VRIJE
UNIVERSITEIT
BRUSSEL

Proefschrift ingediend met het oog op het behalen van een doctoraatsdiploma
Dissertation submitted in fulfilment of the requirement for the degree of Doctor of Philosophy in Sciences

SCALABLE DESIGNS FOR ABSTRACT INTERPRETATION OF CONCURRENT PROGRAMS:

Application to Actors and Shared-Memory Multi-Threading

Quentin Stiévenart

Promotors: Prof. Dr. Coen De Roover
Prof. Dr. Wolfgang De Meuter

Faculty of Science and Bio-Engineering Sciences

Dissertation submitted in fulfilment of the requirement for the degree of Doctor of Philosophy in Sciences

Scalable Designs for Abstract Interpretation of Concurrent Programs:

Application to Actors and Shared-Memory Multi-Threading

Quentin Stiévenart

Promotors:
Prof. Dr. Coen De Roover
Prof. Dr. Wolfgang De Meuter

Jury:
Prof. Dr. Viviane Jonckers, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Simon Keizer, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. Coen De Roover, Vrije Universiteit Brussel, Belgium (promotor)
Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel, Belgium (promotor)
Prof. Dr. Ann Dooms, Vrije Universiteit Brussel, Belgium
Prof. Dr. Em. Theo D'Hondt, Vrije Universiteit Brussel, Belgium
Prof. Dr. David Van Horn, University of Maryland, USA
Prof. Dr. Philipp Haller, KTH Royal Institute of Technology, Sweden

May 15, 2018

Abstract

Concurrent programs are difficult for developers to reason about. They consist of concurrent processes, of which the execution can be interleaved in an exponential number of ways. Contemporary concurrent programs moreover feature dynamic process creation and termination, exacerbating the need for tool support.

Static program analysis can be a powerful enabler of such tool support, but current static analysis designs for concurrent programs are limited with respect to one or more of the following desirable properties: automation, soundness, scalability, precision and support for dynamic process creation. Moreover, existing analyses are designed with a single concurrency model in mind, and uniform design methods applicable to multiple concurrency models are lacking.

We study the applicability of a recent design method for static program analyses—*abstracting abstract machines* (AAM)—to concurrent programming languages. Applying this method results in analyses featuring all desirable properties except scalability. We present MacroConc and ModConc, two AAM-inspired uniform design methods that, when applied to the operational semantics of a concurrent programming language, result in static analyses featuring all the desired properties.

The first design method, MacroConc, introduces Agha's 1997 notion of macro-stepping into AAM analyses for concurrent programs. Refining the default all-interleavings semantics for concurrent processes with macro-stepping reduces the number of interleavings the analysis has to explore. The resulting analyses remain exponential in their worst-case complexity, but mitigate the scalability issues of existing analyses without compromising their precision. The second design method, ModConc, introduces Cousot and Cousot's 2002 notion of modularity into AAM analyses for concurrent programs. Analyses resulting from this design method consider each process of a concurrent program in isolation to infer potential interferences with other and newly created processes, which will have to be reconsidered until a fixed point is reached. Process interleavings are not explicitly modeled by the analysis but still accounted for. This analysis design trades off precision to yield process-modular analyses that scale linearly with the number of processes created in the program under analysis.

To demonstrate generality, we apply each design method to two prominent concurrent programming models: concurrent actors and shared-memory multi-threading. We prove the soundness and termination properties for each of the resulting analyses formally, and evaluate their running times, scalability and precision empirically on a set of 56 concurrent benchmark programs. Analyses resulting from the application of MacroConc achieve high precision, yet exhibit a reduction in running time of up to four orders of magnitude, compared to analyses resulting from a naive application of AAM. Analyses resulting from the application of ModConc exhibit a more consistent reduction, compared to both analyses resulting from the application of AAM and of MacroConc, but this at the cost of lower precision.

The complementary design methods presented in this dissertation enable one to select an analysis design fitting their needs in terms of scalability and precision, enabling future tool support for contemporary concurrent programs.

Samenvatting

Concurrente programma's kunnen voor ontwikkelaars moeilijk zijn om over te redeneren. Zulke programma's bestaan uit meerdere concurrente processen waarvan de instructies in een exponentieel aantal volgordes uitgevoerd kunnen worden. Dat moderne concurrente programma's bovendien processen dynamisch aanmaken en beëindigen, versterkt alleen de nood aan ondersteuning voor ontwikkelaars.

Zulke ondersteuning kan geboden worden onder de vorm van statische programma-analyse, maar de ontwerpen voor bestaande analyses schieten tekort op een of meerdere van de volgende desiderata: automatisatie, correctheid, schaalbaarheid, precisie, en ondersteuning voor dynamische gecreëerde processen.

In dit proefschrift bestuderen we de inzetbaarheid van een recente ontwerpmethode voor statische programma-analyses, abstracting abstract machines (AAM), voor concurrente programmeertalen. De toepassing van deze methode resulteert namelijk in analyses die voldoen aan alle desiderata, behalve schaalbaarheid. Daarom stellen we twee nieuwe, door AAM geïnspireerde, ontwerpmethoden MacroConc en ModConc voor die, wanneer toegepast op de operationele semantiek van een concurrente programmeertaal, resulteren in statische analyses die aan alle desiderata voldoen.

De eerste ontwerpmethode, MacroConc, vertaalt het concept van macro-stepping (Agha, 1997) naar AAM-gebaseerde analyses. De standaard semantiek voor concurrente programma's verfijnen tot een macro-stepping semantiek zorgt ervoor dat deze analyses minder uitvoeringsvolgordes moeten verkennen. Dit verbetert de schaalbaarheid van de resulterende analyses, maar de worst-case complexiteit blijft exponentieel. De tweede ontwerpmethode, ModConc, vertaalt het concept van een modulair ontwerp (Cousot en Cousot, 2002) naar AAM-gebaseerde analyses voor concurrente programma's. De resulterende analyses beschouwen elk proces van een concurrent programma afzonderlijk, maar herbeschouwen elk proces waarvoor potentiële interferenties vanuit een ander proces gevonden zijn tot een fixpunt bereikt wordt. De dusdanig verkregen resultaten zijn representatief voor elke mogelijke uitvoeringsvolgorde, zonder dat deze expliciet verkend zijn. De ModConc ontwerpmethode ruilt dus precisie van resultaten in voor analyses die lineair schalen met het aantal processen dat door het ge-analyseerde programma gecreëerd wordt.

Om de algemeenheid van onze ontwerpmethoden te demonstreren, passen we elke methode toe op twee prominente modellen voor concurrent programmeren: concurrent actors en shared-memory multi-threading. We bewijzen de correctheid en de eindigheid van elke resulterende analyse formeel, en evalueren hun snelheid, schaalbaarheid en precisie empirisch op een verzameling van 56 concurrente programma's. De uit MacroConc resulterende analyses zijn precies, en toch nog tot vier ordegroottes sneller dan analyses die resulteren uit een naïeve toepassing van AAM. De uit ModConc resulterende analyses zijn dan weer consistenter sneller dan analyses resulterende uit zowel AAM als MacroConc, maar dit wel ten koste van precisie.

De complementaire ontwerpmethoden die in dit proefschrift voorgesteld worden, stellen ontwerpers van statische programma-analyses in staat een ontwerp te selecteren overeenkomstig de gewenste precisie en schaalbaarheid —en komt op die manier tegemoet aan de nood aan ondersteuning voor ontwikkelaars van concurrente programma's.

ACKNOWLEDGEMENTS

First of all, I would like to thank the members of my jury (Viviane Jonckers, Simon Keizer, Ann Dooms, Theo D'Hondt, David Van Horn, and Philipp Haller) for their insightful comments and feedback.

I would also like to express all of my gratitude towards my promotors, Coen De Roover and Wolfgang De Meuter. None of this work would have been possible without their support and feedback. My work and this resulting dissertation have greatly profited from their many (many) helpful comments, remarks, and suggestions.

Jens Nicolay deserves a special thanks, as most of my research these past years has been done in cooperation with him, thanks to his sudden bursts in my office with new ideas and insights, most of which still need to be investigated; and thanks to his ability to restructure a first draft into a paper with a structure of great quality.

I'm also thankful to those who proofread part of this dissertation: thanks Maarten, Noah, and Jens; and to my promotors who have read it from front to back in no time: thanks Coen and Wolf.

A big thanks to all of the SOFT members, who are all part of what makes SOFT a particularly interesting work environment in which people can both have fun and at the same time strive for work of a great quality. There have been many great moments, including research presentations, drinks, ice-cream breaks, interesting discussions, vape sessions, memorable conference trips, and many others. This all helps in the life of a PhD student, and I'm glad to be a part of this lab.

I would also like to thank Roxane, my family, and my friends for their support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

In this dissertation, we present and study two analysis design methods that, when applied to the operational semantics of a concurrent programming language, result in static program analyses that are automated, sound, scalable, precise and that support dynamic process creation in the program under analysis. We apply these design methods to two models of concurrent programming: concurrent actors and shared-memory multi-threading. We prove soundness and termination properties of each analysis formally and evaluate their running times, scalability and precision empirically.

1.1. Research Context

Concurrent programs tend to be difficult for developers to reason about. They consist of concurrent processes, of which the execution can be interleaved in an exponential number of ways. Errors present in concurrent programs tend to arise only under specific process interleavings and in a non-deterministic manner. This is complicated further by the fact that concurrent programs may create and terminate new processes dynamically, rendering such programs harder to understand. Tool support is required in order to aid developers reasoning about concurrent programs.

Static program analysis is a powerful enabler of tool support for developers. A static analysis reasons about the possible run-time behaviors of a program without executing it. Applications can be found in program comprehension, bug detection, and automated program verification. By accounting for all possible run-time behaviors, a static analysis can reveal potential defects that could otherwise manifest only late in the development process or even after the application has been deployed. More interestingly, a static analysis can prove the absence of defects by over-approximating all the possible behaviors of a program.

Static analysis of concurrent programs is a challenging undertaking for a number of reasons, but its necessity is growing as concurrent programs gain prominence. We review the major challenges faced when designing a static analysis for concurrent programs.

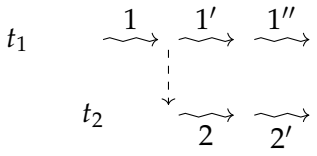
1.1.1. Static Analyses for Concurrent Programs Scale Poorly

Concurrent programs have long been considered problematic for static analysis (Cousot and Cousot, 1980; Cousot and Cousot, 1984). Concurrent programs can be highly non-deterministic in their execution, as multiple processes are executed concurrently and their executions may interleave in an exponential number of possible ways (see Figure 1.1). This is known as the state explosion problem in analysis of concurrent programs (Valmari, 1996). Mitigations for this problem have been studied in the context of model checking, leading to partial-order reduction techniques (Godefroid, 1996; Flanagan and Godefroid, 2005) which reduce the number of interleavings a model checker has to explore, rendering it more scalable and enabling the verification of more complex programs. Such techniques do not solve the explosion problem for concurrency, as their worst-case complexity remains exponential, but mitigate the problem sufficiently so that real-world concurrent programs can be verified within minutes or hours (Yang *et al.*, 2008; Abdulla *et al.*, 2014). While model checking is free of false positives (any detected error is a true error) and can prove correctness without false negatives (no undetected errors) for finite systems, it is limited to analyzing concurrent programs that exhibit finite behavior, while contemporary concurrent programs may exhibit infinite behavior (e.g., a concurrent web server is not supposed to terminate).

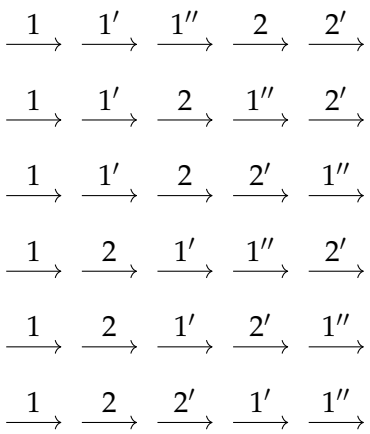
Abstract interpretation (Cousot and Cousot, 1977) is an established approach to static analysis capable of supporting infinite systems. Unfortunately, partial-order reduction techniques are not applicable to abstract interpretation because they do not support

Non-determinism in concurrent programs

Consider a program initially executing a single thread t_1 performing the following operations: instruction 1 spawns another thread t_2, before executing instructions $1'$ and $1''$, while thread t_2 executes instructions 2 and $2'$. This concurrent behavior is depicted in Figure 1.1a. The execution of this program orders the execution of threads t_1 and t_2 in a specific order. If no ordering is enforced through synchronization mechanisms such as locks, these instructions can be interleaved in six different ways, represented in Figure 1.1b. Depending on the interleaving, the result of the program may differ.



(a) Representation of a concurrent computation between two threads. Plain arrows denote execution of an instruction, and dashed arrows denote process creation.



(b) Possible interleavings for the concurrent execution of the two threads of Figure 1.1a. Plain arrows denote execution of an instruction. Each line represents a possible interleaving.

