. Finally, these concrete data are translated back into inputs for the Erlang program under test.

The paper is structured as follows. In Sect. 7.2, we recall some basic notions on the Erlang and CLP programming languages. In Sect. 7.3, we present the framework for Property-Based Testing based on PropEr [113]. In Sect. 7.4, we show how from any given data type definition, written in the type language of Erlang, we derive a CLP generator for such data type. In Sect. 7.5, we describe our CLP interpreter for a sequential fragment of Erlang. In Sect. 7.6, we show the use of coroutining and, in Sect. 7.7, we present some experimental results obtained by the ProSyT tool, which implements our PBT technique. Finally, in Sect. 7.8, we compare our approach with related work in constraint-based testing.

7.2 Preliminaries

In this section we present the basic notions of the Erlang and CLP languages.

The Erlang language. Erlang is a concurrent, higher-order, functional programming language with dynamic, strong typing [5]. Its concurrency is based on the

Actor model [62] and it allows asynchronous communications among processes. These features make the Erlang language suitable for distributed, fault-tolerant, and soft real-time applications. An Erlang program is a sequence of function definitions of the form: $f(X_1, \ldots, X_n)$ -> e, where f is the function name, $X_1, \ldots, X_n$ are variables, and e is an Erlang expression, whose syntax is shown in the box below, together with that of values and patterns. For reasons of simplicity, we have considered a subset of Core Erlang, that is, the intermediate language to which Erlang programs are translated by the Erlang/OTP compiler language [25]. This subset, in particular, does not include letrec expressions, nor commands for raising or catching exceptions, nor primitives for supporting concurrent computations.

$$
\begin{array}{rrcl}
\mathsf{Values} \ni & v & ::= & l \mid c(v_1, \ldots, v_n) \mid \mathsf{fun}\,(X_1, \ldots, X_n) \text{ -> } e \\
\mathsf{Patterns} \ni & p & ::= & p' \text{ when } g \\
& p' & ::= & l \mid X \mid c(X_1, \ldots, X_n) \\
\mathsf{Expressions} \ni & e & ::= & l \mid X \mid f \mid c(e_1, \ldots, e_n) \mid e_0\,(e_1, \ldots, e_n) \mid \mathsf{let}\ X = e_1\ \mathsf{in}\ e \\
& & & \mid \mathsf{case}\ e\ \mathsf{of}\ (p_1 \text{ -> } e_1)\,;\ \ldots\,;\ (p_n \text{ -> } e_n)\ \mathsf{end} \mid \mathsf{fun}\,(X_1, \ldots, X_n) \text{ -> } e
\end{array}
$$

In these syntactic definitions: (i) by '$\mathsf{Values} \ni v$' we mean that v (possibly with subscripts) is a meta-variable ranging over Values, and analogously for Patterns and Expressions, (ii) l ranges over literals (such as integers, floats, atoms, and the empty list [ ]), (iii) c is either the list constructor $[\_|\_]$ or the tuple constructor $\{\_, \ldots, \_\}$, (iv) X (possibly with subscripts) ranges over variables, (v) $\mathsf{fun}\,(X_1, \ldots, X_n)$ -> e denotes an anonymous function (we stipulate that the free variables in the expression e belong to $\{X_1, \ldots, X_n\}$), (vi) g ranges over *guards*, that is, boolean expressions (such as comparisons of terms using =<, ==, etc.) (vii) f ranges over function names.

The evaluation of an expression is performed in the *call-by-value* regime and returns a value. Variables are bound to values via the usual *pattern matching* mechanism. In Erlang each variable is bound to a value only once (this feature is known as *single assignment*). During the evaluation of a function call, the patterns of the `case-of` expression are considered, one after the other, in left-to-right order. The first pattern for which the pattern matching succeeds with a true guard, determines the corresponding expression to be evaluated. If there is no matching pattern with a true guard, a `match_fail` run-time error occurs.

The running example: *a faulty insertion program.* Below we show an Erlang function which is intended to insert an integer `I` in a list `L` of integers sorted in ascending order, thereby producing a new, extended sorted list. That function has an error as

we have indicated in the line ERR. In what follows we will show how to automatically generate input values for detecting that error.

```
insert(I,L) -> case L of
 [] -> [I];
 [X|Xs] when I=<X -> [X,I|Xs];    % ERR: [X,I|Xs] should be [I,X|Xs]
 [X|Xs] -> [X] ++ insert(I,Xs)    % '++' denotes list concatenation
end.
```

Constraint Logic Programming. By CLP(X) we denote the CLP language based on constraints in the domain X, where X is: either (i) FD (the domain of integer numbers belonging to a finite interval), or (ii) R (the domain of floating point numbers), or (iii) B (the domain of boolean values) [69].

A *constraint* is a quantifier free, first-order formula whose variables range over the domain X. A *user-defined predicate* is a predicate symbol not present in the constraint language. An *atom* is an atomic formula $\mathtt{p}(\mathtt{t}_1, ..., \mathtt{t}_k)$, where p is a user-defined predicate and $\mathtt{t}_1, ..., \mathtt{t}_k$ are first-order terms constructed out of constants, variables, and function symbols. A CLP(X) program is a set of *clauses* of the form either `A.` or `A :- c, A1,..., An.`, where `A, A1,..., An` are atoms and `c` is a constraint on the domain X. A *query* is written as `?- c, A1,..., An.` A term, or an atom, is said to be *ground* if it contains no variables. As an example, below we list a CLP(FD) program for computing the `factorial` function ('`#>`' and '`#=`' denote the greater-than and equality relations, respectively):

```
factorial(0,1).
factorial(N,FN) :- N #> 0, M #= N-1, FN #= N*FM, factorial(M,FM).
```

For the operational semantics of CLP(X), we assume that, in the normal execution mode, constraints and atoms in a query are selected from left to right. In Sect. 7.6 we will see how the selection order is altered by using *coroutining* constructs (in particular, by using `when` declarations). When a constraint is selected, it is added to the *constraint store*, which is the conjunction of all constraints derived so far, thereby deriving a new constraint store. Then, the satisfiability of the new store is checked. The search for a clause whose head is unifiable with a given atom is done by following the textual top-down order of the program and, as usual for Prolog systems, the search tree is visited in a depth-first manner.

7.3 A Framework for PBT of Erlang Programs

In this section we introduce the fragment of the PropEr framework developed by Papadakis and Sagonas [113], which we use to specify PBT tasks. PropEr relies on

a set of predefined functions for specifying the properties of interest for the Erlang programs. We consider the following PropEr functions.

- `?FORALL(Xs, XsGen, Prop)`, which is the main function used for property specification. `Xs` is either a variable, or a list of variables, or a tuple of variables. `XsGen` is a *generator* that produces a value for `Xs`. `Prop` is a boolean expression specifying a property that we want to check for the program under test. We assume that `Xs` includes all the free variables occurring in `Prop`. For instance, `?FORALL(X, integer(), mult1(X) >= X)` (i) uses the predefined generator `integer()`, which generates an integer, and (ii) specifies the property `mult1(X) >= X` for the function `mult1(X) -> X*(X+1)`.

- `?LET(Xs, XsGen, InExpr)`, which allows the definition of a *dependent generator*. `Xs` and `XsGen` are like in the `?FORALL` function, and `InExpr` is an expression whose free variables occur in `Xs`. `?LET(Xs, XsGen, InExpr)` generates a value obtained by evaluating `InExpr` using the value of `Xs` produced by `XsGen`. For instance, `?LET(X, integer(), 2*X)` generates an even integer.

- `?SUCHTHAT(Xs, XsGen, Filter)`, which allows the definition of a generator of values satisfying a given *filter* expression. `Xs` and `XsGen` are like in the `?FORALL` function, and `Filter` is a boolean expression whose free variables occur in `Xs`. `?SUCHTHAT(Xs, XsGen, Filter)` generates a value, which is a value of `Xs` produced by `XsGen`, for which the `Filter` expression holds `true`. For instance, `?SUCHTHAT(L, list(integer()), L=/=[])` generates nonempty lists of integers.

In PropEr a generator is specified by using: (i) type expressions, (ii) `?LET` functions, and (iii) `?SUCHTHAT` functions. We consider generators of first-order values only. However, higher-order functions may occur in `Prop`, `InExpr`, and `Filter`.

A *type expression* (whose semantics is a set of first-order values) is defined by using either the following PropEr *predefined types*:

- `any()`: all first-order Erlang values;
- `integer(L,H)`: the integers between `L` and `H` (these bounds can be omitted);
- `float(L,H)`: the floats between `L` and `H` (these bounds can be omitted);
- `atom()`: all Erlang atoms;
- `boolean()`: the boolean values `true` and `false`;

or PropEr *user-defined types*, which are defined by using type parameters and recursion, as usual. For instance, the type of binary trees with elements of a parameter type `T` can be defined as follows:

```
-type tree(T) ::  'leaf' | {'node',tree(T),T,tree(T)}.
```

Compound type expressions can be defined by the following *type constructors*:

- $\{\mathtt{T}_1, \ldots, \mathtt{T}_N\}$: the tuples of `N` elements of types $\mathtt{T}_1, \ldots, \mathtt{T}_N$, respectively;
- `list(T)`: the lists with elements of type `T`;
- $[\mathtt{T}_1, \ldots, \mathtt{T}_N]$: the lists of `N` elements of types $\mathtt{T}_1, \ldots, \mathtt{T}_N$, respectively;
- `union`$([\mathtt{T}_1, \ldots, \mathtt{T}_N])$: all elements x such that x is of type either $\mathtt{T}_1$ or ... or $\mathtt{T}_N$;
- `exactly(lt)`: the singleton consisting of the literal `lt`.

Types can be used for specifying a *contract* [1] for an Erlang function `Func` by writing a declaration of the form:

```
-spec Func(ArgType1, ..., ArgTypeN) -> RetType.
```

A property is specified by declaring a nullary function (whose name, by convention, starts with `prop_`) of the form:

```
prop_name() -> ?FORALL(Xs, XsGen, Prop)
```

Here is an example of a *property specification*, which we will use for testing the `insert` function presented in Sect. 7.2.

```
prop_ordered_insert() ->                                % property_spec
  ?FORALL({E,L}, {integer(),ne_ordered_list()}, ordered(insert(E,L))).
ne_ordered_list() ->                                    % generator_1
  ?SUCHTHAT(L, non_empty(list(integer())), ordered(L)).
non_empty(T) -> ?SUCHTHAT(L, T, L=/=[]).                % generator_2
ordered(L) -> case L of                                 % filter
  [A,B|T] -> A =< B andalso ordered([B|T]);
  _ -> true
end.
```

In order to run the `prop_ordered_insert()` function, PropEr needs an *ad-hoc* implementation of the function `ne_ordered_list()` that generates a *non-empty ordered* list. If such a function is not provided by the user, PropEr executes the `ne_ordered_list()` generator in a very inefficient way by randomly generating non-empty lists of integers until it produces a list which is ordered [113, Sect. 4.2].

[1] More detailed information about types and contract specifications can be found at `http://erlang.org/doc/reference_manual/typespec.html`.

The main contribution of this paper is a technique that relieves the programmer from implementing generator functions and, instead, it derives efficient generators directly from their specifications. By doing so, we mitigate the problem of ensuring that the implementation of the generator is indeed correct, and we also avoid, in most cases, the inefficiency of a generate-and-test behavior by a suitable interleaving (via coroutining) of the process of data structure generation with the process of checking the constraint satisfaction (that is, filter evaluation). The implementation of our technique consists of the following six components.

1. A *translator from PropEr to CLP*, which translates the property specification, together with the definitions of Erlang/PropEr types and functions that are used, to a CLP representation.

2. A *type-based generator*, which implements a CLP predicate `typeof(X,T)` that generates datum `X` of any given (predefined or user-defined) type `T`. `typeof` queries can be run in a symbolic way, thereby computing for `X` a term containing variables, possibly subject to constraints.