Figure 1.1.: Non-determinism in concurrent programs.

the cyclic state spaces present in abstract interpretation (Flanagan and Godefroid, 2005). A mitigation technique for the state space explosion problem in concurrent programs adapted to static analysis is therefore required. Existing dynamic analysis tools for concurrent actor programs (Sen and Agha, 2006b; Lauterburg *et al.*, 2009) rely on *macro-stepping* (Agha *et al.*, 1997) to reduce the state space explored. Macro-stepping has, however, yet not been adapted to an abstract interpretation setting.

A more adequate solution to the state explosion problem in analysis of concurrent programs consists of modularizing an analysis (Cousot and Cousot, 2002) by analyzing parts of a program (e.g., its processes) in isolation. Interleavings of different processes are not explicitly represented in a process-modular analysis but are still over-approximated in a sound manner, at the cost of precision. Such analyses are not prone to the state explosion problem and have been used to verify complex industrial concurrent programs (Miné and Delmas, 2015), but only a few modular analyses have been designed explicitly for concurrent programs (Flanagan *et al.*, 2005; Kahlon *et al.*, 2005; Miné, 2014; Midtgaard *et al.*, 2016b), none of which supports dynamic process creation.

1.1.2. Static Analyses for Concurrent Programs Lack Support for Dynamic Process Creation

Modern concurrent programs exhibit highly dynamic behavior such as dynamic process creation and termination. For example, a server distributing concurrent requests among a pool of worker processes may create new processes at run time under high load and terminate existing processes when the load decreases (Lea, 1997). Dynamic process creation is a challenging feature for static analysis to support (Huch, 1999). Existing static analyses featuring partial-order reduction (Sen and Agha, 2006b; Lauterburg *et al.*, 2009; Sen and Agha, 2006a; Tasharofi *et al.*, 2012; Godefroid, 1996; Flanagan and Godefroid, 2005; Christakis *et al.*, 2013) or adhering to a process-modular design (Flanagan *et al.*, 2002; Henzinger *et al.*, 2003; Miné, 2014; Monat and Miné, 2017; Gotsman *et al.*, 2007; Midtgaard *et al.*, 2016b) may scale well and have been shown to verify complex concurrent programs (Miné and Delmas, 2015), but do not support dynamic process creation, which limits them to analyzing programs with a constant set of processes that is known a priori. Existing static analyses supporting dynamic process creation are few and far between, an exception being the Soter tool (D'Osualdo *et al.*, 2012), but this tool fails to scale beyond small benchmark programs due to its non-modularity, as demonstrated in Chapter 4.

1.1.3. Static Analyses for Different Concurrency Models Lack Uniformity in their Design

Shared-memory multi-threading is perhaps the predominant concurrency model available in mainstream programming languages. Analysis support for this model exists in the form of model checking (Godefroid and Wolper, 1991; Godefroid, 1996) and process-modular analysis (Flanagan *et al.*, 2005; Miné, 2014). The actor model (Hewitt *et al.*, 1973; Agha, 1986) is a concurrent programming model that has been growing in popularity

over the past years. It has been formulated several times resulting in multiple variants of the model (Hewitt and Smith, 1975; Yonezawa *et al.*, 1986; Agha, 1990; Armstrong *et al.*, 1993; Varela and Agha, 2001; Miller *et al.*, 2005; Van Cutsem *et al.*, 2007; Haller and Odersky, 2009; Srinivasan and Mycroft, 2008; Caromel *et al.*, 2009). We refer to De Koster *et al.* (2016) for a detailed history and a taxonomy of the variants of the actor model. Distributed variants of the actor model are, for instance, increasingly advocated as a foundation for micro-service architectures (Haller and Sommers, 2012; Nash and Waldron, 2016). However, few static analyses support the actor model (e.g., D'Osualdo *et al.* (2013) and Stiévenart *et al.* (2017)). Analyses for other concurrency models are few and far between. Channel concurrency (Hoare, 1978) has seen support from a number of static analyses, but these remain limited to programs with a constant set of processes that is known a priori (Mercouroff, 1991; Ng and Yoshida, 2016; Stadtmüller *et al.*, 2016; Colby, 1995; Martel and Gengler, 2000; Ladkin and Simons, 1992). Software transactional memory (Shavit and Touitou, 1997) has been studied in the context of compiler optimization (Afek *et al.*, 2010), but lacks support from general-purpose static analyses. Atomics have seen static analysis support in the form of abstract interpretation (Might and Van Horn, 2011), but this approach fails to scale due to its explicit modeling of all execution interleavings, as demonstrated in Chapter 2. As each of the mentioned analyses is dedicated to a single concurrency models, improvements designed for such an analysis may not apply to analyses designed for other models. We strive for unified designs that are applicable to multiple concurrent programming paradigms.

1.2. Problem Statement

Concurrent programs tend to be difficult for developers to reason about, as they consist of concurrent processes of which the execution can be interleaved in an exponential number of ways. Contemporary concurrent programs moreover feature dynamic process creation and termination, increasing the need for tool support. Static program analysis can be a powerful enabler of such tool support, but current static analysis designs for concurrent programs following from model checking and abstract interpretation are limited with respect to one or more of the following desirable properties.

Automation
: Automation is important to reduce the burden of using an analysis for a developer, enabling developers to use static analysis tools without requiring expertise in static analysis.

Soundness
: Soundness is necessary to ensure that the results given by an analysis can be trusted: if the analysis reports a program as safe, the user of the analysis can expect the program to be safe in all of its executions.

Scalability
: As concurrent programs may exhibit highly non-deterministic behavior, static

analyses have to scale well in order to support analyzing programs larger than a few lines of code.

Precision
Developers are prone to ignore the results of an imprecise analysis that reports too many false positives (Johnson *et al.*, 2013; Bessey *et al.*, 2010). Precision is therefore another important criterion for a static analysis.

Support for dynamic process creation
Finally, modern concurrent programs exhibit dynamic process creation, where processes may be created an terminated at any point in the program's execution, which has to be supported by static analyses.

We therefore aim to provide a solid foundation for designing static analyses for concurrent programs that combine all these crucial properties.

1.3. Thesis

The thesis defended in this dissertation is the following. Static program analysis is a powerful enabler of tool support for developers, but current static analysis designs for concurrent programs are limited with respect to one or more of the following properties: automation, soundness, scalability, precision and support for dynamic process creation. We propose two design methods that, when applied to the operational semantics of a concurrent programming language, result in static analyses featuring these desirable properties. The first design method mitigates scalability issues of existing analyses without compromising their precision, while the second trades off precision to yield truly scalable analyses.

1.4. Overview of The Approach

Our analysis design methods find their root in the abstracting abstract machines (AAM) analysis design method of Van Horn and Might (2010). In this design method, programming language semantics are encoded in a small-step operational manner, with all values and continuations allocated at addresses in stores so that abstraction can be performed by rendering the addresses finite. Starting from analyses resulting from the naive application of the AAM method, we derive two distinct AAM-inspired uniform designs, MacroConc and ModConc, that, when applied to the operational semantics of a concurrent programming language, result in static analyses featuring automation, soundness, scalability, precision and support for dynamic process creation in the program under analysis.

The first design method, MacroConc, introduces Agha's notion of *macro-stepping* (Agha *et al.*, 1997) into AAM analyses for concurrent programs. Refining the default all-interleavings semantics for concurrent processes with macro-stepping reduces the number of interleavings an analysis has to explore. The resulting analyses remain exponential

in their worst-case complexity, but mitigate the scalability issues of existing analyses without compromising their precision.

The second design method, ModConc, introduces Cousot and Cousot's notion of modular analysis (Cousot and Cousot, 2002) into AAM analyses for concurrent programs. Analyses resulting from this design method consider each process of a concurrent program in isolation to infer potential interferences with other and newly created processes, which will have to be reconsidered until a fixed point is reached. Process interleavings are not explicitly modeled by the analysis but still accounted for. This analysis design method trades off precision to yield process-modular analyses that scale linearly with the number of processes created in the program under analysis.

To evaluate our design methods, we apply each method to the design of an analysis for a concurrent actor language and for a multi-threaded language with shared memory. We provide a formal model of the languages and the analyses developed in this dissertation, and we implement each analysis on top of our Scala-AM static analysis framework (Stiévenart *et al.*, 2016b). The formalization enables proving the theoretical properties of the resulting analyses, while the implementations enable evaluating each analysis in terms of running times, scalability and precision.

1.5. Contributions

The contributions of this dissertation are the following.

MacroConc: A design method for all-interleavings macro-stepping analysis
: Our first contribution is an analysis design method that improves the efficiency of AAM-style analyses for concurrent programs without compromising precision, when existing improvements based on partial-order reduction are not applicable. We take inspiration from the concept of macro-stepping introduced by Agha *et al.* (1997) in the context of actor programs to reduce the number of interleavings that have to be accounted for in a static analysis. We apply this design method to an analysis for concurrent actor programs and to the equivalent analysis for shared-memory concurrent programs. We demonstrate the soundness and termination of the resulting analyses formally and evaluate their running times, scalability and precision empirically.

A study of mailbox abstraction strategies
: We perform an in-depth study of a macro-stepping analysis for actor programs resulting from the application of the MacroConc design method. We propose several mailbox abstractions and study how such abstractions affect the running time and the precision of the analysis. We categorize the mailbox abstractions according to whether they preserve the ordering of messages and according to whether they preserve the multiplicity of messages. We formally prove these abstractions sound and empirically evaluate their impact on the running time and precision of the analysis.

ModConc: A design method for process-modular analyses

As an alternative to all-interleavings analyses that are subject to the state explosion problem, albeit mitigated through macro-stepping, we propose a design method for process-modular analyses that scales for concurrent programs with dynamic process creation. ModConc introduces Cousot and Cousot's notion of modularity (Cousot and Cousot, 2002) into AAM-style analyses for concurrent programs. Analyses resulting from this design method consider each process of a concurrent program in isolation to infer potential interferences with other and newly created processes, which will have to be reconsidered for analysis until a fixed point is reached. Process interleavings are not explicitly modeled by the analysis but still accounted for. This analysis design trades off precision to yield process-modular analyses that scale linearly with the number of processes created in the program under analysis. We describe and apply this design method to concurrent actor programs and to shared-memory multi-threading. We demonstrate the soundness and termination of the resulting analyses formally and evaluate their running times, scalability and precision empirically.

Scala-AM: A static analysis framework

In the context of this dissertation, we have developed Scala-AM, a framework for implementing static program analyses. The framework is designed to be modular and extensible, so that static analysis developers can easily explore and integrate new static analysis ideas. The framework's flexibility is demonstrated by the use of Scala-AM to evaluate all the analyses designed in this dissertation, as well as its use in a number of other works (Vandercammen *et al.*, 2015; Vandercammen and De Roover, 2016; De Bleser *et al.*, 2017; Vandercammen and De Roover, 2017; Van Es *et al.*, 2017b).

1.6. Supporting Publications

This dissertation is supported by the following conference papers.

- Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover, "Detecting concurrency bugs in higher-order programs through abstract interpretation", in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, M. Falaschi and E. Albert, Eds., ACM, 2015, pp. 232–243, ISBN: 978-1-4503-3516-4. DOI: 10.1145/2790449.2790530. This conference paper describes and evaluates the performance of a naive application of the AAM design method for an analysis of multi-threaded concurrency in the same line as the analyses presented in Chapter 2.

- Q. Stiévenart, M. Vandercammen, W. De Meuter, and C. De Roover, "Scala-am: A modular static analysis framework", in *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, IEEE Computer Society, 2016, pp. 85–90, ISBN: 978-1-5090-3848-0. DOI:

10.1109/SCAM.2016.14. This short conference paper describes the architecture of our static analysis framework with which we implement the analyses developed in Chapters 2 and 4 to 6.

- Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover, "Building a modular static analysis framework in scala (tool paper)", in *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, A. Biboudis, M. Jonnalagedda, S. Stucki, and V. Ureche, Eds., ACM, 2016, pp. 105–109, ISBN: 978-1-4503-4648-1. DOI: 10.1145/2998392.3001579. This symposium tool paper describes the details of the implementation of our static analysis framework.

- Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover, "Mailbox abstractions for static analysis of actor programs", in *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, P. Müller, Ed., ser. LIPIcs, vol. 74, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 25:1–25:30, ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.25. This conference paper describes a macro-stepping analysis for concurrent actor programs, in the same line as the macro-stepping analysis for concurrent actors presented in Chapter 4, and provides the categorization of the mailbox abstractions presented in Chapter 5.

The following journal articles are currently under review.

- Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover, "Smap: Scalable modular static analysis of actor programs", *Submitted on Oct. 17, 2017 to Information & Software Technology*, This paper describes a process-modular analysis for concurrent actor programs, in the same line as the process-modular analysis for concurrent actor programs presented in Chapter 6.

- Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover, "A general method for rendering analyses for diverse concurrency models modular", *Submitted on Feb. 22, 2018 to Journal of Systems and Software*, This paper introduces the ModConc uniform analysis design method, presented in Chapter 6.

A number of additional publications document our academic track record, touching upon topics addressed in this dissertation including static analysis, interpreters and abstract machines in the context of higher-order languages.

- Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover, "Poster: Static analysis of concurrent higher-order programs", in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds., IEEE Computer Society, 2015, pp. 821–822. DOI: 10.1109/ICSE.2015.265.

- M. Vandercammen, Q. Stiévenart, W. De Meuter, and C. De Roover, "STRAF: A scala framework for experiments in trace-based JIT compilation", in *Grand Timely*

Topics in Software Engineering - International Summer School GTTSE 2015, Braga, Portugal, August 23-29, 2015, Tutorial Lectures, J. Cunha, J. P. Fernandes, R. Lämmel, J. Saraiva, and V. Zaytsev, Eds., ser. Lecture Notes in Computer Science, vol. 10223, Springer, 2015, pp. 223–234. DOI: 10.1007/978-3-319-60074-1_10.

- N. Van Es, J. Nicolay, Q. Stiévenart, T. D'Hondt, and C. De Roover, "A performant scheme interpreter in asm.js", in *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, S. Ossowski, Ed., ACM, 2016, pp. 1944–1951, ISBN: 978-1-4503-3739-7. DOI: 10.1145/2851613.2851748.
- J. De Bleser, Q. Stiévenart, J. Nicolay, and C. De Roover, "Static taint analysis of event-driven scheme programs", in *10th European Lisp Symposium, ELS 2017, April 3-4, 2017, Brussels, Belgium*, 2017, pp. 80–87.
- N. Van Es, Q. Stiévenart, J. Nicolay, T. D'Hondt, and C. De Roover, "Implementing a performant scheme interpreter for the web in asm.js", *Computer Languages, Systems & Structures*, vol. 49, pp. 62–81, 2017. DOI: 10.1016/j.cl.2017.02.002.
- J. Nicolay, Q. Stiévenart, W. De Meuter, and C. De Roover, "Purity analysis for javascript through abstract interpretation", *Journal of Software: Evolution and Process*, e1889–n/a, 2017, e1889 smr.1889, ISSN: 2047-7481. DOI: 10.1002/smr.1889.

1.7. Dissertation Outline

This dissertation is structured as follows.

Chapter 2. Introduction to Abstract Interpretation of Concurrent Programs
We start our exposition with an introduction to concurrent programming and to the abstract interpretation approach to static analysis used throughout the dissertation. A base sequential language λ_0 is introduced, as well as two separate extensions with support for concurrency. The first extension adds the concept of concurrent actors to λ_0, giving rise to the λ_α language. The second extension adds multi-threading and shared memory to λ_0, giving rise to the λ_τ language. Both extensions feature dynamic process creation, an essential feature of contemporary concurrent programs. For each of these extensions, we provide the formal syntax and semantics, as well as a naive application of the AAM analysis design method (Van Horn and Might, 2010). We demonstrate that this naive application of AAM to these concurrent programming languages results in static analyses that scale poorly. As such, this design method forms a baseline that is improved upon in the subsequent chapters.

Chapter 3. State of the Art in Static Analysis of Concurrent Programs
This chapter discusses the state of the art in analysis of concurrent programs. It presents the related work that comprises many different approaches to program analysis: bug finding, abstract interpretation, model checking, types systems and

proof systems. We highlight that existing analysis designs are limited with respect to one of the following properties: automation, soundness, scalability and precision, or support for dynamic thread creation. The naive application of the AAM method developed in the previous chapter only lacks scalability, and we therefore review two approaches to address scalability issues for static analysis of concurrency models: state space reduction and process-modular analysis.

Chapter 4. MacroConc: Designing Macro-Stepping Analyses

This chapter presents MacroConc, our first analysis design method that results in an analysis featuring the desired properties (albeit scalable only to a limited extent). We introduce the notion of macro-stepping, adapted from Agha *et al.* (1997), in order to speed up the analysis by ignoring certain interleavings that are shown not to be for the verification of local process properties. Macro-stepping was initially described in an actor setting (Agha *et al.*, 1997), which we generalize to other concurrency models. Semantics featuring macro-stepping perform more than a single step on a process before investigating interleavings with other processes. We apply the MacroConc design method for analyses of the λ_α actor language and the λ_τ multi-threaded language. Instantiations of macro-stepping define a function that states which communication effects are not allowed in the same macro step, after a certain effect has been produced. This results in analyses that perform significantly better than the analyses resulting from a naive application of AAM, as demonstrated in the evaluation section of this chapter. However, scalability remains limited as there are programs in our benchmark suite for which the analyses do not terminate within their time budget.

Chapter 5. A Study of Mailbox Abstractions

In this chapter, we take a detour for an in-depth study of our macro-stepping analysis for concurrent actor programs with ordered mailboxes. We describe two important properties for actor programs: absence of errors and mailbox bounds. We demonstrate that preserving ordering and multiplicity on the abstraction of mailboxes is important in order to verify such properties. We categorize different mailbox abstractions according to whether they preserve ordering information and according to whether they preserve multiplicity information. We formally prove the mailbox abstractions sound and empirically evaluate the impact of each mailbox abstraction for verifying absence of errors and for inferring mailbox bounds.

Chapter 6. ModConc: Designing Modular Analyses

This chapter presents ModConc, our second analysis design method which takes inspiration from the concept of modular analysis of Cousot and Cousot (2002), and adds support for dynamic process creation. We present it as a uniform method that we apply for static analysis of concurrent actor programs and of multi-threaded programs. Each process is analyzed in isolation through an intra-process analysis, relying on communication effects to infer created processes and other communications between processes. The result of the intra-process analysis is the set of

created processes and of process communications. This inferred information is then reflected on the global analysis state by an inter-process analysis. Using a fixed-point formulation, the inter-process analysis terminates after having analyzed every created process with every communication between processes. We formally prove that the resulting analyses are sound, terminate, and are scalable. We empirically demonstrate the improved scalability compared to macro-stepping analysis, and evaluate the impact on the analysis' precision.

Chapter 7. Conclusion and Future Work

We conclude this dissertation by recapitulating our main contributions. We also identify possible directions for future work on the presented analysis design methods. First, while we applied our design methods to static analysis of concurrent actors and multi-threading, they remain to be investigated in the context of other concurrency models, and combinations thereof. Second, we identify possible extensions to our concurrency models to more closely match real-world implementations. Finally, we discuss possibilities to increase the precision of our analyses on three axes: by considering context-sensitivities and process-sensitivities, by incorporating order information into the ModConc design method, and by combining modularity and macro-stepping into one analysis design method. We conclude with potential applications for the analyses described in this dissertation, which include program comprehension tools as well as automated verification methods for diverse concurrency models.

2

INTRODUCTION TO ABSTRACT INTERPRETATION OF CONCURRENT PROGRAMS

In this chapter, we introduce a base sequential language λ_0, and two separate extensions of this language that each implement a different concurrency model, namely the actor model in λ_α and the model of shared-memory multi-threading in λ_τ. Each language is presented with its concrete semantics, formulated as the fixed-point of a transfer function relying on a transition relation. The transition relation itself is annotated with *communication effects*. Communication effects describe information about the concurrent behavior of a program's execution. This formulation enables performing static analysis by abstract interpretation for each language, following the AAM approach of Van Horn and Might (2010).

We call the application of AAM to the semantics of these languages presented in this chapter *naive*, in the sense that the resulting static analyses perform no optimizations for scalability, and model all possible process interleavings explicitly. These analyses are subject to the state explosion problem due to the high number of possible interleavings, and these analyses therefore suffer from scalability issues. We formally prove that the resulting analyses are sound and always terminate. We also evaluate these naive abstract interpretations empirically in terms of soundness and running time, on a set of 56 benchmark programs, confirming the scalability issues. While sound, the resulting analyses fail to analyze most benchmarks within their time budget due to scalability issues.

As we work on three different languages (λ_0, λ_α, and λ_τ), we introduce the following convention to avoid confusion: formalization of language components are given in boxes labeled with the corresponding language name.

2.1. A Functional Sequential Subset: λ_0

The base language used as the sequential foundation for both our concurrent languages is called λ_0 and is based on the λ-calculus in A-Normal Form (Flanagan *et al.*, 1993). A-Normal Form, or ANF, is a restricted syntactic form for λ-calculus programs, where operators and operands are restricted to *atomic expressions*. Atomic expressions *ae* are expressions that can be evaluated immediately without impacting the program state, as opposed to expressions *e* that are not known to terminate and may impact the program state. This syntactic simplification of the language does not limit its expressiveness, as any λ-calculus program can be automatically rewritten into its A-Normal Form (Flanagan *et al.*, 1993). When presenting example programs, we do not restrict their syntax to ANF, but rather rely on the Scheme subset supported by our implementation, which features more data types (e.g., integers, cons cells, vectors), operations (e.g., `+`, `cons`, `make-vector`) and language constructs (e.g., `if`, `do`).

2.1.1. Syntax of λ_0

The λ_0 language, of which the syntax is given in Figure 2.1, includes recursive bindings (`letrec`), closure creation (λ) and application ($(f\ ae)$), run-time errors (`error`), and has closures as sole values. Variable references and closure creations are atomic expressions. Variables are named syntactically and there is a finite number of variable names in any given program.

λ_0

$$
\begin{aligned}
e \in Exp &::= ae \\
&\mid (f\ ae) \\
&\mid (\texttt{letrec}\ ((x\ e_1))\ e_2) \\
&\mid (\texttt{error})
\end{aligned}
\qquad
\begin{aligned}
f, ae \in AExp &::= x \mid lam \\
lam \in Lam &::= (\lambda\ (x)\ e) \\
x, y \in Var &\text{ a finite set of names}
\end{aligned}
$$

Figure 2.1.: Syntax of λ_0.

2.1.2. Concrete Semantics of λ_0

We define the concrete semantics of λ_0 in a small-step operational manner, as a transition relation ($\hookrightarrow$) acting on states ς, memory heaps (called *value stores*) σ, and continuation stacks (called *continuation stores*) Ξ. We write $\varsigma, \sigma, \Xi \hookrightarrow \varsigma', \sigma', \Xi'$ to denote a transition from state ς with value store σ and continuation store Ξ to state ς', value store σ' and continuation store Ξ'.

State Space

Figure 2.2 depicts the state space for λ_0. A state ς is composed of a control component c and a continuation address k. The control component c dictates whether a state is an

evaluation state (**ev**), in which an expression is evaluated in a given environment, a value state (**val**), in which a value has been reached, or an error state (**err**), in which the execution has reached a user error. The continuation address k of a state points to the continuation of the computation in the continuation store. Environments ρ map variables to addresses, value stores σ map addresses to values and continuation stores Ξ map continuation addresses to continuations. A continuation κ is either a frame ϕ and a continuation address for the next continuation on the stack, or it is empty, representing the absence of continuation. We use this form of continuation allocated through a continuation store to enable abstraction later on. The λ_0 language only has a single type of frame: $\mathbf{letrec}(e, a, \rho)$, used to evaluate the body of a `letrec` expression, where e is the body expression, a is the address in the store of the newly bound variable being evaluated, and ρ is the definition environment that records lexical scope information. The language only supports closures (**clo**) as values. A closure pairs a lambda expression *lam* with a binding environment ρ. We leave the set of addresses *Addr* and *KAddr* as parameters, and give a possible instantiation when discussing allocation.

λ_0

$$
\begin{array}{rlrl}
\varsigma \in \Sigma = \mathit{Control} \times \mathit{KAddr} & & \sigma \in \mathit{Store} = \mathit{Addr} \to \mathit{Val} & \\
c \in \mathit{Control} ::= & \mathbf{ev}(e, \rho) & \Xi \in \mathit{KStore} = \mathit{KAddr} \to \mathit{Kont} & \\
\mid & \mathbf{val}(v) & \kappa \in \mathit{Kont} ::= & \phi : k \mid \epsilon \\
\mid & \mathbf{err} & \phi \in \mathit{Frame} ::= & \mathbf{letrec}(e, a, \rho) \\
v \in \mathit{Val} ::= & \mathbf{clo}(\mathit{lam}, \rho) & a \in \mathit{Addr} & \text{a set of addresses for values} \\
\rho \in \mathit{Env} = \mathit{Var} \to \mathit{Addr} & & k \in \mathit{KAddr} & \text{a set of addresses for continuations}
\end{array}
$$

Figure 2.2.: State space for λ_0.

Atomic Evaluation

Atomic expressions are expressions that can always be evaluated in a single small step without impacting the stores of the program state. Figure 2.3 depicts the atomic evaluation relation for λ_0 programs. We denote atomic evaluation as $\rho, \sigma \vdash \mathit{ae} \Downarrow v$, meaning that atomic expression *ae* evaluates to value v in environment ρ and store σ. A variable reference evaluates to the value found in the store at the address for this variable found in the environment (rule Var). A lambda expression evaluates to a closure that pairs the expression with the current environment (rule Lambda).

λ_0

$$\frac{v = \sigma(\rho(x))}{\rho, \sigma \vdash x \Downarrow v}\ \textsc{Var} \qquad \frac{}{\rho, \sigma \vdash lam \Downarrow \mathbf{clo}(lam, \rho)}\ \textsc{Lambda}$$

Figure 2.3.: Atomic evaluation for λ_0.