3. A *CLP interpreter* for filter functions, that is, functions occurring in filter expressions. The interpreter handles the subset of the Core Erlang language presented in Sect. 7.2. In particular, it defines a predicate `eval(In,Env,Out)` such that, for an Erlang expression `In` whose variables are bound to values in an environment `Env`, `eval` computes, according to the semantics of Erlang, an output expression `Out`. The evaluation of `eval` is performed in a *symbolic* way, as the values in the bindings in `Env` may contain CLP variables, possibly subject to constraints. Thus, by running a query consisting of the conjunction of a `typeof` atom and an `eval` atom, we get as answer a term whose ground instances are values of the desired type, satisfying a given filter.

4. A *value generator*, which takes as input a term produced by running the type-based generator (Component 2) and then the interpreter (Component 3). The value generator can also be run immediately after the type-based generator, if no filter is present. Term variables, if any, may be subject to constraints. Concrete instances of the term (i.e., ground terms) satisfying these constraints are generated by choosing values (in a deterministic or random way) from the domains of the variables.

5. A *translator from CLP to Erlang*, which translates the values produced by the value generator (Component 4) to Erlang values.

6. A *property evaluator*, which evaluates, using the Erlang system, the boolean Erlang expression `Prop` whose inputs are the values produced by the translator (Component 5). Then the property evaluator checks whether or not one of the following three cases occurs: (i) `Prop` holds, (ii) `Prop` does not hold, or (iii) the evaluation of `Prop` crashes, that is, produces a runtime error.

The above six components have been implemented in a fully automatic tool, called ProSyT[2] (Property-Based Symbolic Testing).

7.4 Type-Based Value Generation

Type-based generation (Component 2 of our ProSyT tool) is achieved through the implementation of the `typeof` predicate. Given a type `T`, the predicate `typeof(X,T)` holds iff `X` is a CLP term encoding an Erlang value of type `T`. If `T` is a predefined type, `typeof` invokes a `T`-specific predicate for generating the term `X`. For example, for the type `list(A)`, that is, the type of the lists whose elements are of type `A`, `typeof` is implemented by the following clause:

```
typeof(X,list(A)) :- list(X,A).
```

where the binary predicate `list` is defined by the following two clauses:

```
list(nil,T).
list(cons(X,Xs),T) :- typeof(X,T), list(Xs,T).
```

where `nil` and `cons` are the CLP representations of the Erlang empty list and list constructor, respectively. If `T` is a user-defined type, `typeof` invokes the clause:

```
typeof(X,T) :- typedef(T,D), typeof(X,D).
```

where `typedef(T,D)` holds iff `D` is the (CLP representation of the Erlang) definition of type `T`. The clauses for `typedef` are generated during the translation from Erlang to CLP. For example, for the definition of the type `tree(T)` of binary trees, introduced in Sect. 7.3, we have the following clause:

```
typedef(tree(T), union([
       exactly(lit(atom,leaf)),
       tuple([exactly(lit(atom,node)),tree(T),T,tree(T)]) ])).
```

where: (i) `union([T1,T2])` denotes the union of the two types `T1` and `T2`, (ii) `exactly(E)` denotes a type consisting of the term `E` only, and (iii) `tuple(L)` denotes the type of the tuples $\{t_1, \ldots, t_n\}$ of terms such that t_i has the type specified by the `i`-th element of the list `L` of types.

[2] `https://fmlab.unich.it/testing/`

Apart from the case when the type `T` is `exactly(lit(...))`, the query `?- typeof(X,T)` returns answers of the form `X=t`, where `t` is a non-ground term, whose variables may be subject to constraints. Here follow some examples of use of the `typeof` predicate. If we run the query `?- typeof(X,integer)` we get a single answer of the form `X=lit(int,_1320), _1320 in inf..sup`, where `_1320` is a variable that can take any integer value in the interval `inf..sup`, where `inf` and `sup` denote the minimum and the maximum integer, respectively. We can explicitly specify a range for integers. For instance, the answer to the query `?- typeof(X,integer(10,20))` is `X=lit(int,_1402), _1402 in 10..20`.

The query `?- typeof(X,list(integer))` produces a first answer of the form `X=nil`. If we compute an additional answer for that query, then we get `X=cons(lit(int,_1618), nil), _1618 in inf..sup` denoting the `nil` terminated list containing a single integer value. If we continue asking for additional answers, then by the standard Prolog execution mechanism, based on backtracking and depth-first search, we get answers with lists of increasing length.

When dealing with recursively defined data types, we have to care about the size of the generated terms, with the objective of avoiding the possible non-termination of the evaluation of the `typeof` predicate. The size of a term is defined to be the number of list or tuple constructors occurring in it. Thus, for instance, the term `lit(X,integer)` encoding an integer, has size `0`, and the size of a list of integers is equal to its length. The size of any term generated by `typeof` is constrained to be in the interval `min_size..max_size`, where `min_size` and `max_size` are configurable non-negative integers.

As an alternative to the built-in mechanisms for size management, we also provide the predicate `typeof(X,T,S)` which holds if `X` is a term of type `T` and size `S`. By using specific values of `S` or providing constraints on `S`, the user can specify the term size he desires and can control the answer generation process.

The user can also generate terms of *random* size, instead of terms of increasing size, as obtained by standard Prolog execution. For this purpose, we provide configuration options allowing `typeof` to generate data structures whose size is randomly chosen within the interval `min_size..max_size`.

It is also possible to use randomness during the generation of tuples and unions. For instance, every run of the query `?- typeof(X,tree(integer),2)` using standard Prolog execution, produces the same *first* answer, which is a tree consisting of the root and its right child. In order to modify such a behavior, we have introduced the `random_tuple` option that makes `typeof` generate tuples by randomly choosing one of its elements. (Recall that non-empty trees are indeed defined as tuples.) By doing so, the first answer to the above query is the tree consisting of the root

and either its right child or its left child.

Similarly, for `union` types, we can introduce randomness through the use of the `random_union` option. For example, suppose that the type `color` has the two values `black` and `white` only. Thus, its translation into CLP is as follows:

```
typedef(color,union([exactly('black'),exactly('white')])).
```

Then, if we run the query `?- typeof(X,color)` using the standard Prolog execution mechanism, we will always obtain `black` as the first answer. However, if we use the `random_union` option we may get either `black` or `white` with equal frequency. More in general, we provide a `weighted_union` type, which allows the association of frequencies with types using non-negative integers, so that elements of types with higher frequencies are generated more often.

Random generation of *ground* terms (Component 4 of ProSyT) is achieved through the use of the `rand_elem(X)` predicate. For example, the clauses used for the generation of (possibly, non-flat) lists of integers are the following ones:

```
rand_elem(nil).
rand_elem(cons(X,L)) :- rand_elem(X), rand_elem(L).
rand_elem(lit(int,V)) :- rand_int(V).
rand_int(V) :- int_inf(V,Inf), int_sup(V,Sup), random_between(Inf,Sup,V).
```

where `rand_int(V)` holds iff `V` is a random integer value in the range `Inf..Sup`, `Inf` and `Sup` being the minimum and maximum values that `V` can take, subject to the constraints that are present in the constraint store. For instance, the query:

`?- typeof(X,list(integer(1,10)),3), rand_elem(X).` may return the answer:

```
X = cons(lit(int,6), cons(lit(int,4), cons(lit(int,9), nil))).
```

A similar mechanism is used for generating ground terms containing floats.

Finally, ground CLP terms are translated to Erlang values (Component 5 of ProSyT) by using the `write_elem` predicate. For instance, if we append `write_elem(X)` to the above query, we get the Erlang list `[6,4,9]`.

7.5 The Interpreter of Filter Functions

The CLP interpreter, which is Component 3 of ProSyT, provides the predicate `eval(In,Env,Out)` that computes the output value `Out` of the input expression `In` in the environment `Env`. The environment `Env`, which maps variables to values, is represented by a list of pairs of the form `('X',V)`, where `'X'` is the CLP constant representing the Erlang variable `X` and `V` is the CLP term representing its value. By means of a symbolic representation of Erlang expressions and values occurring in the environment (by using possibly non-ground CLP terms subject to suitable

constraints), the evaluation of any input expression via the interpreter allows the exhaustive exploration of all program computations without explicitly enumerating all the concrete input values.

In the interpreter, a function definition is represented by a CLP term of the form `fun(Pars,Body)`, where `Pars` and `Body` are the formal parameters and the function body, respectively. As an example of how the interpreter is defined, Fig. 7.1 lists the CLP implementation of the semantic rule for function application, represented by a term of the form `apply(Func,IExps)`, where `Func` is the name of the function to be applied to the actual parameters `IExps`.

```
eval(apply(Func,IExps),Env,Out) :-
  fundef(Func,fun(Pars,Body)),        % 1
  eval_args(IExps,Env,AExps),         % 2
  zip_binds(Pars,AExps,Binds),        % 3
  constrain_output_exp(Func,Out),     % 4
  eval(Body,Binds,Out).               % 5
```

Figure 7.1: CLP interpreter for applying the function `Func` to the actual parameters `IExps`.

The behavior is as follows. First, the interpreter retrieves (at line 1) the definition of the function `Func`. Then, it evaluates (at line 2) the actual parameters `IExps` in the environment `Env`, thereby deriving the list of expressions `AExps`. Then, the interpreter binds (at line 3) the formal parameters `Pars` to the actual parameters `AExps`, thereby deriving the new environment `Binds`. If a contract for `Func` has been provided (see Sect. 7.3) and the `--force-spec` option of ProSyT has been used, then (at line 4) a constraint is added on the CLP variables occurring in the output expression `Out`. For instance, let us suppose that the programmer specifies the following contract for the `listlength` function that computes the length of a list:

```
-spec listlength(list(any())) -> non_neg_integer().
```

where the `non_neg_integer()` type requires the output of `listlength` on lists of any type to be a non-negative integer. Thus, the constraint `M#>=0` is imposed on the CLP variable `M` occurring in the output expression `lit(int,M)` computed by `listlength`. Finally, the interpreter evaluates (at line 5), the `Body` of the function `Func` in the new environment `Binds`, thereby deriving the output expression `Out`.

Now, let us consider the filter function `ordered_list` of Sect. 7.3. In order to generate symbolic ordered lists, which will be used for producing the test cases for `insert`, we run the following query:

```
?- typeof(I,non_empty(list(integer))),
   eval(apply(var('ordered',1),[var('L')]),[('L',I)],lit(atom,true)).
```

In the above query `eval` calls `ordered` in an environment where the `'L'` parameter is bound to `I`, and outputs an expression denoting the atom `true`. As a result of query evaluation, `typeof` binds the CLP variable `I` to a nonempty list of integers and

`eval` adds constraints on the elements of the list enforcing them to be in ascending order. Among the first answers to the query we obtain:

```
I = cons(lit(int,_54),cons(lit(int,_55),nil)), _55#>=_54 ;
I = cons(lit(int,_51),cons(lit(int,_52),cons(lit(int,_53),nil))),
    _52#>=_51, _53#>=_52
```

Then, by running the predicates `rand_elem` and `write_elem`, for each non-ground list whose elements are in ascending order, we can (randomly) generate one or more ordered Erlang lists, without backtracking on the generation of lists whose elements are not ordered. The following command runs ProSyT on the file `ord_insert_bug.erl` that includes the bugged `insert` function and the `prop_ordered_insert()` property specification.

```
$ ./prosyt.sh ord_insert_bug.erl prop_ordered_insert
```

By default, ProSyT runs 100 tests by generating non-ground ordered lists of increasing length, which are then instantiated by choosing integers from the interval `-1000..1000`. The 100 tests produce as output a string of 100 characters such as (we show an initial part of that string only):

```
x.x.xxxxxxx...xxxx.xxxxxxx.xxxxx.xxxx..
```

Each character represents the result of performing a test case: (i) the character '`.`' means that the desired property `prop_ordered_insert` holds, and (ii) the character '`x`' means that it does not hold, hence revealing a bug.

The generation of the ordered lists for the 100 test cases takes 42ms (user time) on an Intel® Core™ i7-8550U CPU with 16GB running Ubuntu 18.04.2.

7.6 Coroutining the Type-Based Generator and the Filter Interpreter

The process of symbolic test case generation described in the previous section has a good performance when the filter does not specify constraints on the *skeleton* of the data structure, but only on its *elements*. For instance, in the case of ordered lists, the filter `ordered(L)` does not enforce any constraint on the length of the symbolic list `L` generated by the type-based generator, but only on its elements.