Transition Relation

Figure 2.4 depicts the transition relation $\hookrightarrow$ of λ_0, which encodes the common call-by-value λ-calculus semantics. Rule Atomic evaluates an atomic expression using atomic evaluation. Rule App evaluates a function application by atomically evaluating the function and its argument, and stepping into the function body with an extended environment and value store. The address with which the store is extended is generated by the *alloc* allocation function, discussed later in this section. This allocation function returns a fresh address, i.e., an address that has not been used before. Rules Letrec1 and Letrec2 encode the semantics of `letrec`: first a new address is allocated for the variable to be bound using the *alloc* allocation function, and the evaluation steps into the expression computing the value for this variable, pushing a frame for the evaluation of the body on top of the continuation stack, encoded in the continuation store. This push is performed by allocating a new continuation address using the allocation function *kalloc*, described later in this section. This allocation function returns a fresh continuation address. When the value for the variable has been evaluated, the store is extended to incorporate it, and evaluation continues with the body of the `letrec`. Finally, rule Error steps the program into an error state when the `error` primitive is called.

λ_0

$$\frac{\rho, \sigma \vdash ae \Downarrow v}{\langle \mathbf{ev}(ae, \rho), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{val}(v), k\rangle, \sigma, \Xi} \textsc{Atomic}$$

$$\frac{\rho, \sigma \vdash f \Downarrow \mathbf{clo}((\lambda\ \ (x)\ \ e), \rho') \qquad \rho, \sigma \vdash ae \Downarrow v \qquad a = alloc(x, \sigma)}{\langle \mathbf{ev}((f\ ae), \rho), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{ev}(e, \rho'[x \mapsto a]), k\rangle, \sigma[a \mapsto v], \Xi} \textsc{App}$$

$$\frac{a = alloc(x, \sigma) \qquad k' = kalloc(e_2, \rho, \sigma, \Xi) \qquad \rho' = \rho[x \mapsto a]}{\begin{array}{c}\langle \mathbf{ev}((\texttt{letrec}\ ((x\ e_1))\ \ e_2), \rho), k\rangle, \sigma, \Xi \hookrightarrow \\ \langle \mathbf{ev}(e_1, \rho'), k'\rangle, \sigma, \Xi[k' \mapsto \mathbf{letrec}(e_2, a, \rho') : k]\end{array}} \textsc{Letrec1}$$

$$\frac{\Xi(k) = \mathbf{letrec}(e, a, \rho) : k'}{\langle \mathbf{val}(v), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{ev}(e, \rho), k'\rangle, \sigma[a \mapsto v], \Xi} \textsc{Letrec2}$$

$$\frac{}{\langle \mathbf{ev}((\texttt{error}), \rho), k\rangle, \sigma, \Xi \hookrightarrow \langle \mathbf{err}, k\rangle, \sigma, \Xi} \textsc{Error}$$

Figure 2.4.: Transition relation of λ_0.

Allocation

Concrete allocation functions for λ_0 are defined in Figure 2.5. The allocation of value addresses is defined by the *alloc* allocation function, which takes as arguments an atomic expression, and the current value store σ to generate an address. The allocation of continuation addresses is defined by the *kalloc* allocation function, which takes as arguments an expression e that remains to be evaluated in the continuation of the computation, the current environment ρ, the current value store σ and the current continuation store Ξ. When using concrete semantics, value addresses and continuation addresses are typically represented by integers in the concrete state space (Van Horn and Might, 2010). They are allocated by returning the next available address.

λ_0

$$\begin{array}{rr} Addr = \mathbb{N} & alloc(ae, \sigma) = |\operatorname{dom}(\sigma)| \\ KAddr = \mathbb{N} & kalloc(e, \rho, \sigma, \Xi) = |\operatorname{dom}(\Xi)| \end{array}$$

Figure 2.5.: Allocation for λ_0.

Injection Function

The injection function $\mathcal{I} : Exp \to \Sigma$ depicted in Figure 2.6 injects a program e into an initial program state, where k_0 denotes a specific continuation address that is always

mapped to the empty continuation ϵ in the continuation store.

λ_0

$$\mathcal{I}(e) = \langle \mathbf{ev}(e, []), k_0 \rangle$$

Figure 2.6.: Injection function of λ_0.

Collecting Semantics

Concrete interpretation of a λ_0 program starts by applying the injection function to the program, resulting in an initial state ς. Next, the transition relation is applied repeatedly to this initial state, with an initial value store and an initial continuation store. The transition relation is applied until a final state is reached. A final state is a value state (**val**) where the current continuation address is the address of the empty continuation (k_0). As no transition rule is applicable anymore to this state, it represents the end of the execution of the program. The initial value store (denoted $[]$) is empty and the initial continuation store (denoted $[k_0 \mapsto \epsilon]$) maps k_0 to the empty continuation ϵ. The concrete interpretation of a program starts with the injection of the program into an initial state, followed by the application of the transition relation until no more rules can be applied. The concrete run of an example program is depicted in Figure 2.7.

In order to define concrete interpretation formally, we use *collecting semantics* (Cousot and Cousot, 1977), as it accounts for future abstraction and for possible non-determinism in the semantics. The collecting semantics of a program is the set of all the states reachable during the execution of this program. Collecting semantics enable modeling non-determinism, which is necessary for abstract interpretation where non-determinism results from approximations. Non-determinism is inherent to concurrent programs, so collecting semantics are a natural way to describe *all possible executions* of a concurrent programs. We define the collecting semantics through a transfer function $\mathcal{F}_e^{\lambda_0} : \mathcal{P}(\Sigma \times \mathit{Store} \times \mathit{KStore}) \to \mathcal{P}(\Sigma \times \mathit{Store} \times \mathit{KStore})$ parameterized by the program e to execute, and aggregating reachable states with their corresponding stores. This transfer function is depicted in Figure 2.8.

λ_0

$$\mathcal{F}_e^{\lambda_0}(S) = \{(\mathcal{I}(e), [], [k_0 \mapsto \epsilon])\} \quad (1)$$

$$\cup \bigcup_{\substack{(\varsigma,\sigma,\Xi)\in S \\ \varsigma,\sigma,\Xi \hookrightarrow \varsigma',\sigma',\Xi'}} (\varsigma', \sigma', \Xi') \quad (2)$$

Figure 2.8.: Transfer function for λ_0.

The transfer function $\mathcal{F}_e^{\lambda_0}$ defines that the following types of states are reachable from

$$
\begin{array}{ll}
 & \langle \mathbf{ev}(\texttt{(letrec ((f (}\lambda\texttt{ (x) (x x)))) (f (}\lambda\texttt{ (y) y)))}, []), k_0\rangle, \\
 & [], [k_0 \mapsto \epsilon] \\
\xhookrightarrow{\textsc{Letrec1}} & \langle \mathbf{ev}(\texttt{(}\lambda\texttt{ (x) (x x))}, [\texttt{f} \mapsto a_f]), k_1\rangle, \\
 & [], [k_0 \mapsto \epsilon, k_1 \mapsto \mathbf{letrec}(\texttt{(}f(\lambda\ (y)y)), a_f, [\texttt{f} \mapsto a_f]) : k_0] \\
\xhookrightarrow{\textsc{Atomic}} & \langle \mathbf{val}(\mathbf{clo}(\texttt{(}\lambda\texttt{ (x) (x x))}, [\texttt{f} \mapsto a_f])), k_1\rangle, \\
 & [], [k_0 \mapsto \epsilon, k_1 \mapsto \ldots] \\
\xhookrightarrow{\textsc{Letrec2}} & \langle \mathbf{ev}(\texttt{(f (}\lambda\texttt{ (y) y))}, [\texttt{f} \mapsto a_f]), k_0\rangle, \\
 & [a_f \mapsto \mathbf{clo}(\texttt{(}\lambda\texttt{ (x) (x x))}, [\texttt{f} \mapsto a_f])], [k_0 \mapsto \epsilon, k_1 \mapsto \ldots] \\
\xhookrightarrow{\textsc{App}} & \langle \mathbf{ev}(\texttt{(x x)}, [\texttt{f} \mapsto a_f, \texttt{x} \mapsto a_x]), k_0\rangle, \\
 & [a_f \mapsto \mathbf{clo}(\texttt{(}\lambda\texttt{ (x) (x x))}, [\texttt{f} \mapsto a_f]), \\
 & \ a_x \mapsto \mathbf{clo}(\texttt{(}\lambda\texttt{ (y) y)}, [\texttt{f} \mapsto a_f])], [k_0 \mapsto \epsilon, k_1 \mapsto \ldots] \\
\xhookrightarrow{\textsc{App}} & \langle \mathbf{ev}(\texttt{y}, [\texttt{f} \mapsto a_f, \texttt{y} \mapsto a_y]), k_0\rangle, \\
 & [a_f \mapsto \mathbf{clo}(\texttt{(}\lambda\texttt{ (x) (x x))}, [\texttt{f} \mapsto a_f, \texttt{x} \mapsto a_x]), \\
 & \ a_x \mapsto \mathbf{clo}(\texttt{(}\lambda\texttt{ (y) y)}, [\texttt{f} \mapsto a_f]), \\
 & \ a_y \mapsto \mathbf{clo}(\texttt{(}\lambda\texttt{ (y) y)}, [\texttt{f} \mapsto a_f])], [k_0 \mapsto \epsilon, k_1 \mapsto \ldots] \\
\xhookrightarrow{\textsc{Atomic}} & \langle \mathbf{val}(\mathbf{clo}(\texttt{(}\lambda\texttt{ (y) y)}, [\texttt{f} \mapsto a_f])), k_0\rangle, \\
 & [a_f \mapsto \mathbf{clo}(\texttt{(}\lambda\texttt{ (x) (x x))}, [\texttt{f} \mapsto a_f, \texttt{x} \mapsto a_x]), \\
 & \ a_x \mapsto \mathbf{clo}(\texttt{(}\lambda\texttt{ (y) y)}, [\texttt{f} \mapsto a_f]), \\
 & \ a_y \mapsto \mathbf{clo}(\texttt{(}\lambda\texttt{ (y) y)}, [\texttt{f} \mapsto a_f])], [k_0 \mapsto \epsilon, k_1 \mapsto \ldots]
\end{array}
$$

Figure 2.7.: Trace of a concrete execution of the λ_0 program `(letrec ((f (λ (x) (x x)))) (f (lambda (y) y)))`.

a given set of states S:

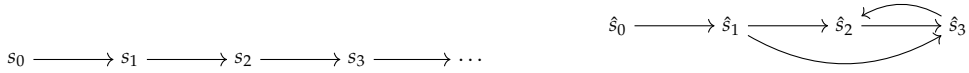
1. the initial injected state, with the initial empty value store and initial continuation store mapping the k_0 continuation address to the empty continuation, and
2. states that can be reached from reachable states, with the corresponding stores.

The collecting semantics is then defined as the least fixed point of this transfer function, $\mathrm{lfp}(\mathcal{F}_e^{\lambda_0})$, which is the set of all states reachable during all possible executions of program e.

2.1.3. Abstract Semantics of λ_0

The collecting semantics for a λ_0 program describes all states that are reachable during any execution of the program. However, this set of states may be infinite and not computable in the general case. A static analysis has to be able to analyze *any* program of the language under analysis. Relying on the concrete collecting semantics for static analysis is therefore not feasible. This is why we introduce an abstracted version of the λ_0 collecting semantics, which is appropriate for static analysis and follows the principles of abstract interpretation (Cousot and Cousot, 1977).

Through abstraction, possibly infinite behaviors of a program are rendered finite, enabling static analysis. The resulting abstract semantics over-approximate the concrete semantics in a sound manner. For example, the possibly infinite trace of a program (depicted in Figure 2.9a) may be approximated by a graph that over-approximates all possible concrete traces (depicted in Figure 2.9b). We call such a graph a *flow graph*.



(a) Trace of the program execution under concrete semantics.

(b) Flow graph approximating the program execution under abstract semantics.

Figure 2.9.: Approximation of a program trace through abstract interpretation.

This abstraction is performed through the *Abstracting Abstract Machines* method, abbreviated AAM (Van Horn and Might, 2010). The AAM method is a systematic approach to designing abstract interpretations of higher-order languages. This method can be seen as a recipe to turn language semantics expressed as an abstract machine into a finite version of this semantics called an *abstracted* abstract machine. The recipe requires allocating the elements of every infinite component of the state space at an address in a store and limiting the set of addresses to a finite set. AAM then guarantees that the analysis *terminates* when it is expressed through a monotone transfer function.

Besides termination, another crucial property is *soundness*. Soundness means that the results computed by the analysis of a program hold for any possible execution of the program, and can therefore be trusted. A sound analysis may over-approximate information, leading to potential false positives (e.g., the analysis reports potential errors that in reality may never arise), but can never under-approximate information (e.g., the

analysis reports no potential errors while some execution of the program under analysis may lead to an error).

Addresses are generated according to an *allocation strategy*, which is the only component that has to be tuned to obtain a finite state space once the semantics has been abstracted. This allocation strategy is defined by the *alloc* and *kalloc* allocation functions (Figure 2.5). The allocation strategy influences the precision of the analysis (Gilray *et al.*, 2016a), but not its soundness, as any allocation strategy leads to a sound analysis (Might and Manolios, 2009). We can therefore prove the soundness of the analysis without considering the allocation strategy used. We do not study the impact of allocation strategies in this dissertation as this discussion is orthogonal to the contributions of our work.

In our formalization of the abstract semantics, we highlight in gray the main changes with respect to the concrete semantics. The abstracted version of a concrete component is denoted by the same name decorated with a hat.

Abstract State Space

Figure 2.10 depicts the abstract state space for λ_0. In order to abstract the semantics, the sets of addresses (value addresses *Addr* and continuation addresses *KAddr*) are rendered finite. This simple change propagates through the state space. Abstract environments $\hat{\rho}$ map variable names to *abstract* addresses $\hat{a}$. Abstract closures **clo** contain abstract environments $\hat{\rho}$. Abstract control components $\hat{c}$ may contain abstract values $\hat{v}$ and environments $\hat{\rho}$. Abstract frames $\hat{\phi}$ contain abstract addresses $\hat{a}$ and abstract environments $\hat{\rho}$. Abstract continuations $\hat{\kappa}$ contain abstract continuation addresses $\widehat{k}$. The major changes happen in the stores: because the set of abstract value addresses and abstract continuation addresses are finite, both the value store and the continuation store have to map respectively to *sets* of values and to *sets* of continuations. This introduces non-determinism in the semantics: looking up a value or a continuation in a store may yield multiple results. This non-determinism may lead to a precision loss in the analysis, which is the price to pay for termination of the analysis.

λ_0

$$\hat{\varsigma} \in \widehat{\Sigma} = \widehat{Control} \times \widehat{KAddr}$$

$$\hat{c} \in \widehat{Control} ::= \mathbf{ev}(e, \hat{\rho}) \mid \mathbf{val}(\hat{v}) \mid \mathbf{err}$$

$$\hat{\rho} \in \widehat{Env} = Var \to \widehat{Addr}$$

$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \mathcal{P}(\widehat{Val})$$

$$\widehat{\Xi} \in \widehat{KStore} = \widehat{KAddr} \to \mathcal{P}(\widehat{Kont})$$

$$\hat{\kappa} \in \widehat{Kont} ::= \hat{\phi} : \widehat{k} \mid \epsilon$$

$$\hat{\phi} \in \widehat{Frame} ::= \mathbf{letrec}(e, \hat{a}, \hat{\rho})$$

$$\hat{v} \in \widehat{Val} ::= \mathbf{clo}(lam, \hat{\rho})$$

$\hat{a} \in \widehat{Addr}$ a finite set of addresses

$\widehat{k} \in \widehat{KAddr}$ a finite set of addresses

Figure 2.10.: Abstract state space for λ_0.

Abstract Atomic Evaluation

The abstract version of atomic evaluation depicted in Figure 2.11 now accounts for the fact that one address in the value store may map to more than one possible value. The atomic evaluation relation hence becomes non-deterministic as seen by the membership operation in rule Var. Rule Lambda remains the same under abstraction.

λ_0

$$\frac{\hat{v} \in \hat{\sigma}(\hat{\rho}(x))}{\hat{\rho}, \hat{\sigma} \vdash x \mathbin{\widehat{\Downarrow}} \hat{v}}\ \textsc{Var} \qquad \frac{}{\hat{\rho}, \hat{\sigma} \vdash lam \mathbin{\widehat{\Downarrow}} \mathbf{clo}(lam, \hat{\rho})}\ \textsc{Lambda}$$

Figure 2.11.: Abstract atomic evaluation relation for λ_0.

Abstract Allocation

As noted before, any allocation strategy leads to a sound analysis (Might and Manolios, 2009). We depict in Figure 2.12 the allocation strategy introduced by Gilray *et al.* (2016b) that leads to a flow-sensitive, context-insensitive 0-CFA analysis, with precise call and return matching.

λ_0

$$\hat{a} \in \widehat{Addr} = Exp \qquad \widehat{alloc}(e, \hat{\sigma}) = e$$
$$\widehat{k} \in \widehat{KAddr} = Exp \times \widehat{Env} \qquad \widehat{kalloc}(e, \hat{\rho}, \hat{\sigma}, \widehat{\Xi}) = (e, \hat{\rho})$$

Figure 2.12.: Abstract allocation functions for λ_0.

Abstract Transition Relation.

The abstract transition relation (Figure 2.13) is adapted to the abstract state space. Values stored in the value store are now *sets* of values, and updates to the value store now become *join* operations, as expressed by the $\sqcup$ operator (see Appendix A), in order to group values that share the same address. Similarly, the continuation store now maps to *sets* of continuations, and updates to the continuation store also become *join* operations for the same reason.

λ_0

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \hat{v}}{\langle \mathbf{ev}(ae, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \leadsto \langle \mathbf{val}(\hat{v}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{Atomic}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash f \mathbin{\widehat{\Downarrow}} \mathbf{clo}((\lambda\ (x)\ e), \hat{\rho}') \qquad \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \hat{v} \qquad \hat{a} = \widehat{alloc}(x, \hat{\sigma})}{\langle \mathbf{ev}((f\ ae), \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \leadsto \langle \mathbf{ev}(e, \hat{\rho}'[x \mapsto \hat{a}]), \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}} \textsc{App}$$

$$\frac{\hat{a} = \widehat{alloc}(x, \hat{\sigma}) \qquad \widehat{k}' = \widehat{kalloc}(e_2, \hat{\rho}, \hat{\sigma}, \widehat{\Xi}) \qquad \hat{\rho}' = \hat{\rho}[x \mapsto \hat{a}]}{\begin{array}{c}\langle \mathbf{ev}((\texttt{letrec}\ ((x\ e_1))\ e_2), \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \leadsto \\ \langle \mathbf{ev}(e_1, \hat{\rho}'), \widehat{k}' \rangle, \hat{\sigma}, \widehat{\Xi} \sqcup [\widehat{k}' \mapsto \left\{\mathbf{letrec}(e_2, \hat{a}, \hat{\rho}') : \widehat{k}\right\}]\end{array}} \textsc{Letrec1}$$

$$\frac{\widehat{\Xi}(\widehat{k}) \ni \mathbf{letrec}(e, \hat{a}, \hat{\rho}) : \widehat{k}'}{\langle \mathbf{val}(\hat{v}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \leadsto \langle \mathbf{ev}(e, \hat{\rho}), \widehat{k}' \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}} \textsc{Letrec2}$$

$$\frac{}{\langle \mathbf{ev}(\texttt{(error)}, \rho), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \leadsto \langle \mathbf{err}, \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{Error}$$

Figure 2.13.: Abstract transition relation for λ_0.

Abstract Injection

The abstract injection function $\widehat{\mathcal{I}} : Exp \to \widehat{\Sigma}$, depicted in Figure 2.14 is adapted from the concrete semantics by using a specific abstract continuation address $\widehat{k}_0$ for the address of the empty continuation.

λ_0

$$\widehat{\mathcal{I}}(e) = \langle \mathbf{ev}(e, []), \widehat{k}_0 \rangle$$

Figure 2.14.: Abstract injection function of λ_0.

Abstract Collecting Semantics

The abstract collecting semantics (Figure 2.15) are adapted from the concrete collecting semantics by using the abstract transition relation. Note that because addresses are finite, all other components of the state space become finite as well, rendering the transfer function $\widehat{\mathcal{F}}_e^{\lambda_0}$ monotone and the abstract collecting semantics computable (see Section 2.1.4 for the proof).

λ_0

$$\widehat{\mathcal{F}}_e^{\lambda_0}(S) = \left\{(\widehat{\mathcal{I}}(e), [], [\widehat{k}_0 \mapsto \epsilon])\right\} \quad (1)$$

$$\cup \bigcup_{\substack{(\hat{\varsigma},\hat{\sigma},\widehat{\Xi})\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \,\widehat{\leadsto}\, \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'}} (\hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}') \quad (2)$$

Figure 2.15.: Abstract transfer function for λ_0.

The fixed point of the transfer function for a program e, $\mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_0})$, is an over-approximation of the set of reachable states in all possible executions of the program e (Theorem 1). The abstract collecting semantics therefore forms a sound foundation for building static analyses. For example, one can adapt its definition to include not only reachable states but also the transitions taken between the states, resulting in a flow graph such as depicted in Figure 2.9.

Another variant of the collecting semantics can be obtained by applying *global-store widening* (Van Horn and Might, 2010). Global-store widening further abstracts the domain of the transfer function to a set of reachable program states, a single global value store, and a single continuation store, i.e. using the domain $\mathcal{P}(\widehat{\Sigma}) \times \widehat{Store} \times \widehat{KStore}$ instead of $\mathcal{P}(\widehat{\Sigma} \times \widehat{Store} \times \widehat{KStore})$. This reduces the precision of the resulting analysis, for an improved worst-case time complexity, which becomes polynomial in the size of the program under analysis (Van Horn and Might, 2010). We systematically apply this global-store widening in the implementation of all analyses developed in this dissertation. Using the allocation strategy presented in Figure 2.12 and global-store widening, the worst-case time complexity of the resulting analysis is of $\mathcal{O}(|Exp|^3)$ (Gilray *et al.*, 2016b), i.e., polynomial in the size of the analyzed program.

2.1.4. Soundness and Termination

For a static analysis to be able to prove properties about a program, it must be *sound*, and for it to support analyzing any program statically, it must always *terminate*. The analysis described here satisfies both properties, and soundness and termination of analyses resulting from applying AAM have been proven by Van Horn and Might (2010). To prove both properties formally, we have to introduce an ordering relation $\sqsubseteq$, which, with the domain of the transfer function ($\mathcal{P}(\widehat{\Sigma} \times \widehat{Store} \times \widehat{KStore})$), forms a finite lattice. Moreover, to prove soundness formally, we introduce the abstraction function used to abstract of the state space, $\alpha : (\Sigma \times Store \times KStore) \to (\widehat{\Sigma} \times \widehat{Store} \times \widehat{KStore})$. We provide the definition of this ordering relation and abstraction function in Appendix B.1.1.

Theorem 1 (Soundness)**.** $\mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_0})$ *is a sound over-approximation of* $\mathrm{lfp}(\mathcal{F}_e^{\lambda_0})$.

Proof. The proof is detailed in Appendix B.1.1. The idea of the proof is the following: we want to show that $\forall S, \alpha(\mathcal{F}_e^{\lambda_0}(S)) \sqsubseteq \widehat{\mathcal{F}}_e^{\lambda_0}(\alpha(S))$. This can be reduced to showing that the

abstract transition relation soundly over-approximates the concrete transition relation: if $\varsigma, \sigma, \Xi \hookrightarrow \varsigma', \sigma', \Xi'$, then $\exists \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'$ such that $\alpha(\varsigma, \sigma, \Xi) \widehat{\hookrightarrow} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}' \wedge \alpha(\varsigma', \sigma', \Xi') \sqsubseteq \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'$. This is shown by a case analysis over the rules of the transition relation. □

Theorem 2 (Termination)**.** *The computation of* $\mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_0})$ *always terminates.*

Proof. The proof is detailed in Appendix B.1.1. It entails showing that $\widehat{\mathcal{F}}_e^{\lambda_0}$ is a monotone transfer function, and that the lattice $(\mathcal{P}(\widehat{\Sigma} \times \widehat{Store} \times \widehat{KStore}), \sqsubseteq)$ is finite. By Tarski's fixed-point theorem (Tarski, 1955), the least-fixed point of a monotone function over a finite lattice will be reached in a finite number of steps. Hence, the abstract interpretation always terminates. □

2.2. The Actor Model: λ_α

The actor model is a widely used model of concurrency introduced by Hewitt *et al.* (1973) and further developed by Agha (1986). We formalize an actor extension to λ_0, as the λ_α language. Before introducing λ_α, we first give a high-level overview of an actor extension to Scheme. The λ_α language embodies the core concepts of this extension.

2.2.1. Overview of Actors

In the actor model, a program is composed of entities called *actors* that can communicate with each other by sending asynchronous messages. Actors are executed concurrently to each other in an isolated manner, and each actor has a specific behavior according to which it processes messages. This behavior can change dynamically. Actors receive messages in their mailbox. Once a message is ready to be processed by an actor, the actor extracts the message from the mailbox and executes the code defined in a message handler associated with this type of message. Mailboxes are ordered queues of messages, to which messages are enqueued in the front and from which messages are dequeued from the back. When processing a message, actors can perform a number of operations, such as dynamically creating other actors, sending messages to other actors or to themselves, or dynamically updating their behavior, thereby changing how future messages will be processed.

Defining actor behaviors

Actor behaviors are defined through the `actor` construct. This construct first defines the state variables that will form the internal state of the actor. These state variables are not mutable, but can be updated when changing the behavior of an actor, through the `become` construct. The `actor` construct also defines a number of message handlers, one for each type of message that the actor can receive. A message is composed of a selector, or *tag*, and a number of message arguments. When an actor receives a message, the body of the corresponding message handler, selected according to the tag of the message, will be executed in an environment extended with the formal parameters of

the handler bound to the arguments given in the message. Note that this merely defines an actor behavior, but does not create an actor yet.

```
1 (define an-actor
2   (actor (state-var1 state-var2)
3     (tag1 (a b) body1)
4     (tag2 (c) body2)))
```

Creating actors.