Now, let us consider the following filter function `avl`, which checks whether or not a given binary tree is an AVL tree, that is, a *binary search tree* that satisfies the constraint of being *height-balanced* [31].

```
avl(T) -> case T of
    leaf -> true;
    {node,L,V,R} ->
```

```
      B = height(L)-height(R) andalso B >= -1 andalso B =< 1 andalso  % 1
      ltt(L,V)  andalso  gtt(R,V)  andalso                            % 2
      avl(L)  andalso  avl(R);
     _ -> false
  end.
```

The recursive clause of the `case-of` checks whether or not any tree {`node,L,V,R`} rooted in `V` (the value of the node) is height-balanced (line 1), all the values in the left subtree `L` are smaller than `V`, and all the values in the right subtree `R` are larger than `V` (line 2). `avl` uses the following utility functions: (i) `height(T)`, which returns the height of the tree `T`, (ii) `ltt(T,V)`, and (iii) `gtt(T,V)`, which return `true` if the value of each node n in the tree `T` is less than, or greater than `V`, respectively. In order to generate AVL trees, we run the following query:

```
?- typeof(X,tree(integer)),
   eval(apply(var('avl',1),[var('T')]),[('T',X)],lit(atom,true)),
   rand_elem(X).
```

However, unlike the case of ordered lists, among the answers to the query `typeof(X, tree(integer))` just a few instances of `X` turn out to be AVL trees. Hence, `eval` repeatedly fails until `typeof` generates a binary tree satisfying the constraints specified by the filter. As an example, for trees of size 10, `eval` finds 10 AVL trees out of 9000 trees generated by `typeof`.

In order to make the generation process more efficient, we use the *coroutining* mechanism to implement a data-driven cooperation [76], thereby interleaving the execution of the type-based generator `typeof` and that of the interpreter `eval`. Coroutining is obtained by interchanging the order of the `typeof` and `eval` atoms in the query, so that the `eval` call is selected before the `typeof` call. However, the execution of `eval` is *suspended* on inputs of the filter function that are not instantiated enough to decide which clauses of a `case-of` expression can be used to proceed in the function evaluation. The execution of `eval` is then *resumed* whenever the input to the filter function gets further instantiated by the `typeof` execution. By doing so, during the generation of complex data structures, `typeof` must comply with the constraints enforced by `eval`. This mechanism can dramatically improve efficiency, because the unsatisfiability of the given constraints may be detected before the entire data structure is generated.

Coroutining is implemented by using the `when(Cond,Goal)` primitive provided by SWI-Prolog [131], which suspends the execution of `Goal` until `Cond` becomes true. In particular, `when` declarations are used in the rule of the interpreter shown below, which defines the operational semantics of `case-of` expressions.

```
eval(case(CExps,Cls),Env,Exp) :-
  eval(CExps,Env,EExps),
  suspend_on(Env,EExps,Cls,Cond),
  when(Cond,( match(Env,EExps,Cls,MEnv,Cl), eval(Cl,MEnv,Exp) )).
```

The evaluation of expressions of the form '`case CExps of Cls end.`', encoded as `case(CExps,Cls)`, in the environment `Env` behaves as follows. The expressions `CExps` are evaluated in the environment `Env`, thereby getting the expressions `EExps` to be matched against one of the patterns of the clauses `Cls`. Then, `suspend_on(Env,EExps,Cls,Cond)` generates a condition `Cond` of the form `( nonvar(`V_1`), ..., nonvar(`V_n`) )`, where $V_1, \ldots, V_n$ are the CLP variables occurring in `EExps` that would get bound to either a list or a tuple while matching the expressions `EExps` against the patterns of the clauses `Cls`. Such a condition forces the suspension of the evaluation of the goal occurring as a second argument of the `when` primitive until all of these variables get bound to a non-variable term. If the evaluation of the `case-of` binds all the variables to terms which are neither lists nor tuples, then `suspend_on` produces a `Cond` that holds `true`. Thus, when the goal of the `when` primitive is executed: (i) `match(Env,EExps,Cls,MEnv,Cl)` selects a clause `Cl` from `Cls` whose pattern matches `EExps`, hence producing the environment `MEnv` that extends `Env` with the new bindings derived by matching, and (ii) `eval(Cl,MEnv,Exp)` evaluates `Cl` in `MEnv` and produces the output expression `Exp`. Now, if we run the following query:

```
?- eval(apply(var('avl',1),[var('T')]),[('T',X)],lit(atom,true)),
   typeof(X,tree(integer)),
   rand_elem(X).
```

the CLP variable `X`, shared between `typeof` and `eval`, forces the type-based generator and the filter to cooperate in the generation of AVL trees. Indeed, as soon as the `typeof` (partially) instantiates `X` to a binary tree, the evaluation of the filter function adds constraints on the skeleton of `X` (corresponding to the properties at lines `1` and `2` of the definition of the `avl` function). The advantage of this approach is that the constraints on `X` restrict the possible ways in which its left and right subtrees can be further expanded by recursive calls of `typeof`. As an example, suppose we want to test the following function `avl_insert` that inserts the integer element `E` into the AVL tree `T`:

```
avl_insert(E,T) -> case T of
 {node,L,V,R} when E < V -> re_balance(E,{node,avl_insert(E,L),V,R});
 {node,L,V,R} when E > V ->
   re_balance(E,{node,L,V,avl_insert(E,R)});
```

```
  {node,L,V,R} when E == V -> {node,L,V,R};
  leaf -> {node,leaf,E,leaf}
 end.
```

This function uses the following utility functions shown below: (i) `re_balance`, which given an integer element `E` and a binary search tree `T`, performs suitable rotations on `T` so as to make it height-balanced, and (ii) `right_rotation`, and (iii) `left_rotation`, which perform a right rotation, and a left rotation on `T`, respectively. The definition of `re_balance` has two errors: (1) at line `ERR_1`, where '<' should be '>', and (2) at line `ERR_2`, where '>' should be '<'.

```
re_balance(E,T) ->
 {node,L,V,R} = T,
 case height(L) - height(R) of
   2 -> {node,_,LV,_} = L,              % Left unbalanced tree
   if E < LV -> right_rotation(T);
      E > LV -> right_rotation({node,left_rotation(L),V,R})
   end;
  -2 -> {node,_,RV,_} = R,              % Right unbalanced tree
   if E < RV -> left_rotation(T);                              % ERR_1
      E > RV -> left_rotation({node,L,V,right_rotation(R)})    % ERR_2
   end;
   _ -> T
 end.


right_rotation({node,{node,LL,LV,LR},V,R}) ->
  {node,LL,LV,{node,LR,V,R}}.


left_rotation({node,L,V,{node,RL,RV,RR}}) ->
  {node,{node,L,V,RL},RV,RR}.
```

The following test case specification states that after inserting an integer element `E` in an AVL tree, we get again an AVL tree:

```
 avl() -> ?SUCHTHAT(T, tree(integer()), avl(T)).
 prop_insert() ->
    ?FORALL({E,T}, {integer(),avl()}, avl(avl_insert(E,T))).
```

The following command runs ProSyT on the file `avl_insert_bug.erl` that includes the above bugged `avl_insert` function and the test case specification.

```
$ ./prosyt.sh avl_insert_bug.erl prop_insert --force-spec\
  --min-size 3 --max-size 20 --inf -10000 --sup 10000 --tests 200
```

In this command we have used the following options:
(i) `--min-size` and `--max-size` specify the values of `min_size` and `max_size`, respectively, determining the size of the data structure (see Sect. 7.3), (ii) `--inf` and `--sup` specify the bounds of the interval where integer elements are taken from (see Sect. 7.3), and (iii) `--tests N` specifies the number of tests to be run.

Here is an initial part of the string of characters we get:

```
..x...x...cx..x.xxc....x..c.x..x..x.c..
```

The generation of the 200 test cases takes 550ms (user time). Several '`x`' characters are generated, corresponding to runs of `avl_insert` that do not return an AVL tree, and hence reveal bugs. Moreover, the '`c`' characters in the output string correspond to crashes of the execution, due to the fact that the `right_rotation` or `left_rotation` functions threw a `match_fail` exception when applied to a tree on which those rotations cannot be performed.

7.7 Experimental evaluation

In this section we present the experimental evaluation we have performed for assessing the effectiveness and the efficiency of the test case generation process we have presented in this paper and we have implemented in ProSyT.

Benchmark suite. The suite consists of 10 Erlang programs: (1) `ord_insert`, whose input is an integer and an ordered list of integers (see Sect. 7.2); (2) `up_down_seq`, whose input is a list of integers of the form: $[w_1, \ldots, w_m, z_1, \ldots, z_m]$, with $w_1 \leq \ldots \leq w_m$ and $z_1 \geq \ldots \geq z_m$; (3) `n_up_seqs`, whose input is a list of ordered lists of integers of increasing length; (4) `delete`, whose input is a triple $\langle w, u, v \rangle$ of lists of integers such that w is the ordered permutation of the list obtained by concatenating the ordered lists u and v; (5) `stack`, whose input is a pair $\langle s, n \rangle$, where s is a stack encoded as a list of integers, and n is the length of that list; (6) `matrix_mult`, whose input is a pair of matrices encoded as a pair of lists of lists, (7) `det_tri_matrix`, whose input is a lower triangular matrix encoded as a list of lists of increasing length of the form: $[[v_{11}], [v_{21}, v_{22}], \ldots, [v_{n1}, \ldots, v_{nn}]]$, (8) `balanced_tree`, whose input is a height-balanced binary tree [31]; (9) `binomial_tree_heap`, whose input is a binomial tree satisfying the minimum-heap property [31]; (10) `avl_insert`, whose input is an AVL tree (see Sect. 7.6). The benchmark suite is available online as part of the ProSyT tool (the suffixes `_bug.erl` and `_ok.erl` denotes the buggy and correct versions of the programs, respectively).

Experimental processes. We have implemented the following two experimental processes. (i) *Generate-and-Test*, which runs PropEr for randomly generating a value

Program	PropEr		ProSyT	
	Time	*N*	*Time*	*N*
1. `ord_insert`	300.00	0	300.00	67,083
2. `up_down_seq`	300.00	0	300.00	22,500
3. `n_up_seqs`	300.00	0	300.00	24,000
4. `delete`	300.00	0	9.21	100,000
5. `stack`	143.71	100,000	19.57	100,000
6. `matrix_mult`	300.00	0	300.00	76,810
7. `det_tri_matrix`	300.00	304	32.28	13,500
8. `balanced_tree`	300.00	121	21.54	100,000
9. `binomial_tree_heap`	300.00	0	43.45	4,500
10. `avl_insert`	300.00	0	300.00	23,034

Table 7.1: Column *Time* reports the seconds needed to generate N ($\leq 100{,}000$) test cases of size in the interval $[10, 100]$ within the time limit of 300 seconds.

of the given data type, and then tests whether or not that value satisfies the given filter; this process uses the predefined generator for lists and a simple user-defined generator for trees. (ii) *Constrain-and-Generate*, which runs ProSyT by coroutining the generation of the skeleton of a data structure and the evaluation of the filter expression (see Sect. 7.6), and then randomly instantiating that skeleton.

Technical resources. The experimental evaluation has been performed on a machine equipped with an Intel® Core™ i7-8550U CPU @ 1.80GHz $\times$ 8 processor and 16GB of memory running Ubuntu 18.04.2 LTS. The timeout limit for each run of the test cases generation process has been set to 300 seconds.

Results. We have run PropEr and ProSyT for generating up to $100{,}000$ test cases whose size is in the interval $[10, 100]$, and we made the random generator for integer and real values to take values in the interval $[-10000, 10000]$. ProSyT has been configured so that the random instantiation phase can produce at most 1500 concrete test cases for every data structure skeleton found. The results of the experimental evaluation are summarized in Table 7.1.