An actor is created, or *spawned*, through the `create` construct, which has to be provided as arguments the behavior to instantiate for the new actor, and the initial values for the state variables of this actor. In the code fragment below, actor `A` is spawned with the previously defined behavior `an-actor`, and with state variable `state-var1` bound to `1`, and `state-var2` bound to `2`.

```
1 (define A (create an-actor 1 2))
```

Sending messages.

Messages are sent using the `send` construct, where the target actor is given as first argument, followed by the tag of the message to send and by the message arguments. When actor `A` receives the message `tag1(#t, 'hello)`, it will execute `body1` in an environment extended with `a` bound to `#t` and `b` bound to `'hello`. Messages are sent asynchronously; a call to `send` returns immediately and does not block the sending actor until the target actor has processed the sent message.

```
1 (send A tag1 #t 'hello)
```

Changing the behavior of an actor.

The `become` construct changes the behavior of an actor dynamically, and updates the state variables of the actor. In the following snippet, the actor with the `an-actor` behavior will update its state variables upon the receipt of a `tag1` message, by binding `state-var1` to the value of the first argument of the message, and leaving `state-var2` unchanged. An actor can also change its behavior to a different behavior, as shown in the body of the message handler for `tag2`.

```
1 (define an-actor
2   (actor (state-var1 state-var2)
3     (tag1 (a b) (become an-actor a state-var2))
4     (tag2 (c) (become another-actor))))
5 (define A (create an-actor 1 2))
```

Note that, for the sake of simplicity but without loss of generality, we require the body of all message handlers to end with a `become` statement. In real-world actor languages, it is assumed that if no `become` statement is executed in the body of a message handler, the actor does not change its behavior.

Example actor program.

We give a more involved example actor program in Listing 2.1, adapted from Agha (1986). This program computes and displays the factorial of a number read from user input. After the definition of three behaviors (lines 1, 10, and 15), two actors are created: actor `fact` with behavior `fact-actor` (line 20), and actor `displayer` that displays the messages it receives (line 21). On line 22 the number returned by `read-integer` (assumed non-negative) is sent to actor `fact` along with actor `displayer`, the *customer* that will receive the answer that `fact` computes.

When actor `fact` receives the `compute` message with $n = 0$, it sends 1 as answer to the customer (line 5). Otherwise $n \neq 0$, and `fact` spawns an actor with behavior `customer-actor` (line 7), passing n and `fact`'s customer as arguments. Actor `fact` then sends itself a `compute` message to compute $(n-1)!$, passing along the newly created customer (line 7). Actors with the `customer-actor` behavior multiply the number they receive by the number given at their creation, and send the result to the customer given at their creation (line 12).

Putting everything together, when `fact` receives the initial `compute` message with number n and customer `displayer`, it creates a customer `c` with n and `displayer` as arguments. This customer represents the continuation of the factorial computation. Actor `fact` then computes $(n-1)!$ by sending a `compute` message to itself with $n-1$ and `c` as arguments. When the computation of $(n-1)!$ is completed, the result is sent to customer `c`, which computes $n * (n-1)! = n!$, and sends the result to `displayer`, which displays the result on the screen.

```
1  (define fact-actor
2    (actor ()
3      (compute (n customer)
4        (if (= n 0)
5          (send customer result 1)
6          (let ((c (create customer-actor
7                           n customer)))
8            (send self compute (- n 1) c)))
9        (become fact-actor))))
10 (define customer-actor
11   (actor (n customer)
12     (result (k)
13       (send customer result (* n k))
14       (become customer-actor n customer))))
15 (define displayer-actor
16   (actor ()
17     (result (v)
18       (display v)
19       (become displayer-actor))))
20 (define fact (create fact-actor))
21 (define displayer (create displayer-actor))
22 (send fact compute (read-integer) displayer)
```

Listing 2.1: Program computing the factorial of a user-given number with actors, adapted from Agha (1986).

Figure 2.16 depicts the evolution of the actor topology throughout the execution of the factorial program. We see that the `fact` actor creates a chain of customers that propagate and multiply a value until the resulting value (the computed factorial) reaches the `display` actor.

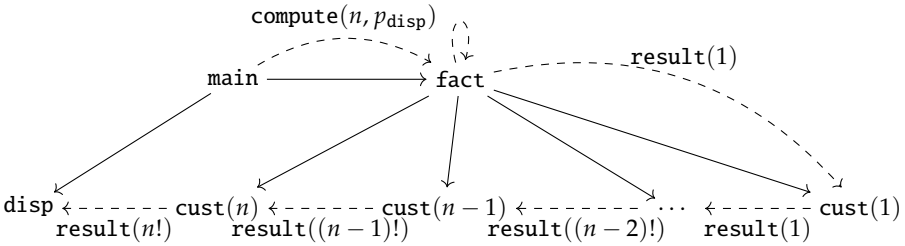


Figure 2.16.: Representation of the actor topology during the execution of the factorial program in Listing 2.1. Plain edges represent actor creation, and dashed edges represent message sends. When annotated, the dashed edges indicate the message being sent. Actors denoted by `disp` and `cust` respectively denote actors with behaviors `displayer-actor` and `customer-actor`. The dots in the chain of `cust` actors represent a number of actors that depend on the value of n.

2.2.2. Syntax of λ_α

Figure 2.17 depicts the syntax of λ_α. To model actor concurrency formally, we extend λ_0 with the concepts of actor definition, creation and evolution and with messages.

1. **Actor definition, creation and evolution**. Actor behaviors are defined through the `actor` construct, and actors are created by instantiating an actor behavior with the `create` construct, which creates a new process. Actors have but one *state variable*, which characterizes the state of the actor. We limit the model to a single state variable, without loss of generality. An actor's behavior and state variable can be updated through the `become` construct.

2. **Messages**. Actors can send messages with the `send` construct, and receive messages in their mailbox. Each actor defines message handlers for the types of message it can receive. Each message carries a *tag*, which will be used to select the message handler that handles this type of message when received, as well as an argument. We limit the model to a single argument per message, without loss of generality.

Like variable names, tags are syntactic elements of which there is a finite number within a program. To simplify the presentation, but without loss of generality, λ_α is limited to actors with one state variable, and messages with one argument. Our implementation however supports an arbitrary number of state variables and an arbitrary number of message arguments. Note that the dotted parts (...) of the following definitions refer to the fact that we extend the syntax and semantics of λ_0. For example, atomic expressions correspond to the usual atomic expressions of λ_0 (variables and lambda expressions), extended with the notion of actor expression (*act*).

λ_α

$$
\begin{array}{lll}
e \in Exp ::= \ldots & & ae \in AExp ::= \ldots \mid act \\
\quad \mid \texttt{(create } ae\ ae\texttt{)} & & act \in Act ::= \texttt{(actor (}x\texttt{)} \\
\quad \mid \texttt{(send } ae\ t\ ae\texttt{)} & & \qquad\qquad \texttt{(}t\ \texttt{(}y\texttt{)}\ e\texttt{)}^*\texttt{)} \\
\quad \mid \texttt{(become } ae\ ae\texttt{)} & & t \in Tag \text{ a finite set of tags}
\end{array}
$$

Figure 2.17.: Syntax of λ_α.

We also define auxiliary functions in Figure 2.18 to extract specific components from expressions: VAR extracts the name of the state variable of an actor expression, and HANDLER extracts from an actor the handler for a specific tag from an actor expression.

λ_α

$$
\begin{aligned}
\textsc{var}(\texttt{(actor (}x\texttt{) ...)}) &= x \\
\textsc{handler}(\texttt{(actor (}x\texttt{) ... (}t\ \texttt{(}y\texttt{)}\ e\texttt{) ...)}, t) &= (y, e)
\end{aligned}
$$

Figure 2.18.: Auxiliary functions for λ_α.

2.2.3. Concrete Semantics of λ_α

We define the concrete semantics of λ_α as an extension to the semantics of λ_0. The semantics is defined by a concurrent transition relation ($\rightsquigarrow$), annotated with *communication effects*. Communication effects describe the concurrent operation performed within a transition. They are only used for an informational purpose in this chapter, but could be used to perform a communication topology analysis (Colby, 1995; Martel and Gengler, 2000), where the processes dynamically created in an concurrent program are identified along with their interactions. In Chapters 4 and 6, we use these communication effects to improve the scalability of the analysis.

State Space

Figure 2.19 depicts the state space for λ_α. As each actor has a specific behavior, a process state (ς) is a tuple containing a control component c, a current actor behavior b, and a continuation address k. The control component can comprise an additional value compared to λ_0: an actor can be **wait**ing for messages. Actor definitions (**actor**) and process identifiers (**pid**) are first-class values in the language, and are respectively the resulting value of the `actor` and `create` constructs. An actor behavior is either a user-defined actor definition paired with its extended environment (**beh**), or the behavior of the main actor (**main**). This distinction is used because the main actor, i.e., the actor that is initially spawned when running an actor program, acts differently: it executes a body of code and does not receive any messages. The global state of the computation is held in a process map (π), associating with each process identifier a current process

state and a mailbox. We discuss possible instantiations of process identifiers later in this section. A mailbox (*mb*) is an ordered sequence of messages (*m*), where each message is a pair of a tag and an argument value.

λ_α

$$\begin{aligned}
\pi \in \Pi &= PID \to (\Sigma \times Mbox) \\
\varsigma \in \Sigma &= Control \times Beh \times KAddr \\
c \in Control &::= \ldots \mid \mathbf{wait} \\
v \in Val &::= \ldots \mid \mathbf{pid}(p) \mid \mathbf{actor}(act, \rho) \\
b \in Beh &::= \mathbf{beh}(act, \rho) \mid \mathbf{main} \\
mb \in Mbox &= Message^* \\
m \in Message &= Tag \times Val \\
p \in PID &\text{ a set of process identifiers}
\end{aligned}$$

Figure 2.19.: State space for λ_α.

Communication Effects

Figure 2.20 depicts the communication effects for λ_α. Actors can generate four types of communication effects during their execution. Creation effects $\mathbf{c}(p, \varsigma)$ are generated when an actor creates another actor, with initial state ς and process identifier p. An actor can also send a message with tag t and argument v to an actor with process identifier p, generating a $\mathbf{snd}(p, t, v)$ communication effect. An actor that starts processing a message with tag t and argument v generates a $\mathbf{prc}(t, v)$ communication effect. An actor changing its behavior to b with state variable set to value v generates a $\mathbf{b}(b, v)$ communication effect.

λ_α

$$eff \in Effect ::= \mathbf{c}(p, \varsigma) \mid \mathbf{snd}(p, t, v) \mid \mathbf{prc}(t, v) \mid \mathbf{b}(b, v)$$

Figure 2.20.: Communication effects for λ_α.

Mailboxes

Figure 2.21 depicts the mailboxes of λ_α. Mailboxes are queues of messages in the formal model. When a message is sent to an actor, it is enqueued in the front of the mailbox, and when a message is processed by the actor, it is dequeued from the back of the mailbox. This behavior is encoded in the *enq* and *deq* functions. The *deq* function returns a set of pairs of dequeued messages and resulting mailboxes. In a concrete setting, this set is either a singleton set to represent a successful dequeue, or empty to represent the

absence of messages to dequeue. The *size* function computes the size of a mailbox, and the *empty* symbol denotes the empty mailbox.

λ_α

$$enq(m, mb) = m : mb \qquad deq(\epsilon) = \varnothing$$
$$size(mb) = |mb| \qquad deq(mb : m) = \{(m, mb)\}$$
$$empty = \epsilon$$

Figure 2.21.: Mailboxes of λ_α.

Atomic Evaluation

Figure 2.22 depicts the atomic evaluation relation for λ_α. The semantics for λ_α extends the atomic evaluation relation with support for actor definitions: evaluating an actor definition (*act*) yields an **actor** value that pairs the definition with the binding environment (rule Actor). Rules Var and Lambda from λ_0 still apply.

λ_α

$$\frac{}{\rho, \sigma \vdash b \Downarrow \mathbf{actor}(act, \rho)}\ \textsc{Actor}$$

Figure 2.22.: Atomic evaluation relation for λ_α.

Transition Relation

The semantics of λ_α is formalized as a concurrent transition relation denoted $\rightsquigarrow$. This transition relation always carries the process identifier p of the actor performing the transition ($\rightsquigarrow_p$), and may also carry a communication effect ($\overset{eff}{\rightsquigarrow}_p$). The process p that transitions at a given time in the execution of the program is selected by the transfer function, defined later in this section. The concurrent transition relation acts on process maps π, value stores σ, and continuation stores Ξ, and a transition is written $\pi, \sigma, \Xi \rightsquigarrow_p \pi', \sigma', \Xi'$. We explicitly mention in the text when we use the notation $\rightsquigarrow_p$ for a transition that may generate any effect, otherwise this notation states that no communication effect is generated.

Sequential transitions. The Seq rule depicted in Figure 2.23 applies the transition relation of the sequential subset of the language, λ_0, to actor transitioning, leaving its mailbox untouched. No effect is generated.

λ_α

$$\frac{\pi(p) = (\varsigma, mb) \qquad \varsigma, \sigma, \Xi \hookrightarrow \varsigma', \sigma', \Xi'}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto (\varsigma', mb)], \sigma', \Xi'} \textsc{Seq}$$

Figure 2.23.: Sequential transitions for λ_α.