The experiments show that the *Constrain-and-Generate* process used by ProSyT performs much better than the *Generate-and-Test* process used by PropEr. Indeed, *Generate-and-Test* is able to find valid test cases only when the filter is very simple. Actually, in some examples, PropEr generates test cases with very small size only (less than the minimum specified size limit of 10). In particular, for the `ord_insert` program, PropEr generates ordered lists of length at most 8, while ProSyT is able

to generate lists of length up to 53. Also for the programs `det_tri_matrix` and `balanced_tree`, the size of the largest data structure generated by PropEr (a 5×5 matrix and a 15 node balanced tree) is much smaller than the largest data structure generated by ProSyT (a 12×12 matrix and a 22 node balanced tree). Finally, note that for the programs `det_tri_matrix` and `binomial_tree_heap`, ProSyT halted before the time limit of 300 seconds because no more skeletons exist within the given size interval.

7.8 Related Work

Automated test generation has been suggested by many authors as a means of reducing the cost of testing and improving its quality [122]. Property-Based Testing, and in particular the QuickCheck approach [29], is one of the most well-established methods for automating test case generation (see also the references cited in the Introduction).

PropEr [113, 119] is a popular Property-Based Testing tool for Erlang programs that follows the QuickCheck approach. PropEr was proposed as an open-source alternative to Quviq QuickCheck [7], a proprietary, closed-source tool for Property-Based Testing of Erlang programs. In addition, PropEr was designed to be integrated with the Erlang type specification language.

However, a critical point of PropEr (and of other PBT frameworks) is that the user bears most of the burden of writing correct, efficient generators of test data. Essentially, PropEr only provides an automated way for converting type specifications into generators of *free* data structures, but very limited support is given to automatically generate data structures subject to *constraints*, such as the sorted lists and AVL-trees we have considered in this paper. In this respect, the main contribution of our work is a technique for the automated generation of test data from PropEr specifications. Indeed, our approach, based on a CLP interpreter for (a subset of) Erlang, allows the automated generation of test data in an efficient way. Test data are generated by interleaving, via coroutining, the data structure generation, the filtering of those data structures based on constraint solving, and the random instantiation of variables. This mechanism, implemented in the ProSyT tool, has demonstrated good efficiency on some non-trivial examples.

The work closest to ours is the one implemented by the FocalTest tool [23]. FocalTest is a PBT tool designed to generate test data for programs and properties written in the functional language Focalize. Its main feature is a translation of Focalize programs into CLP(FD) programs extended with the *constraint combinators* `apply`

and `match`, which encode function application and pattern matching, respectively. `apply` and `match` are implemented by using *freeze* and *wake-up* mechanisms based on the instantiation of logical variables, and in particular, the evaluation of `apply` and `match` is waken up when their arguments are bound to non-variable terms.

A difference between FocalTest and ProSyT comes from the fact that, Focalize is a statically typed language and Erlang is a dynamically typed language. Static type information is used by FocalTest for instantiating variables, while in ProSyT type-based instantiation is performed through the `typeof` data structure generator. Static typing is also exploited in the proof of correctness of FocalTest, whose operational semantics has been formalized in Coq [22]. In contrast, we handle Erlang's dynamic typing discipline by writing an interpreter of (a subset of) the language, which also models failure due to runtime typing errors.

The development of the CLP interpreter for Erlang and, more in general, for the PropEr framework, is indeed the most significant and distinctive feature of our approach. From a methodological point of view, a direct implementation of the operational semantics in a rule-based language like CLP, requires a limited effort for the proof of correctness with respect to a formal semantics (we did not deal with this issue in the present paper, but we tested our interpreter on several examples). From a practical point of view, the use of the interpreter avoids the need of extending CLP with special purpose constraint operators like `apply` and `match`. Moreover, our interpreter-based approach lends itself to possible optimizations for improving the efficiency of test case generation, such as *partial evaluation* [71], for automatically deriving specialized test case generators.

The freeze and wake-up mechanisms used by FocalTest are quite related to the coroutining mechanism implemented by ProSyT, which, however, is realized by the interpreter, rather than by the constraint solving strategy.

Other differences between FocalTest and ProSyT concern numerical variables and random instantiation. FocalTest handles integer numerical variables using the CLP(FD) constraint solver and randomly instantiates those variables using a strategy called *random iterative domain splitting*. ProSyT handles integer and float numerical variables using CLP(FD) and CLP(R), respectively, for solving constraints on those variables, and their random instantiation is done by using CLP(FD) and CLP(R) built-ins. ProSyT is also able to perform random generation of data structures, by using a randomized version of the predicate `typeof` (see Sect. 7.4), possibly specifying a distribution for the values of a given type.

The idea of interleaving, via coroutining, the generation of a data structure with the test of consistency of the constraints that the data structure should satisfy, is related to the lazy evaluation strategy used by *Lazy SmallCheck* [121], a PBT tool for

Haskell. Lazy SmallCheck checks properties for partially defined values and lazily evaluates parallel conjunction to enable early pruning of the set of candidate test data. Lazy SmallCheck does not use symbolic constraint solving, and exhaustively generates all values up to a given bound.

Besides functional programming languages, PBT has also been applied to Prolog [4]. Similarly to ProSyT, the PrologCheck tool [4] implements randomized test data generation for Prolog. However, when the test data specification contains constraints, PrologCheck follows a *generate-and-test* approach, and no mechanism is provided by the tool for coroutining data generation and constraint solving (unless this is coded directly by the programmer).

The use of constraint-based reasoning techniques for test-case generation is a well-established approach [38, 58, 60, 94], which has been followed for the implementation of several tools in various contexts. Among them, we would like to mention: GATeL [95], a test sequence generator for the synchronous dataflow language LUSTRE, AUTOFOCUS [118], a model-based test sequence generator for reactive systems, JML-TT [15], a tool for model-based test case generation from specifications written in the Java Modeling Language (JML), Euclide [59], a tool for testing safety-critical C programs, and finally, tools for *concolic testing*, such as PathCrawler [141], CUTE [125], and DART [56], which combine concrete execution with constraint-based path representations of C programs.

Our work is also related to approaches and tools proposed in the context of languages for specifying and testing meta-logic properties of formal systems. In particular, αCheck [27] follows an approach very much inspired by PBT for performing bounded model-checking of formal systems specified in αProlog, which is a Horn clause language based on nominal logic. Related concepts are at the basis of QuickChick, a PBT tool for the Coq proof assistant [114]. Lampropoulos et al. [79] also address the problem of deriving correct generators for data satisfying suitable inductive invariants on top of QuickChick. In that work, the mechanism for data generation makes use of the narrowing technique, which similarly to our resolution-based approach, builds upon the unification algorithm.

Declarative approaches for test data generation have been proposed in the context of *bounded-exhaustive testing* (BET) [30], whose goal is to test a program on all input data satisfying a given invariant, up to a fixed bound on their size. One of the most well-known declarative frameworks for BET is Korat [16], which is a tool for testing Java programs. Given a Java predicate specifying a data structure, subject to a given invariant and a size bound on its input, Korat uses backtracking to systematically explore the bounded input space of the predicate by applying a generate-and-test strategy. JMLAutoTest [142] implements a technique, based on

statistical methods, for avoiding the generation of many useless test cases by exploiting JML specifications.

A different domain-specific language for BET of Java programs is UDITA [55]. It provides non-deterministic choice operators and an interface for generating linked structures. UDITA improves efficiency with respect to the generate-and-test approach by applying the *delayed choice* principle, that is, postponing the instantiation of a variable until it is first accessed.

It has been shown that CLP-based approaches, which exploit built-in unification and special purpose constraint solving algorithms, can be very competitive with respect to domain-specific tools for BET [44].

7.9 Conclusions

We have presented a technique for automated test case generation from *test case specifications*. We have considered the Erlang functional programming language and a test case specification language based on the PropEr framework [113, 119].

In this paper we have shown how we can relieve the programmer from writing generators of test data by using constraint logic programming (CLP). However, even if our approach to automated PBT is based on CLP, the programmer is not required to deal with any concept related to logic programming, and Prolog code is fully transparent to the programmer. Indeed, we provide both (i) a translator from PropEr and Erlang specifications and programs to CLP, and (ii) a translator of the generated test data from CLP syntax to Erlang syntax.

At present, the ProSyT tool, which implements our PBT technique, does not provide any *shrinking* mechanism to try to generate an input of *minimal* size in case the program under test does not satisfy the property of interest. However, we think that this mechanism can efficiently be realized by using the primitives for controlling term size provided by our tool, together with Prolog default search strategy based on backtracking.

Finally, we would like to notice that, even if we developed our technique in the context of PBT of Erlang programs, the approach we followed is to a large extent independent of the specific programming language, as it is based on writing a CLP interpreter of the programming language under consideration. Thus, as future work, we plan to apply a similar scheme to other programming languages by providing suitable CLP interpreters.

Acknowledgments

We would like to thank the anonymous reviewers for their very helpful and constructive comments.

Part III

General Discussion of Results

Chapter 8

Discussion

In this chapter, we discuss the main results achieved in this thesis. We have split this discussion into three categories, corresponding to the main topics covered in our work. In particular, Section 8.1 summarizes the (mostly, theoretical) results on reversible term rewriting. Section 8.2 discusses the combination of theoretical and practical results achieved in the context of reversible debugging (reversibility theory, causal-consistent reversible debugging and causal-consistent replay debugging). Finally, in Section 8.3 we discuss our results in the area of constraint-based testing, namely concolic testing and property-based testing.

8.1 Reversible Term Rewriting

The beginning of Chapter 2 presents a conservative extension of term rewriting that is reversible. First, we define reversible term rewriting for the case of unconditional term rewriting, which is conceptually simpler and helps to understand the main attributes of our approach. Then, we extend this notion and results to DCTRSs. Here, we prove theorems 2.5 and 2.17 which state that reversible rewriting is a conservative extension of standard rewriting for both cases. Then, we prove the reversibility of our reversible forward relation $\rightharpoonup_{\mathcal{R}}$ (stated in theorems 2.9 and 2.20) and the fact that our reversible backward relation $\leftharpoondown_{\mathcal{R}}$ is deterministic and confluent (stated in theorems 2.11 and 2.21). Termination is trivially guaranteed since the trace length decreases with every backward step.

Then, we consider two refinements of our approach. The first one is an improvement that reduces the size of the traces by removing positions from traces

(Section 2.4). We introduce a subclass of DCTRs, namely pcDCTRSs, which makes storing positions unnecessary since reduction in pcDCTRSs only takes place at top-most reductions. Then, we propose a set of transformations that we prove to be correct for transforming a DCTRS into a pcDCTRS: flattening (theorem 2.25) and simplification of constructor conditions (theorems 2.26 and 2.27). We conclude this section with a proof for theorem 2.28 which basically states that innermost and top reductions are equivalent for pcDCTRSs.

The second refinement, described in Section 2.5, aims at compiling the reversible extension of rewriting into the system rules. Given a pcDCTRS $\mathcal{R}$, we aim at producing two new systems $\mathcal{R}_f$ and $\mathcal{R}_b$ such that standard rewriting in $\mathcal{R}_f$ (i.e., $\to_{\mathcal{R}_f}$), is equivalent to the forward reversible extension $\rightharpoonup_{\mathcal{R}}$ in the original system, and analogously $\to_{\mathcal{R}_b}$ is equivalent to the backward reversible extension $\leftharpoondown_{\mathcal{R}}$. The correctness of our transformations (with respect to the original reversible relation) is proven for theorem 2.33 (injectivization) and theorem 2.36 (inversion). Moreover, we present an improvement of the injectivization transformation based on an injectivity analysis for removing labels in trace terms associated with injective functions.

In Section 2.6, we show the application of our reversibilization technique to solve the view-update problem. Basically, we produce a bidirectional transformation by means of our injectivization and inversion transformations. We note that the example considered in [97] cannot be handled by our approach. However, an additional transformation from pcDCTRSs to functional programs with treeless functions could allow us to apply the technique in [97]. In summary, our approach can solve the view-update problem as long as the view function can be encoded in a pcDCTRS. Recently, [112] introduced a syntactic bidirectionalization technique based on a combination of our approach with other approaches in syntactic bidirectionalization [97] and semantic bidirectionalization [140]. In fact, we expect our approach to be applied on more occasions in the context of bidirectional program transformation, as well as in other contexts like reversible debugging or parallel discrete event simulation.

Finally, we present an implementation of the reversibilization transformations introduced in Section 2.5. The tool can read an input TRS file [1] and then it applies sequentially the following transformations: flattening, simplification of constructor conditions, injectivization, and inversion. The tool prints out the CTRSs obtained at each transformation step, and it is publicly available through a web interface from `http://kaz.dsic.upv.es/rev-rewriting.html`, where we have included a number of examples to easily test the tool.

8.2 Reversible Debugging

In Section 3.4 we propose an uncontrolled version of a reversible semantics for Erlang. We prove the reversible forward semantics to be a conservative extension of the standard semantics (stated in theorem 3.8). Moreover, we prove other properties of this semantics: the loop lemma (lemma 3.11), the square lemma (lemma 3.13), the switching lemma[1] (lemma 3.16), the rearranging lemma (lemma 3.18) and the shortening lemma (lemma 3.20). These lemmas are then used to prove causal consistency of the reversible semantics (theorem 3.21).

In Section 3.5, we propose a rollback semantics for Erlang. We first prove the soundness of the rollback semantics as stated in theorem 3.25, and then its completeness (lemma 3.26).

Finally, Section 3.6 presents a proof-of-concept implementation of the uncontrolled reversible semantics for Erlang (Section 3.3). The implementation is bundled together with a graphical user interface (shown in Figure 8.1) in order to facilitate the interaction of users with the reversible semantics (in the following, we refer to this as "the application"). Nonetheless, the application was developed in a modular way so that parts of it could be included in other projects.

The implementation of our reversible semantics considers a language equivalent to a subset of Core Erlang. In short, Core Erlang is an intermediate language used by the Erlang compiler. Generally, Erlang programs are translated to Core Erlang by the Erlang/OTP system before their compilation, so that the resulting code is simplified. For instance, pattern matching can occur almost anywhere in an Erlang program, whereas in Core Erlang, pattern matching can only occur in case statements. Nevertheless, we remark that our implementation considers Core Erlang code translated from the Erlang programs provided by the user.

When the application is started, the first step is to select an Erlang source file. The source file is then translated to Core Erlang, and the resulting code is shown in the Code window. Then, the user is able to select any of the functions from the module and write the arguments that she wants to evaluate the function with. When the START button is pressed, an initial system (composed of an empty global mailbox and a single process performing the specified function application) appears on the State window (shown in Figure 8.1). At this point, the user is able to control the evaluation of the system by selecting the rules from the reversible semantics that she wants to fire.

The application includes two different modes for controlling the evaluation of

[1] Not included in other proof schemes [35], but necessary in our case.

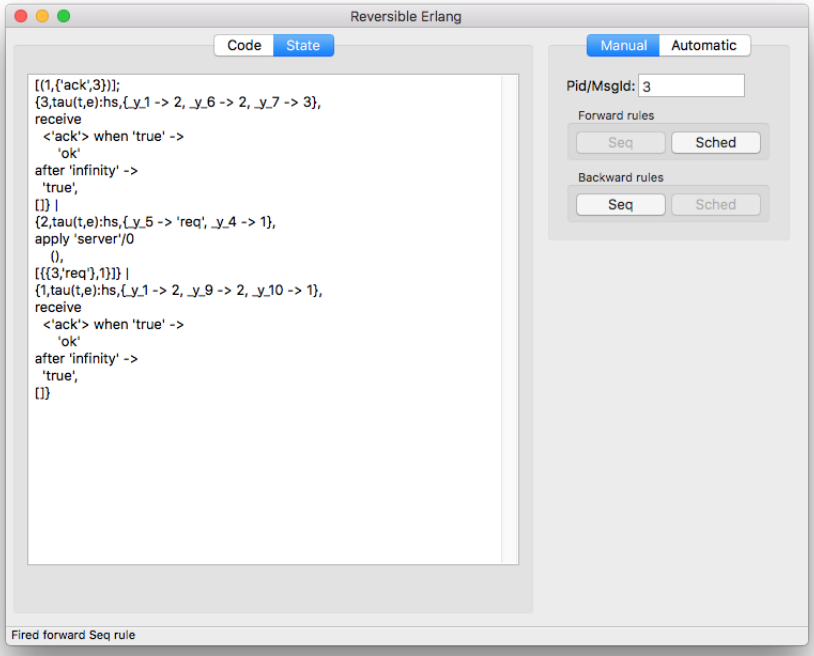


Figure 8.1: Screenshot of the application

the reversible semantics. The first one is a Manual mode, where the user selects the rule to be fired for a particular process or message. Here, the user is in charge of controlling the reversible semantics in a stepwise manner, although this mode can become exhausting after some time. The second mode is the Automatic mode. Here, the user specifies a number of steps and chooses a direction (i.e., forward or backward). Then, the rules to be applied are randomly selected—for the chosen direction—until the specified number is reached or no more rules can be applied. Alternatively, the user can move the state forward up to a normalized system. A normalized system is reached when no rule other than *Sched* can be applied. Hence, in a normalized system, either all processes are blocked (waiting for some message to arrive) or the system state is final. This option allows the user to perform all reductions that do not depend on the network. Then, reductions that depend on the network can be applied one by one for easier understanding of their impact on the computation.

An updated release version (v1.1, with fixes and GUI improvements) of the application is publicly available from `https://github.com/mistupv/rev-erlang` under the MIT license. The linked repository also includes some documentation and a few examples to easily test the application. Furthermore, the application is completely written in Erlang, therefore the only requirement to build it is to have Erlang/OTP installed in your system and built with wxWidgets.

In Chapter 4, we first define a new controlled semantics which is more appropriate for a causal-consistent reversible debugger than the one in Chapter 3.

In Section 4.4, we present CauDEr, our causal-consistent reversible debugger for Erlang. In the same way that the implementation described earlier, CauDEr is conveniently bundled together with a graphical user interface (shown in Figure 8.2). The release version (v1.0) of CauDEr is publicly available from `https://github.com/mistupv/cauder` under the MIT license. As in the previous implementation, the only requirement is to have Erlang/OTP installed and built with wxWidgets, and we include documentation and examples for testing it.

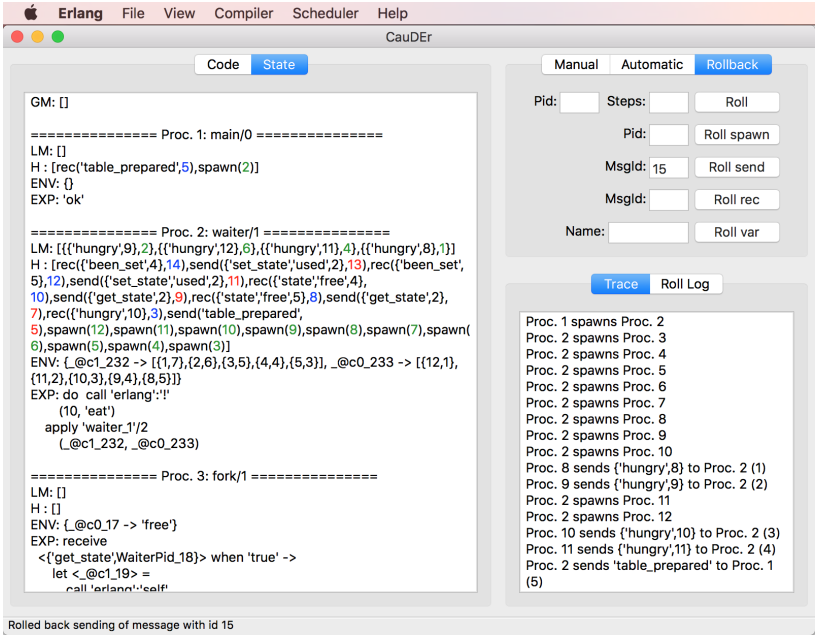


Figure 8.2: CauDEr screenshot

Although CauDEr is heavily based on the proof-of-concept implementation described in Section 3.6, we add a number of features that make CauDEr more sophis-

ticated when it comes to debugging in Erlang. In the following, we summarize the extra features of CauDEr with respect to the previous implementation:

- The reversible semantics embedded in CauDEr allows one to focus on undoing the actions of a specific process. In contrast, the previous rollback semantics targeted a particular checkpoint—introduced by the user. Moreover, this allows us to define several rollback operators for undoing:
  - the sending of a message with a particular identifier,
  - the receiving of a message with a particular identifier,
  - the spawning of a process with a particular pid, or
  - the introduction of a binding for a given variable.[2]

  Hence, the new reversible semantics is more useful for debugging because it allows one to focus on undoing specific actions, instead of undoing actions up to a checkpoint introduced by the user before the debugging process.
- The option to choose between two random schedulers in the Automatic mode:
  - a scheduler with a uniform distribution, and
  - a scheduler which gives priority to process actions (as in the normalization strategy described in Section 3.6).

  Of course, none of these schedulers is able to replicate the behavior of the Erlang/OTP scheduler (since it depends on many factors), but the normalization strategy allows the user to focus on actions that depend on the network (sending and receiving).
- An additional Rollback mode that lets users start off any of the rollback operators previously discussed.
- A large number of GUI improvements to the State tab, where the system state can be inspected. In particular, we added these features:
  - Message and process identifiers are highlighted in color.
  - The option to hide any component of the process representation (local mailbox, environment, history or expression).
  - The option to show all actions or just concurrent actions in histories.

[2] renaming in CauDEr enforces unique variable names

  - The option to show all variable bindings or just relevant ones[3] in environments.

- A new Trace tab which gives a linearized description of the concurrent actions performed in the system. This tab aims at providing a global picture of the system state, possibly highlighting anomalies in process communications.

- A further Roll Log tab which is updated in case of rollbacks. Basically, it shows which actions have been undone upon a rollback request. This tab allows the user to understand the causal dependencies of the process targeted by the rollback request, frequently highlighting undesired or missing dependencies caused by a bug.

Then, we show how to use our debugger to find concurrency bugs in the well-know problem of the dining philosophers (Erlang code available from `https://github.com/mistupv/dining-philos`). First, we describe the usage of our debugger in the context of a message order violation scenario. We note that it would not be easy to find the same bug using a standard debugger (i.e., using breakpoints) or a reversible debugger (like Actoverse [126]). In fact, the CauDEr facility for rollbacking a particular message receiving, in addition to unique identifiers for messages, is shown to be key in ensuring the localization of this concurrency bug. Then, we also describe the usage of our debugger in the context of a livelock scenario. Livelocks are hard-to-find bugs since no global progress is achieved but local progress (between a few processes) keeps the program alive. Again, the rollback facilities of CauDEr are essential in finding the present bug. We note that this kind of bugs are typically hard to find with other debugging tools. For instance, it is not possible to use Concuerror [61] since it requires a finite computation.

In Chapter 5, we first introduce the notion of logged computations. We provide a logging semantics and discuss that, in a practical implementation, one should aim at performing a program instrumentation so that the execution of the program in an actual environment produces the sequence of events required for replay debugging.

Then, we introduce an uncontrolled version of causal-consistent replay semantics in Section 5.4. We state the correctness and completeness of our semantics in theorem 5.18, which ensures that a misbehavior occurring in logged computation is replayed in any possible fully-logged derivation.

In Section 5.5, we propose a controlled version of the replay semantics. In contrast to the uncontrolled version, which allows one to replay a given derivation and

[3]We consider a variable binding to be relevant for a particular expression if the variable occurs within the expression

be guaranteed to replay eventually any local misbehavior, this semantics allows one to focus on a specific process or even some of its actions.

Finally, the implementation of a logger and an implementation of a causal-consistent replay debugger following the ideas in Chapter 5 (based on CauDEr) can be found in [87]. More details and results related to Chapter 5 can be found in [88].

8.3 Constraint-based Testing

In Chapter 6, we first introduce an instrumented semantics for Erlang so that the execution of an Erlang program with this semantics produces a sequence of events. Then, we present a program instrumentation for generating the same sequence of events with the instrumented program during standard execution in any environment. Moreover, the instrumentation is proved to be equivalent to running the original program with the instrumented semantics (theorem 6.8). Then, we formalize a procedure using Prolog for reconstructing a symbolic execution that mimicks the concrete execution corresponding to the generated sequence of events. The implementation of the instrumenting program is made in Erlang and can be tested at `http://kaz.dsic.upv.es/instrument.html`.