Actor management transitions. Figure 2.24 depicts the transition rules for actor management in λ_α. To create an actor, the behavior and state variable provided to `create` are evaluated with the atomic evaluation relation. The new actor starts in a waiting state, with an empty continuation stack and an empty mailbox. A process identifier is allocated for the new actor, and the initial state of the actor is added to the process map, with its environment extended to map its state variable to the address at which the value of the argument given to `create` resides. The actor performing the `create` reaches as value the process identifier of the created actor, and a **c** communication effect is generated (rule Create). An actor can change its behavior through a `become` statement, which binds the state variable to its new value, and sets the actor to a **wait** state, generating a **b** communication effect (rule Become).

λ_α

$$\frac{\begin{array}{c}\pi(p) = (\langle \mathbf{ev}(\texttt{(create } ae\ ae'\texttt{)}, \rho), b, k\rangle, mb) \qquad \rho, \sigma \vdash ae \Downarrow \mathbf{actor}(act, \rho') \\ \rho, \sigma \vdash ae' \Downarrow v \qquad \varsigma' = \langle \mathbf{wait}, \mathbf{beh}(act, \rho'[x \mapsto a]), k_0\rangle \\ a = alloc(x, \sigma) \qquad p' = palloc(\varsigma', \pi) \qquad x = \textsc{var}(act)\end{array}}{\begin{array}{c}\pi, \sigma, \Xi \overset{\mathbf{c}(p', \varsigma')}{\rightsquigarrow}_p \\ \pi[p \mapsto (\langle \mathbf{val}(\mathbf{pid}(p')), b, k\rangle, mb), p' \mapsto (\varsigma', empty)], \sigma[a \mapsto v], \Xi\end{array}} \textsc{Create}$$

$$\frac{\begin{array}{c}\pi(p) = (\langle \mathbf{ev}(\texttt{(become } ae\ ae'\texttt{)}, \rho), b, k\rangle, mb) \\ \rho, \sigma \vdash ae \Downarrow \mathbf{actor}(act, \rho') \qquad \rho, \sigma \vdash ae' \Downarrow v \\ b' = \mathbf{beh}(act, \rho'[x \mapsto a]) \qquad x = \textsc{var}(act) \qquad a = alloc(x, \sigma)\end{array}}{\pi, \sigma, \Xi \overset{\mathbf{b}(b', v)}{\rightsquigarrow}_p \pi[p \mapsto (\langle \mathbf{wait}, b', k\rangle, mb)], \sigma[a \mapsto v], \Xi} \textsc{Become}$$

Figure 2.24.: Actor management transitions rules for λ_α.

Messages transitions. Figure 2.25 depicts the transition rules for messages in λ_α. To evaluate a `send` construct, the transition relation evaluates its first argument to the target actor's process identifier using atomic evaluation, evaluates the message argument using atomic evaluation, adds the message to the mailbox of the target actor, and generates the corresponding **snd** communication effect (rule Send). Note that the actor sending the message might also be the actor receiving the message. This is accounted for in

the enqueuing of messages by first updating the control component of the process p sending the message without modifying its mailbox mb, resulting in a process map π', and only then updating the mailbox mb' of the target process p' separately. This rule can therefore apply if p and p' are different as well as if they correspond to the same process.

An actor that starts processing a message extracts the received message from its mailbox through function *deq*, selects the handler corresponding to the tag of the received message, binds the parameter of the handler to the received value, and proceeds by evaluating the handler body, generating a **prc** communication effect (rule Process). If the mailbox of an actor is empty, *deq* returns an empty set and rule Process cannot apply. This rule models the execution of actors that are in a waiting state and poll their mailbox until a message can be processed.

λ_α

$$\frac{\pi(p) = (\langle \mathbf{ev}(\texttt{(send}\ ae\ t\ ae'\texttt{)}, \rho), b, k\rangle, mb) \qquad \rho, \sigma \vdash ae \Downarrow \mathbf{pid}(p') \qquad \pi' = \pi[p \mapsto (\langle \mathbf{val}(v), b, k\rangle, mb)] \qquad \pi'(p') = (\varsigma, mb') \qquad \rho, \sigma \vdash ae' \Downarrow v}{\pi, \sigma, \Xi \overset{\mathbf{snd}(p', t, v)}{\rightsquigarrow}_p \pi'[p' \mapsto (\varsigma, enq((t, v), mb'))], \sigma, \Xi} \textsc{Send}$$

$$\frac{\pi(p) = (\langle \mathbf{wait}, \mathbf{beh}(act, \rho), k\rangle, mb) \qquad \{((t, v), mb')\} = deq(mb) \qquad (y, e) = \textsc{handler}(act, t) \qquad a = alloc(y, \sigma)}{\pi, \sigma, \Xi \overset{\mathbf{prc}(t, v)}{\rightsquigarrow}_p \pi[p \mapsto (\langle \mathbf{ev}(e, \rho[y \mapsto a]), \mathbf{beh}(act, \rho), k\rangle, mb')], \sigma[a \mapsto v], \Xi} \textsc{Process}$$

Figure 2.25.: Message transitions for λ_α.

Process Identifiers

Figure 2.26 depicts the process identifiers for λ_α. Allocation of process identifiers is performed by the *palloc* function, which takes as argument the actor state ς of the created actor, and the current process map π. A straightforward process identifier allocation strategy in a concrete setting is to represent process identifiers as natural numbers and to allocate them by generating the lowest available process identifier. The size of the process map ($|\pi|$) only increases during the program execution.

λ_α

$$p \in PID = \mathbb{N} \qquad palloc(\varsigma, \pi) = |\pi|$$

Figure 2.26.: Process identifiers for λ_α.

Injection Function

The injection function depicted in Figure 2.27 injects the program to evaluate into an initial process map containing only the main actor, associated with the initial process identifier p_0. The main actor evaluates the expression representing the program and has an empty mailbox.

λ_α

$$\mathcal{I}(e) = [p_0 \mapsto (\langle \mathbf{ev}(e, []), \mathbf{main}, k_0 \rangle, \mathit{empty})]$$

Figure 2.27.: Injection function for λ_α.

Collecting Semantics

The collecting semantics is similar to the one for λ_0 programs, and is defined by the transfer function depicted in Figure 2.28. States explored by the transfer function are tuples $\langle \pi, \sigma, \Xi \rangle$ containing a process map π encoding multiple actor states, a value store σ and a continuation store Ξ, hence $\mathcal{F}_e^{\lambda_\alpha} : \mathcal{P}(\Pi \times \mathit{Store} \times \mathit{KStore}) \to \mathcal{P}(\Pi \times \mathit{Store} \times \mathit{KStore})$. The initial injected state is reachable (1), and any state resulting from the application of a a transition for any running actor on a reachable state is also reachable in the collecting semantics (2). We use $\rightsquigarrow_p$ here as notation for a transition that may generate any effect or no effect.

λ_α

$$\mathcal{F}_e^{\lambda_\alpha}(S) = \{(\mathcal{I}(e), [], [k_0 \mapsto \epsilon])\} \quad (1)$$

$$\cup \bigcup_{\substack{(\pi,\sigma,\Xi)\in S \\ p\in \mathrm{dom}(\pi) \\ \pi,\sigma,\Xi \rightsquigarrow_p \pi',\sigma',\Xi'}} (\pi', \sigma', \Xi') \quad (2)$$

Figure 2.28.: Transfer function for λ_α.

2.2.4. Abstract Semantics of λ_α

We naively apply the AAM design method to λ_α, in order to systematically abstract in a sound manner its semantics to enable static analysis of λ_α programs. This process is analogous to the one for λ_0, rendering the infinite components of the state space finite. In this case, infinite components are not only addresses (*Addr* and *KAddr*), but also process identifiers (*PID*) and mailboxes (*Mbox*). These changes propagate through the state space and transition relation.

Abstract State Space

The abstraction function used for addresses and process identifiers is a parameter of the analysis. We also leave the abstraction of the mailbox as a parameter of the analysis, of which we give an instantiation later in this section. Chapter 5 is dedicated to an in-depth study of potential mailbox instantiations and their effect on running time and precision. Figure 2.29 depicts the abstract state space of λ_α. The abstract process map now maps to sets of pairs of states and abstract mailboxes. This change stems from the abstract semantics having to compute a sound over-approximation with but a finite amount of process identifiers and mailboxes. Other components (store, continuation store, values) and dotted parts of the state space follow the same abstraction as for λ_0, as described in Section 2.1.3.

λ_α

$$\hat{\pi} \in \widehat{\Pi} = \widehat{PID} \to \mathcal{P}(\widehat{\Sigma} \times \widehat{Mbox})$$
$$\hat{\varsigma} \in \widehat{\Sigma} = \widehat{Control} \times \widehat{Beh} \times \widehat{KAddr}$$
$$\hat{c} \in \widehat{Control} ::= \ldots \mid \mathbf{wait}$$
$$\hat{v} \in \widehat{Val} ::= \ldots \mid \mathbf{pid}(\hat{p}) \mid \mathbf{actor}(act, \hat{\rho})$$
$$\hat{b} \in \widehat{Beh} ::= \mathbf{beh}(act, \hat{\rho}) \mid \mathbf{main}$$
$$\widehat{mb} \in \widehat{Mbox} \text{ an abstraction of concrete mailboxes}$$
$$\hat{m} \in \widehat{Message} = Tag \times \widehat{Val}$$
$$\hat{p} \in \widehat{PID} \text{ a finite set of process identifiers}$$

Figure 2.29.: Abstract state space for λ_α.

Abstract Mailboxes

There exist multiple ways of abstracting the mailboxes in actor programs, and we refer to Chapter 5 for a discussion on this matter. In this chapter, we abstract mailboxes to sets of messages, as depicted in Figure 2.30. The empty mailbox is the empty set. Enqueuing a message in a mailbox consists of performing a set union between the mailbox and a singleton set containing the enqueued message. Dequeuing a message from a mailbox returns two possible results for each message to ensure soundness, as sets do not preserve information about the multiplicity of their elements: either the message is contained more than once in the concrete mailbox abstracted by the set and it is still contained in the mailbox after dequeuing, or the message is contained only once in the concrete mailbox that the set abstracts and is removed from the resulting mailbox. The $\setminus$ operator performs a set difference operation (see Appendix A). As sets do not preserve the ordering information of their elements, any of the messages contained in the set abstracting the mailbox may be dequeued. The size of the empty mailbox is 0,

and the size of any other mailbox is approximated to ∞ as there is no information on the number of messages contained in the mailbox.

λ_α

$$\begin{aligned}
\widehat{mb} \in \widehat{Mbox} &= \mathcal{P}(\widehat{Message}) \\
\widehat{empty} &= \varnothing \\
\widehat{enq}(\hat{m}, \widehat{mb}) &= \widehat{mb} \cup \{\hat{m}\} \\
\widehat{deq}(\widehat{mb}) &= \left\{(\hat{m}, \widehat{mb}), (\hat{m}, \widehat{mb} \setminus \{\hat{m}\}) \mid \hat{m} \in \widehat{mb}\right\} \\
\widehat{size}(\varnothing) &= 0 \\
\widehat{size}(\widehat{mb}) &= \infty
\end{aligned}$$

Figure 2.30.: Abstract mailboxes for λ_α.

Abstract Communication Effects

The domain of communication effects preserves the same structure under abstraction, as depicted in Figure 2.31. Abstract domains are however used instead of concrete ones.

λ_α

$$\widehat{eff} \in \widehat{Effect} ::= \mathbf{c}(\hat{p}, \hat{\varsigma}) \mid \mathbf{snd}(\hat{p}, t, \hat{v}) \mid \mathbf{prc}(t, \hat{v}) \mid \mathbf{b}(\hat{b}, \hat{v})$$

Figure 2.31.: Abstract communication effects for λ_α.

Abstract Atomic Evaluation

Atomic evaluation is abstracted in the same way as for λ_0, and is depicted in Figure 2.32. We give here the abstracted Actor rule for atomic evaluation of actor definitions, which preserves the same structure as the concrete rule.

λ_α

$$\frac{}{\hat{\rho}, \hat{\sigma} \vdash \hat{b} \;\widehat{\Downarrow}\; \mathbf{actor}(act, \hat{\rho})}\ \textsc{Actor}$$

Figure 2.32.: Abstract atomic evaluation relation for λ_α.

Transitions

The abstract transition relation rules, acting on components of the abstract state space, are adapted to account for the changes in the state space resulting from abstraction.

These changes arise due to sound over-approximation. We highlight the differences with the concrete rules graphically and explain them.

The process map $\hat{\pi}$ now maps each process identifier to a *set* of pairs of an actor state and a mailbox. Hence the premise $\pi(p) = (\varsigma, mb)$ has to be adapted to $\hat{\pi}(\hat{p}) \ni (\hat{\varsigma}, \widehat{mb})$. This introduces non-determinism when one abstract process identifier is mapped to more than one abstract actor state. For the same reason, and because the store now maps each abstract address to a *set* of values, process map updates and store updates become join operations: $\pi[p \mapsto \ldots]$ becomes $\hat{\pi} \sqcup [\hat{p} \mapsto \{\ldots\}]$. The resulting loss in precision can be mitigated by incorporating *abstract counting* in the abstracted semantics (Might and Shivers, 2006; Might and Van Horn, 2011). Abstract counting enables performing strong updates, i.e. regular updates instead of joins, on the store and process map when an abstract address or an abstract process identifier is mapped to a single element. We do use abstract counting on the process map for improved precision in our implementation.