In Chapter 7, we first describe the core components of our tool for automated test case generation: Section 7.4 for type-based value generation (second component), Section 7.5 for the interpreter of filter functions (third component), and Section 7.6 for coroutining, the mechanism that interleaves the previous components for more efficient results.

Finally, in Section 7.7 we present an experimental evaluation to assess the effectiveness and efficiency of the test case generation method proposed in Chapter 7 and implemented in ProSyT, which is publicly available from: `https://fmlab.unich.it/testing/`. This evaluation compares our tool against PropEr, a popular property-based testing tool for Erlang.

The benchmark suite consists of 10 different Erlang programs (more details in Section 7.7) and is available online as part of the ProSyT tool. We implement two experimental processes:

Generate-and-Test Runs PropEr for randomly generating a value of the given data type, and then tests whether or not that value satisfies the given filter; this process uses the predefined generator for lists and a simple user-defined generator for trees.

Constrain-and-Generate Runs ProSyT by coroutining the generation of the skele-

Program	PropEr		ProSyT	
	Time	*N*	*Time*	*N*
1. `ord_insert`	300.00	0	300.00	67,083
2. `up_down_seq`	300.00	0	300.00	22,500
3. `n_up_seqs`	300.00	0	300.00	24,000
4. `delete`	300.00	0	9.21	100,000
5. `stack`	143.71	100,000	19.57	100,000
6. `matrix_mult`	300.00	0	300.00	76,810
7. `det_tri_matrix`	300.00	304	32.28	13,500
8. `balanced_tree`	300.00	121	21.54	100,000
9. `binomial_tree_heap`	300.00	0	43.45	4,500
10. `avl_insert`	300.00	0	300.00	23,034

Table 8.1: Column *Time* reports the seconds needed to generate N ($\leq 100{,}000$) test cases of size in the interval $[10, 100]$ within the time limit of 300 seconds.

ton of a data structure and the evaluation of the filter expression (see Section 7.6), and then randomly instantiating that skeleton.

PropEr and ProSyT were run for generating up to $100{,}000$ test cases of size within the interval $[10, 100]$, and the random generator for integer and real values were set to take values in the interval $[-10000, 10000]$. The timeout limit for each run of the test case generation process was set to 300 seconds.

The results of the experimental evaluation are summarized in Table 8.1 (data about test case size not shown). The experiments confirm that the *Constrain-and-Generate* process used by ProSyT performs much better than the *Generate-and-Test* process used by PropEr, both in terms of efficiency and test case quality. In fact, the *Generate-and-Test* strategy followed by PropEr is able to find valid test cases only when the filter is quite simple and, even in those cases, it generates test cases of small size (less than the specified minimum size of 10). For instance, for the `ord_insert` program, PropEr generates ordered lists of at most 8 elements, while ProSyT is able to generate lists of up to 53 elements. Similarly, for programs `det_tri_matrix` and `balanced_tree`, the size of the largest data structure generated by ProSyT (a 12×12 matrix and a 22 node balanced tree) is much bigger than the one generated by PropEr (a 5×5 matrix and a 15 node balanced tree). Finally, note that for some programs (e.g., `det_tri_matrix`), ProSyT halted before the time limit of 300 seconds because

no more skeletons exist within the given size interval.

Part IV

Conclusions

Chapter 9

Conclusions and Future Work

In this thesis, we have proposed several techniques based on formal methods for constraint-based testing and reversible debugging in Erlang. In particular, we have designed methods for concolic testing, property-based testing, causal-consistent reversible debugging and causal-consistent replay debugging of Erlang programs. Moreover, we have provided formal proofs for the most interesting properties of our proposals, in addition to publicly-available software tools that experimentally show these approaches to be feasible and efficient in practice. In this chapter, we perform a final review of our contributions and we present a discussion about future work.

9.1 Conclusions

Below we highlight the main contributions in this thesis.

- In Chapter 2, we introduced a conservative extension of term rewriting that is reversible. First, our approach is presented for the case of unconditional term rewriting for easier understanding. We then extended this notion to the more general case of DCTRSs, and proved the soundness and reversibility of our extension of rewriting. Then, in order to introduce a reversibilization transformation for these systems, we also presented a transformation from DCTRSs to pure constructor systems (pcDCTRSs) which is correct for constructor reduction. A further improvement is presented for injective functions, which may have a significant impact in memory usage in some cases.

Then, we showed how to successfully apply our approach in the context of bidirectional program transformation. Finally, we developed a prototype implementation of the reversibilization transformations that was made publicly available through a web interface.

- In Chapter 3, we first introduced an uncontrolled version of a reversible semantics for Erlang, and we proved its most interesting properties (loop lemma, square lemma and causal consistency). Then, we proposed a controlled version of the backward semantics that can be used to model a rollback operator which undoes the actions of a process up to a given checkpoint. Finally, we presented a proof-of-concept implementation of the reversible semantics that shows our approach to be feasible in practice.

- In Chapter 4, we presented the design of CauDEr, a causal-consistent reversible debugger for Erlang. The tool is based on the reversible semantics introduced in Chapter 3, though we introduced in Chapter 4 a new rollback semantics especially tailored for reversible debugging. We showed its application in two different scenarios to demonstrate that some bugs can be more easily located using our tool, thus filling a gap in the collection of debugging tools for Erlang.

- In Chapter 5, we introduced causal-consistent replay, a concept that is strongly related to causal-consistent reversibility, and its instance on debugging (causal-consistent reversible debugging). We proposed both an uncontrolled and a controlled semantics for causal-consistent replay in Erlang. Then, CauDEr was adapted to work with controlled version of the replay semantics, resulting in the implementation of a causal-consistent replay debugger for Erlang.

- In Chapter 6, we introduced a transformational approach to concolic execution based on flattening and instrumenting the source program. The execution of the instrumented program generates a sequence of events that can be easily processed in order to compute the concrete values of the associated symbolic execution, as well as possible alternatives—to produce new test cases. In contrast to interpreter-based approaches, the main advantage is that the instrumented program can be run in any environment, including non-standard ones. This allows one to run the instrumented program in, for instance, a model checking environment like Concuerror [61] so that its execution would produce the sequences of events for all relevant interleavings. Therefore, it might be useful to combine concolic execution with other techniques (e.g., model checking).

- In Chapter 7, we presented a technique for automated test case generation in Erlang. Our technique relieves the user from writing data generators, a well-known drawback of property-based testing. The technique was implemented in ProSyT, a tool composed of six modules, that takes both a data type and a filter specification (written in specification language of the PropEr framework [113, 119]) and generates concrete data of the given type that satisfies the given filter. The core component of ProSyT is a symbolic interpreter that exploits various mechanisms which are specific to constraint logic programming (unification, constraint solving and coroutining). Nonetheless, the user is not required to have prior knowledge of constraint logic programming in order to use ProSyT.

We conclude this section with a discussion about how these works are related. As mentioned in Chapter 2, we expected our work on reversible term rewriting to be useful in the context of reversible debugging. In fact, we simply applied a Landauer embedding in the reversible semantics proposed in Chapter 3, which was enough to deal with the sequential expressions of the language. On the other hand, its concurrent features required us to ensure causal consistency in order to guarantee the reversibility of this semantics. Then, in Chapter 4 we introduced a new rollback semantics particularly intended for reversible debugging, which was clearly inspired by the reversible semantics from Chapter 3. Moreover, we exploited the proof-of-concept implementation introduced in Chapter 3 for the development of CauDEr, the causal-consistent reversible debugger presented in Chapter 4. Later, we realized that we could improve the usage of our reversible debugger by adding a replay mechanism to it. Therefore, we introduced causal-consistent replay debugging in Chapter 5, which is very related to the notion of causal-consistent reversibility (Chapter 3) and causal-consistent reversible debugging (Chapter 4). Not surprisingly, we used CauDEr as the starting point of our implementation for causal-consistent replay debugging. Finally, although the work presented in Chapter 6 did not have a direct impact on Chapter 7, we note that the experience acquired in the development of our concolic execution method was quite beneficial for the design and implementation of ProSyT, the automated test case generation tool presented in Chapter 7.

9.2 Future Work

In this section, we discuss promising directions for future research along the lines of our work. As in Chapter 8, we have split the discussion into a few categories corresponding to the main topics covered in this thesis.

9.2.1 Reversible Term Rewriting

The first branch for future work to consider in reversible term rewriting is to research new methods to further reduce the size of traces. In principle, it would be interesting to define a reachability analysis for DCTRSs. In fact, a reachability analysis for CTRSs without extra variables can be found in [42], but the extension to deal with extra variables in DCTRSs seems challenging. Furthermore, as mentioned in Section 2.5.1, a completion procedure to add default cases to some functions may help to broaden the applicability of the technique and avoid the restriction to constructor reduction. Also, we have proved our reversibilization transformations to be correct with respect to innermost reduction, but it would be interesting to extend these results to other reduction strategies. Finally, we could explore the applicability of our approach in new contexts, as we have done in Section 2.6.

9.2.2 Reversible Debugging

An interesting addition to our work on reversibility would be to define a mechanism to control it so that history information is stored only when we expect to perform a rollback. This feature would allow us to extend Erlang with a new construct for safe sessions, where all actions can be undone if the session aborts. In fact, such a construct could greatly improve the fault-tolerance capabilities of Erlang.

Regarding our causal-consistent reversible debugger CauDEr, we consider the following topics for future work:

- After undoing some steps in a backward computation, we can resume the forward computation, but there is no guarantee that we will reproduce the previous forward steps. Some debuggers (so-called omniscient or back-in-time debuggers) allow us to move both forward and backward along a particular execution. Here, we could define a similar approach for moving along forward causal-consistent steps in the same way that we defined a causal-consistent notion for replay debugging. Such an approach might be useful to determine which processes depend on particular computation step, thereby easing the localization of bugs.

- Instead of debugging systems as a whole, we could develop a fully distributed debugger where each process is equipped with debugging facilities. This would greatly improve the scalability of our debugger, since most of the computational effort would be distributed. However, this approach would require a semantics without any synchronous interaction. For instance, rules $\overline{Send2}$

and $\overline{Spawn2}$ of the rollback debugging semantics (Figure 4.5) should be replaced by a more complex asynchronous protocol.

- An approach to record & replay for languages based on the actor model is introduced in [8]. In our work, we concentrate on the theory, while they focus on low-level issues (dealing with I/O, producing compact logs, etc.). We could consider some of the ideas in [8] and apply them in our work to reduce the size of logged computations and our instrumentation overhead.

Nevertheless, there are two major improvements for CauDEr that we could consider in order to achieve its adoption in industrial environments. As mentioned earlier, CauDEr translates input programs to Core Erlang, a much simpler language than Erlang. Hence, there is not a direct correspondence between the program used as input and the program being shown in CauDEr. This confuses users and complicates the process of debugging. To this end, we could redesign CauDEr so that programs are not translated to Core Erlang. However, this would require us to formalize a much more complicated semantics for Erlang. Another problem is that the system state representation of our debugger is text-based, but we could replace this interface with a visual one more appropriate for debugging. For instance, we could make use of the visualization engine from [14], which would greatly help to identify causal dependencies thanks to the display of happens-before relations. However, we recognize that both improvements would require a huge implementation effort.

9.2.3 Constraint-Based Testing

Regarding our approach to concolic execution, we could extend our approach in order to cover a larger subset of Erlang. However, we consider that the main drawback of our technique is that it is not fully automatic (currently, one should run the instrumented program and then the Prolog procedure for generating alternative bindings). Therefore, we plan to design an automated tool for test case generation by connecting both components. Here, we expect our transformational approach to be useful to cope with concurrent programs.

Regarding the ProSyT tool, we consider to add a shrinking mechanism which allows us to generate an input of minimal size as a counterexample to the property specified by the user. This mechanism could be realized by using the primitives for controlling term size in our tool, together with Prolog default search strategy based on backtracking.

Finally, it is worth noticing that, even though our technique has been developed in the context of property-based testing of Erlang programs, the approach we fol-

lowed is quite independent of the specific programming language since it is based on writing a constraint logic programming interpreter of the language under consideration. Therefore, we could apply a similar approach to other programming languages with suitable interpreters.

Bibliography

[1] Annual international termination competition. URL: `http://www.termination-portal.org/wiki/Termination_Competition`.

[2] S. Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 347(3):441–464, 2005.

[3] J. M. Almendros-Jiménez and G. Vidal. Automatic partial inversion of inductively sequential functions. In Z. Horváth, V. Zsók, and A. Butterfield, editors, *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages (IFL 2006)*, volume 4449 of *Lecture Notes in Computer Science*, pages 253–270. Springer, 2006.

[4] C. Amaral, M. Florido, and V. S. Costa. PrologCheck – Property-Based Testing in Prolog. In M. Codish and E. Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4–6, 2014*, Lecture Notes in Computer Science 8475, pages 1–17. Springer, 2014.

[5] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in Erlang (2nd edition).* Prentice Hall, 1996.

[6] J. Armstrong, R. Virding, and M. Williams. *Concurrent programming in ERLANG.* Prentice Hall, 1993.

[7] T. Arts, J. Hughes, J. Johansson, and U. T. Wiger. Testing telecoms software with Quviq QuickCheck. In M. Feeley and P. W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, pages 2–10. ACM, 2006.

[8] D. Aumayr, S. Marr, C. Béra, E. G. Boix, and H. Mössenböck. Efficient and deterministic record & replay for actor languages. In E. Tilevich and H. Mössenböck, editors, *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang 2018)*, pages 15:1–15:14. ACM, 2018.

[9] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[10] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

[11] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17:525–532, 1973.

[12] C. H. Bennett. Notes on the history of reversible computation. *IBM Journal of Research and Development*, 44(1):270–278, 2000.

[13] J. Bergstra and J. Klop. Conditional Rewrite Rules: confluence and termination. *Journal of Computer and System Sciences*, 32:323–362, 1986.

[14] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging distributed systems: Challenges and options for validation and debugging. *Communications of the ACM*, 59(8):32–37, Aug. 2016.

[15] F. Bouquet, F. Dadeau, and B. Legeard. Automated boundary test generation from JML specifications. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, Lecture Notes in Computer Science 4085, pages 428–443. Springer Berlin Heidelberg, 2006.

[16] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 123–133, New York, NY, USA, 2002. ACM.

[17] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software – quantify the time and cost saved using reversible debuggers, 2012.

[18] R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. A Declarative Debugger for Sequential Erlang Programs. In M. Veanes and L. Viganò, editors,

Proc. of the 7th International Conference on Tests and Proofs (TAP 2013), Lecture Notes in Computer Science, pages 96–114. Springer, 2013.

[19] R. Caballero, E. Martín-Martín, A. Riesco, and S. Tamarit. A declarative debugger for concurrent erlang programs (extended version). Technical Report SIC-15/13, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2013. URL: `http://maude.sip.ucm.es/~adrian/files/conc_cal_tr.pdf`.

[20] R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. EDD: A declarative debugger for sequential erlang programs. In *TACAS*, volume 8413 of *LNCS*, pages 581–586. Springer, 2014.

[21] L. Cardelli and C. Laneve. Reversible structures. In F. Fages, editor, *Proceedings of the 9th International Conference on Computational Methods in Systems Biology (CMSB 2011)*, pages 131–140. ACM, 2011.

[22] M. Carlier, C. Dubois, and A. Gotlieb. A first step in the design of a formally verified constraint-based testing tool: FocalTest. In A. Brucker and J. Julliand, editors, *Tests and Proofs - 6th International Conference, TAP 2012, Prague, Czech Republic, May 31–June 1, 2012. Proceedings*, Lecture Notes in Computer Science 7305, pages 35–50. Springer, 2012.

[23] M. Carlier, C. Dubois, and A. Gotlieb. FocalTest: A Constraint Programming Approach for Property-Based Testing. In J. Cordeiro, M. Virvou, and B. Shishkov, editors, *Software and Data Technologies - 5th International Conference, ICSOFT 2010, Athens, Greece, July 22–24, 2010. Revised Selected Papers*, Communications in Computer and Information Science 170, pages 140–155. Springer, 2013.

[24] R. Carlsson. An Introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, 2001. URL: `http://www.erlang.se/workshop/carlsson.ps`.

[25] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core erlang 1.0.3. language specification, 2004. URL: `https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf`.

[26] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen. Deterministic replay: A survey. *ACM Comput. Surv.*, 48(2):17:1–17:47, 2015.

[27] J. Cheney and A. Momigliano. αCheck: A mechanized metatheory model checker. *Theory and Practice of Logic Programming*, 17(3):311–352, 2017.

[28] K. Claessen et al. Finding race conditions in Erlang with QuickCheck and PULSE. In *ICFP*, pages 149–160. ACM, 2009.

[29] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In M. Odersky and P. Wadler, editors, *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18–21, 2000*, pages 268–279. ACM, 2000.

[30] D. Coppit, W. Le, K. J. Sullivan, S. Khurshid, and J. Yang. Software assurance by bounded exhaustive testing. *IEEE Transactions on Software Engineering*, 31(4):328–339, 2005.

[31] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms (3rd Ed.)*. MIT Press, 2009.

[32] P. Crescenzi and C. H. Papadimitriou. Reversible simulation of space-bounded computations. *Theoretical Computer Science*, 143(1):159–165, 1995.

[33] I. Cristescu, J. Krivine, and D. Varacca. A compositional semantics for the reversible p-calculus. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013*, pages 388–397. IEEE Computer Society, 2013.

[34] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In R. F. Paige, editor, *Proc. of the 2nd Int'l Conf. on Theory and Practice of Model Transformations (ICMT 2009)*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer, 2009.

[35] V. Danos and J. Krivine. Reversible communicating systems. In P. Gardner and N. Yoshida, editors, *Proc. of the 15th International Conference on Concurrency Theory (CONCUR 2004)*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004.

[36] V. Danos and J. Krivine. Transactions in RCCS. In M. Abadi and L. de Alfaro, editors, *Proc. of the 16th International Conference on Concurrency Theory (CONCUR 2005)*, volume 3653 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[37] A. Di Pierro, C. Hankin, and H. Wiklicky. Reversible combinatory logic. *Mathematical Structures in Computer Science*, 16(4):621–637, 2006.

[38] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods, Proceedings of the 1st International Symposium of Formal Methods Europe, Odense, Denmark, April 19–23, 1993*, Lecture Notes in Computer Science 670, pages 268–284. Springer, 1993.

[39] E. D'Osualdo, J. Kochems, and C. L. Ong. Automatic Verification of Erlang-Style Concurrency. In *SAS*, volume 7935 of *LNCS*, pages 454–476. Springer, 2013.

[40] Erlang Solutions. 20 years of open source erlang: Openerlang interview with anton lavrik from whatsapp, 2018. *Link to the Erlang Solutions blog post*.

[41] Frequently Asked Questions about Erlang, 2018. URL: `http://erlang.org/faq/academic.html`.

[42] G. Feuillade and T. Genet. Reachability in Conditional Term Rewriting Systems. *Electronic Notes in Theoretical Computer Science*, 86(1):133–146, 2003.

[43] J. Field and C. A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pages 195–208. ACM, 2005.

[44] F. Fioravanti, M. Proietti, and V. Senni. Efficient generation of test data structures using constraint logic programming and program transformation. *Journal of Logic and Computation*, 25(6):1263–1283, 2015.

[45] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.

[46] M. P. Frank. Introduction to reversible computing: motivation, progress, and challenges. In N. Bagherzadeh, M. Valero, and A. Ramírez, editors, *Proceedings of the Second Conference on Computing Frontiers*, pages 385–390. ACM, 2005.

[47] L.-A. Fredlund. *A framework for reasoning about Erlang code.* PhD thesis, The Royal Institute of Technology, Sweeden, 2001.

[48] L.-A. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In R. Hinze and N. Ramsey, editors, *Proc. of ICFP 2007*, pages 125–136. ACM, 2007.

[49] T. Genet. Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In T. Nipkow, editor, *Proc. of the 9th International Conference on Rewriting Techniques and Applications (RTA'98)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1998.

[50] T. Genet and Y. Salmon. Reachability Analysis of Innermost Rewriting. In M. Fernández, editor, *Proc. of the 26th International Conference on Rewriting Techniques and Applications (RTA'15)*, volume 36 of *LIPIcs*, pages 177–193. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[51] E. Giachino, I. Lanese, and C. A. Mezzina. Causal-consistent reversible debugging. In S. Gnesi and A. Rensink, editors, *Proc. of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE 2014)*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2014.

[52] E. Giachino, I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-consistent reversibility in a tuple-based language. In M. Daneshtalab, M. Aldinucci, V. Leppänen, J. Lilius, and M. Brorsson, editors, *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015*, pages 467–475. IEEE Computer Society, 2015.

[53] E. Giachino, I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-consistent rollback in a tuple-based language. *J. Log. Algebr. Meth. Program.*, 88:99–120, 2017.

[54] A. Giantsios, N. Papaspyrou, and K. Sagonas. Concolic testing for functional languages. *Sci. Comput. Program.*, 147:109–134, 2017.

[55] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In J. Kramer, J. Bishop, P. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, May 2–8 2010, Cape Town, South Africa*, pages 225–234. ACM, 2010.

[56] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12–15, 2005*, pages 213–223. ACM, 2005.

[57] P. Godefroid, M. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[58] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test case generation for object-oriented imperative languages in CLP. *Theory and Practice of Logic Programming*, 10(4–6):659–674, 2010.

[59] A. Gotlieb. Euclide: A Constraint-Based Testing Framework for Critical C Programs. In *2nd International Conference on Software Testing, Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1–4, 2009*, pages 151–160. IEEE Computer Society, 2009.

[60] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In L. J. et al., editor, *Computational Logic - CL 2000*, Lecture Notes in Computer Science 1861, pages 399–413. Springer, Berlin, Heidelberg, 2000.

[61] A. Gotovos, M. Christakis, and K. Sagonas. Test-driven development of concurrent programs using Concuerror. In K. Rikitake and E. Stenman, editors, *Proc. of the 10th ACM SIGPLAN workshop on Erlang*, pages 51–61. ACM, 2011.

[62] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI '73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[63] N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proc. of IJCAR 2008*, volume 5195 of *Lecture Notes in Computer Science*, pages 364–379. Springer, 2008.

[64] C. M. Holloway. Why engineers should consider formal methods. In *16th DASC. AIAA/IEEE Digital Avionics Systems Conference. Reflections to the Future. Proceedings*, volume 1, pages 1–3. IEEE, 1997.

[65] J. Huang, P. Liu, and C. Zhang. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 385–386. ACM, 2010.

[66] J. Huang and C. Zhang. Debugging concurrent software: Advances and challenges. *J. Comput. Sci. Technol.*, 31(5):861–868, 2016.

[67] L. Huelsbergen. A logically reversible evaluator for the call-by-name lambda calculus. In T. Toffoli and M. Biafore, editors, *Proc. of PhysComp96*, pages 159–167. New England Complex Systems Institute, 1996.

[68] COST Action IC1405 on Reversible Computation - extending horizons of computing. URL: `http://revcomp.eu/`.

[69] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[70] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu. CARE: cache guided deterministic replay for concurrent java programs. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 457–467. ACM, 2014.

[71] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[72] junit-quickcheck: Property-based testing, JUnit-style. URL: `https://github.com/pholser/junit-quickcheck`.

[73] G. Kahn. Natural Semantics. In F. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proc. of STACS'87*, pages 22–39, 1987.

[74] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[75] W. E. Kluge. A reversible SE(M)CD machine. In P. W. M. Koopman and C. Clack, editors, *Proc. of the 11th International Workshop on the Implementation of Functional Languages, IFL'99. Selected Papers*, volume 1868 of *Lecture Notes in Computer Science*, pages 95–113. Springer, 2000.

[76] R. A. Kowalski. *Logic for Problem Solving.* North Holland, 1979.

[77] P. Kuang, J. Field, and C. A. Varela. Fault tolerant distributed computing using asynchronous local checkpointing. In E. G. Boix, P. Haller, A. Ricci, and C. Varela, editors, *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control (AGERE! 2014)*, pages 81–93. ACM, 2014.

[78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[79] L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages*, 2:45:1–45:30, 2017.

[80] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.

[81] I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling reversibility in higher-order pi. In J. Katoen and B. König, editors, *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR 2011)*, volume 6901 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011.

[82] I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversibility in the higher-order π-calculus. *Theor. Comput. Sci.*, 625:25–84, 2016.

[83] I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-consistent reversibility. *Bulletin of the EATCS*, 114, 2014.

[84] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. CauDEr website. URL: `https://github.com/mistupv/cauder`.

[85] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. CauDEr: A causal-consistent reversible debugger for Erlang. In J. P. Gallagher and M. Sulzmann, editors, *Proceedings of the 14th International Symposium on Functional and Logic Programming (FLOPS 2018)*, volume 10818 of *Lecture Notes in Computer Science*, pages 247–263. Springer-Verlag, Berlin, 2018.

[86] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. A theory of reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming*, 100:71–97, 2018.

[87] I. Lanese, A. Palacios, and G. Vidal. CauDEr, Causal-consistent Reversible Replay Debugger. Logger: `https://github.com/mistupv/tracer`, debugger: `https://github.com/mistupv/cauder/tree/replay`.

[88] I. Lanese, A. Palacios, and G. Vidal. Causal-consistent replay debugging for message passing programs. Technical report, DSIC, Universitat Politècnica de València, 2019. URL: `http://personales.upv.es/gvidal/german/forte19tr/paper.pdf`.

[89] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Computers*, 36(4):471–482, 1987.

[90] M. Lienhardt, I. Lanese, C. A. Mezzina, and J.-B. Stefani. A reversible abstract machine and its space overhead. In H. Giese and G. Rosu, editors, *Proceedings of the Joint 14th IFIP WG Int'l Conf. on Formal Techniques for Distributed Systems (FMOODS 2012) and the 32nd IFIP WG 6.1 International Conference (FORTE 2012)*, volume 7273 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2012.

[91] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *PPDP*, pages 167–178. ACM Press, 2006.

[92] C. T. Lopez, S. Marr, H. Mössenböck, and E. G. Boix. A study of concurrency bugs and advanced development support for actor-based programs. *CoRR*, abs/1706.07372, 2017.

[93] C. Lutz and H. Derby. Janus: A time-reversible language, 1986. A letter to R. Landauer. URL: `http://tetsuo.jp/ref/janus.pdf`.

[94] B. Marre. Toward automatic test data set selection using algebraic specifications and logic programming. In K. Furukawa, editor, *Logic Programming, Proceedings of the 8th International Conference, Paris, France, June 24–28, 1991*, pages 202–219. MIT Press, 1991.

[95] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions: GATeL. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ASE 2000, Grenoble, France, September 11–15, 2000*, page 229. IEEE Computer Society, 2000.

[96] M. Maruyama, T. Tsumura, and H. Nakashima. Parallel program debugging based on data-replay. In S.-Q. Zheng, editor, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005)*, pages 151–156. IASTED/ACTA Press, 2005.

[97] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In R. Hinze and N. Ramsey, editors, *Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, pages 47–58. ACM, 2007.

[98] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalizing programs with duplication through complementary function derivation. *Computer Software*, 26(2):56–75, 2009. In Japanese.

[99] A. W. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1987.

[100] F. Mesnard, É. Payet, and G. Vidal. Concolic testing in logic programming. *Theory and Practice of Logic Programming*, 15(4-5):711–725, 2015.

[101] A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

[102] K. Morita. Reversible simulation of one-dimensional irreversible cellular automata. *Theoretical Computer Science*, 148(1):157–163, 1995.

[103] K. Morita. Computation in reversible cellular automata. *International Journal of General Systems*, 41(6):569–581, 2012.

[104] S. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In D. Kozen and C. Shankland, editors, *Proc. of the 7th International Conference on Mathematics of Program Construction (MPC 2004)*, volume 3125 of *Lecture Notes in Computer Science*, pages 289–313. Springer, 2004.

[105] M. Nagashima, M. Sakai, and T. Sakabe. Determinization of conditional term rewriting systems. *Theoretical Computer Science*, 464:72–89, 2012.

[106] R. H. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *The Journal of Supercomputing*, 8(4):371–388, 1995.

[107] R. Neykova and N. Yoshida. Let it recover: multiparty protocol-induced recovery. In P. Wu and S. Hack, editors, *Proceedings of the 26th International Conference on Compiler Construction, CC 2017*, pages 98–108. ACM, 2017.

[108] N. Nishida, A. Palacios, and G. Vidal. Reversible term rewriting. In D. Kesner and B. Pientka, editors, *Proc. of the 1st International Conference on Formal Structures for Computation and Deduction (FSCD'16)*, volume 52 of *LIPIcs*, pages 28:1–28:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[109] N. Nishida, A. Palacios, and G. Vidal. A reversible semantics for Erlang. In M. Hermenegildo and P. López-García, editors, *Proc. of the 26th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2016*, volume 10184 of *LNCS*, pages 259–274. Springer, 2017.

[110] N. Nishida, M. Sakai, and T. Sakabe. Soundness of unravelings for conditional term rewriting systems via ultra-properties related to linearity. *Logical Methods in Computer Science*, 8(3-4):1–49, Aug. 2012.

[111] N. Nishida and G. Vidal. Program inversion for tail recursive functions. In M. Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA 2011)*, volume 10 of *LIPIcs*, pages 283–298. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[112] N. Nishida and G. Vidal. Characterizing compatible view updates in syntactic bidirectionalization. In *Reversible Computation - 11th International Conference, RC 2019, Lausanne, Switzerland, June 24-25, 2019, Proceedings*, pages 67–83, 2019.

[113] M. Papadakis and K. Sagonas. A PropEr Integration of Types and Function Specifications with Property-Based Testing. In K. Rikitake and E. Stenman, editors, *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, Tokyo, Japan, September 23, 2011*, pages 39–50. ACM, 2011.

[114] Z. Paraskevopoulou, C. Hritcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational Property-Based Testing. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving - Proceedings of the 6th International Conference, 2015, Nanjing, China, August 24–27, 2015*, Lecture Notes in Computer Science 9236, pages 325–343. Springer, 2015.

[115] C. Pasareanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In C. Pecheur, J. Andrews, and E. D. Nitto, editors, *ASE*, pages 179–180. ACM, 2010.

[116] R. Perera, D. Garg, and J. Cheney. Causally consistent dynamic slicing. In J. Desharnais and R. Jagadeesan, editors, *CONCUR*, volume 59 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[117] I. Phillips and I. Ulidowski. Reversing algebraic process calculi. *J. Log. Algebr. Program.*, 73(1-2):70–96, 2007.

[118] A. Pretschner and H. Lötzbeyer. Model based testing with constraint logic programming: First results and challenges. In *Proceedings of the 2nd ICSE Workshop on Automated Program Analysis, Testing and Verification (WAPATV)*, pages 1–9, 2001.

[119] PropEr: Property-Based Testing for Erlang. URL: `http://proper.softlab.ntua.gr/`.

[120] B. K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, 1973.

[121] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In A. Gill, editor, *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 37–48. ACM, 2008.

[122] J. M. Rushby. Automated test generation and verified software. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments, 1st IFIP TC2/WG2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10–13, 2005, Revised Selected Papers and Discussions*, Lecture Notes in Computer Science 4171, pages 161–172. Springer, 2008.

[123] ScalaCheck: Property-Based Testing for Scala. URL: `http://www.scalacheck.org/`.

[124] M. Schordan, D. R. Jefferson, P. D. B. Jr., T. Oppelstrup, and D. J. Quinlan. Reverse code generation for parallel discrete event simulation. In J. Krivine and J.-B. Stefani, editors, *Proc. of the 7th International Conference on Reversible Computation (RC 2015)*, volume 9138 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2015.

[125] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In M. Wermelinger and H. C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5–9, 2005*, pages 263–272. ACM, 2005.

[126] K. Shibanai and T. Watanabe. Actoverse: A reversible debugger for actors. In *AGERE*, pages 50–57. ACM, 2017.

[127] T. Stanley, T. Close, and M. S. Miller. Causeway: a message-oriented distributed debugger. Technical report, HPL-2009-78, 2009. Available from `http://www.hpl.hp.com/techreports/2009/HPL-2009-78.html`.

[128] B. Stoddart, R. Lynas, and F. Zeyda. A virtual machine for supporting reversible probabilistic guarded command languages. *Electronic Notes in Theoretical Computer Science*, 253(6):33–56, 2010.

[129] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[130] H. Svensson, L.-A. Fredlund, and C. B. Earle. A unified semantics for future Erlang. In *Proc. of the 9th ACM SIGPLAN workshop on Erlang*, pages 23–32. ACM, 2010.

[131] The SWI Prolog Logic Programming System. URL: `http://www.swi-prolog.org/`.

[132] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[133] M. K. Thomsen and H. B. Axelsen. Interpretation and programming of the reversible functional language RFUN. In R. Lämmel, editor, *Proc. of the 27th Symposium on the Implementation and Application of Functional Programming Languages (IFL'15)*, pages 8:1–8:13. ACM, 2015.

[134] F. Tiezzi and N. Yoshida. Reversible session-based pi-calculus. *J. Log. Algebr. Meth. Program.*, 84(5):684–707, 2015.

[135] T. Toffoli. Computation and construction universality of reversible cellular automata. *Journal of Computer and System Sciences*, 15(2):213–231, 1977.

[136] Undo Software. Increasing software development productivity with reversible debugging, 2014. URL: `https://undo.io/media/uploads/files/Undo_ReversibleDebugging_Whitepaper.pdf`.

[137] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. *ACM Trans. Comput. Syst.*, 30(1):3:1–3:24, 2012.

[138] G. Vidal. Towards Symbolic Execution in Erlang (short paper). In *Proc. of the 9th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI'14)*, pages 351–360. Springer LNCS 8974, 2014.

[139] G. Vidal. Concolic Execution and Test Case Generation in Prolog. In M. Proietti and H. Seki, editors, *Proc. of the 24th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'14)*, pages 167–181. Springer LNCS 8981, 2015.

[140] J. Voigtländer. Bidirectionalization for free! (pearl). In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 165–176. ACM, 2009.

[141] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In M. D. Cin, M. Kaâniche, and A. Pataricza, editors, *Proceedings of the 5th European Dependable Computing Conference, EDCC-5, Budapest, Hungary, April 20–22, 2005*, Lecture Notes in Computer Science 3463, pages 281–292. Springer, 2005.

[142] G. Xu and Z. Yang. JMLAutoTest: A novel automated testing framework based on JML and JUnit. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing. FATES 2003*, Lecture Notes in Computer Science 2931, pages 70–85. Springer Berlin Heidelberg, 2004.

[143] T. Yamakami. One-way reversible and quantum finite automata with advice. *Information and Computation*, 239:122–148, 2014.

[144] K. Yatoh, K. Sakamoto, F. Ishikawa, and S. Honiden. ArbitCheck: A Highly Automated Property-Based Testing Tool for Java. In *Proceedings of the 7th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2014, March 31–April 4, 2014, Cleveland, Ohio, USA*, pages 405–412. IEEE Computer Society, 2014.

[145] T. Yokoyama. Reversible computation and reversible programming languages. *Electronic Notes in Theoretical Computer Science*, 253(6):71–81, 2010.

[146] T. Yokoyama, H. Axelsen, and R. Glück. Principles of a reversible programming language. In A. Ramírez, G. Bilardi, and M. Gschwind, editors, *Proc. of the 5th Conference on Computing Frontiers*, pages 43–54. ACM, 2008.

[147] T. Yokoyama, H. Axelsen, and R. Glück. Fundamentals of reversible flowchart languages. *Theoretical Computer Science*, 611:87–115, 2016.

[148] T. Yokoyama, H. B. Axelsen, and R. Glück. Reversible flowchart languages and the structured reversible program theorem. In *Proc. of the 35th International Colloquium on Automata, Languages and Programming (ICALP 2008)*, volume 5126 of *Lecture Notes in Computer Science*, pages 258–270. Springer, 2008.

[149] T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In G. Ramalingam and E. Visser, editors, *Proc. of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2007)*, pages 144–153. ACM, 2007.

[150] P. Zuliani. Logical reversibility. *IBM Journal of Research and Development*, 45(6):807–818, 2001.