Sequential Transitions. The abstract sequential rule depicted SEQ in Figure 2.33 states that if an abstract sequential transition ($\widehat{\hookrightarrow}$) applies for a process $\hat{p}$, then the corresponding concurrent transition can apply. It also accounts for the changes induced by abstraction.

λ_α

$$\frac{\hat{\pi}(\hat{p}) \ni (\hat{\varsigma}, \widehat{mb}) \qquad \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \;\widehat{\hookrightarrow}\; \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \left\{(\hat{\varsigma}', \widehat{mb})\right\}], \hat{\sigma}', \widehat{\Xi}'} \text{ SEQ}$$

Figure 2.33.: Abstract transition relation rules for sequential transitions of λ_α.

Actor Management Transitions. The actor management transitions adhere to the changes following from abstraction, and are depicted in Figure 2.34. Note the fact that a created process is *joined* into the process map, together with the empty mailbox, and not just added to it. This is to ensure that, because the number of abstract process identifiers is finite, if a newly created process uses a pre-existing process identifier, it does not overwrite the local state of the already existing process.

λ_α

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni (\langle \mathbf{ev}(\texttt{(create } ae\ ae'\texttt{)}, \hat{\rho}), \hat{b}, \widehat{k}\rangle, \widehat{mb}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{actor}(act, \hat{\rho}') \\ \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v} \qquad \hat{\varsigma}' = \langle \mathbf{wait}, \mathbf{beh}(act, \hat{\rho}'[x \mapsto \hat{a}]), \widehat{k}_0\rangle \\ \hat{a} = \widehat{alloc}(x, \hat{\sigma}) \qquad \hat{p}' = \widehat{palloc}(\hat{\varsigma}', \hat{\pi}) \qquad x = \textsc{var}(act)\end{array}}{\begin{array}{c}\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{c}(\hat{p}', \hat{\varsigma}')}_{\hat{p}} \\ \hat{\pi} \sqcup [\hat{p} \mapsto \left\{(\langle \mathbf{val}(\mathbf{pid}(\hat{p}')), \hat{b}, \widehat{k}\rangle, \widehat{mb})\right\}, \hat{p}' \mapsto \left\{(\hat{\varsigma}', \widehat{empty})\right\}], \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{v}], \widehat{\Xi}\end{array}}\ \textsc{Create}$$

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni (\langle \mathbf{ev}(\texttt{(become } ae\ ae'\texttt{)}, \hat{\rho}), \hat{b}, \widehat{k}\rangle, \widehat{mb}) \\ \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{actor}(act, \hat{\rho}') \qquad \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v} \\ \hat{b}' = \mathbf{beh}(act, \hat{\rho}'[x \mapsto \hat{a}]) \qquad x = \textsc{var}(act) \qquad \hat{a} = \widehat{alloc}(x, \hat{\sigma})\end{array}}{\begin{array}{c}\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{b}(\hat{b}', \hat{v})}_{\hat{p}} \ \hat{\pi} \sqcup [\hat{p} \mapsto \left\{(\langle \mathbf{wait}, \hat{b}', \widehat{k}\rangle, \widehat{mb})\right\}], \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{v}], \widehat{\Xi}\end{array}}\ \textsc{Become}$$

Figure 2.34.: Abstract transition relation rules for actor management of λ_α.

Messages Transitions. The abstract message transition relations depicted in Figure 2.35 account for abstraction in a similar manner. Note that the $\widehat{deq}$ function returns a set of pairs of abstract messages and mailboxes, which is accounted for by the Process rule through a set membership operation.

λ_α

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni (\langle \mathbf{ev}(\texttt{(send } ae\ t\ ae'\texttt{)}, \hat{\rho}), \hat{b}, \widehat{k}\rangle, \widehat{mb}) \\ \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{pid}(\hat{p}') \qquad \hat{\pi}' = \hat{\pi} \sqcup [\hat{p} \mapsto \left\{(\langle \mathbf{val}(\hat{v}), \hat{b}, \widehat{k}\rangle, \widehat{mb})\right\}] \\ \hat{\pi}'(\hat{p}') \ni (\hat{\varsigma}, \widehat{mb}') \qquad \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v}\end{array}}{\begin{array}{c}\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{snd}(\hat{p}', t, \hat{v})}_{\hat{p}} \ \hat{\pi}' \sqcup [\hat{p}' \mapsto \left\{(\hat{\varsigma}, \widehat{enq}((t, \hat{v}), \widehat{mb}'))\right\}], \hat{\sigma}, \widehat{\Xi}\end{array}}\ \textsc{Send}$$

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni (\langle \mathbf{wait}, \mathbf{beh}(act, \hat{\rho}), \widehat{k}\rangle, \widehat{mb}) \\ ((t, \hat{v}), \widehat{mb}') \in \widehat{deq}(\widehat{mb}) \qquad (y, e) = \textsc{handler}(act, t) \qquad \hat{a} = \widehat{alloc}(y, \hat{\sigma})\end{array}}{\begin{array}{c}\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{prc}(t, \hat{v})}_{\hat{p}} \\ \hat{\pi} \sqcup [\hat{p} \mapsto \left\{(\langle \mathbf{ev}(e, \hat{\rho}[y \mapsto \hat{a}]), \mathbf{beh}(act, \hat{\rho}), \widehat{k}\rangle, \widehat{mb}')\right\}], \hat{\sigma}[\hat{a} \mapsto \hat{v}], \widehat{\Xi}\end{array}}\ \textsc{Process}$$

Figure 2.35.: Abstract transition rules for messages in λ_α.

Abstract Process Identifiers

We use a context-insensitive abstraction for process identifiers. Abstract process identifiers, as depicted in Figure 2.36, are therefore composed of only the initial behavior of the process they correspond to. Figure 2.37 represents the actor topology for Listing 2.1, as inferred by an analysis using such an abstraction for process identifiers: all the concrete `cust` actors are mapped to a single abstract actor.

The instantiation of this parameter influences the precision, but not the soundness of an analysis, as the AAM design method has been proven sound under any allocation strategy (Might and Manolios, 2009; Gilray *et al.*, 2016a). Other instantiations of abstract process identifiers are orthogonal to the issues addressed in this dissertation but are identified as potential future work and discussed in Chapter 7. Introducing a form of process-sensitivity, where the process identifier of a created process would inherit part of the history of the program's execution (e.g., the last k created processes), could lead to improved precision.

λ_α

$$\widehat{PID} = \widehat{Beh} \qquad \widehat{palloc}(\langle \_, b, \_ \rangle, \_) = b$$

Figure 2.36.: Abstract process identifiers for λ_α.

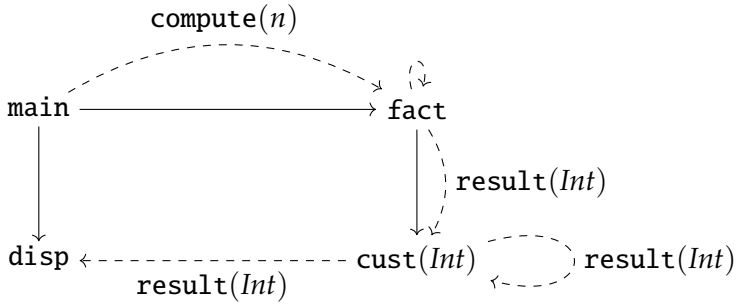


Figure 2.37.: An over-approximation of the actor topology represented in Figure 2.16. Actors denoted by `disp` and `cust` respectively denote actors with behaviors `displayer-actor` and `customer-actor`.

Abstract Injection Function

The abstract injection function, depicted in Figure 2.38 accounts for the fact that the process map now maps to pairs of which the first element has become a *set* of process states.

λ_α

$$\widehat{\mathcal{I}}(e) = [\hat{p}_0 \mapsto (\left\{ \langle \mathbf{ev}(e, []), \mathbf{main}, \widehat{k}_0 \rangle \right\}, \widehat{empty})]$$

Figure 2.38.: Abstract injection function for λ_α.

Abstract Collecting Semantics

The abstract collecting semantics now uses the abstract domains instead of the concrete ones. The fixed point of the abstract transfer function depicted in Figure 2.39, $\mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_\alpha})$, is a set containing all the program states reachable in any execution of program e, under any possible interleaving. Using this formulation of the abstract collecting semantics, a communication topology analysis (Colby, 1995; Martel and Gengler, 2000) can be derived by recording the communication effects that are generated by the transitions explored. We use $\rightsquigarrow_p$ here as notation for a transition that may generate any effect or no effect. Our implementation performs a further store widening abstraction (Van Horn and Might, 2010), using $\mathcal{P}(\widehat{\Pi}) \times \widehat{Store} \times \widehat{KStore}$ as the domain for the transfer function. The domain of the transfer function contains a powerset of process maps, rendering the height of the lattice $\mathcal{P}(\widehat{\Pi}) \times \widehat{Store} \times \widehat{KStore}$ exponential. This means that in the worst case, the analysis will iterate an exponential number of times. Therefore, the worst-case time complexity of the resulting analysis is exponential, i.e., $\mathcal{O}(2^{|Exp|})$.

λ_α

$$\widehat{\mathcal{F}}_e^{\lambda_\alpha}(S) = \left\{ (\widehat{\mathcal{I}}(e), [], [\widehat{k}_0 \mapsto \epsilon]) \right\} \tag{1}$$

$$\cup \bigcup_{\substack{(\hat{\pi}, \hat{\sigma}, \widehat{\Xi}) \in S \\ \hat{p} \in \mathrm{dom}(\hat{\pi}) \\ \hat{\pi}, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p}} \hat{\pi}', \hat{\sigma}', \widehat{\Xi}'}} (\hat{\pi}', \hat{\sigma}', \widehat{\Xi}') \tag{2}$$

Figure 2.39.: Abstract transfer function for λ_α.

2.2.5. Soundness and Termination

Theorems 3 and 4 state that the analysis described by the abstract collecting semantics terminates and is sound, two crucial properties for a static analysis.

Theorem 3 (Soundness)**.** $\mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_\alpha})$ *is a sound over-approximation of* $\mathrm{lfp}(\mathcal{F}_e^{\lambda_\alpha})$*.*

Proof. The proof is detailed in Appendix B.1.2 and performs a case analysis on the transition rules. It relies on Theorem 1 to show that the rule SEQ is sound, and shows that the concurrent transitions are sound. □

Theorem 4 (Termination)**.** *The computation of* $\mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_\alpha})$ *always terminates.*

Proof. The proof is detailed in Appendix B.1.2 and follows the same structure as the proof for Theorem 2: $\widehat{\mathcal{F}}_e^{\lambda_\alpha}$ is a monotone transfer function, $(\mathcal{P}(\widehat{\Sigma} \times \widehat{Store} \times \widehat{KStore}), \sqsubseteq)$ is a finite lattice, hence by Tarski's fixed-point theorem (Tarski, 1955), the analysis always terminates. □

2.3. Threads and Shared Memory: λ_τ

We formalize a multi-threaded extension to λ_0, inspired by threads as they are implemented in SML (Cooper and Morrisett, 1990) and OCaml (Leroy *et al.*, 2008), where mutable shared memory is represented through *mutable references* that can be protected through some form of *locking*. We present this extension in a similar manner as for λ_α: we present an overview of this extension, formalize its syntax, its concrete semantics and its abstract semantics.

2.3.1. Overview of Threads and Shared Memory

The shared-memory concurrency model targeted in this dissertation is an extension to the λ_0 language, in which we introduce support for multi-threading, support for side effects to *mutable references*, and support for delineating critical sections through *locks*.

Creating and Joining Threads

Threads can be created, or *spawned*, in λ_τ through the `spawn` construct, which creates a new thread to execute the expression given as argument. The resulting value is a process identifier that can be used to refer to the newly created thread. The `join` construct enables blocking a thread until the thread of which the process identifier is given as argument finishes its execution. The resulting value is the value returned by the joined thread.

The following snippet evaluates the expression `(+ 1 2)` in a separate thread, and waits for this computation to terminate. The final value of this snippet is therefore `3`.

```
1 (define t (spawn (+ 1 2)))
2 (join t)
```

Shared State

State can be shared among threads through shared mutable references. References can be created using the `ref` construct, which boxes the value of its given expression inside a reference. References can be read using the `deref` construct which returns the content of the reference, and written to using the `ref-set!` construct, which updates the content of the reference to a new value. This is illustrated by the following snippet. At line 1, a reference `x` is created and holds the value 0. The assertion at line 2 shows that accessing the contents of the reference yields the value 0. Line 3 increments the contents of reference `x`. The assertion at line 4 shows that the updated reference holds the value 1.

```
1 (define x (ref 0))
2 (assert (= 0 (deref x)))
3 (ref-set! x (+ (deref x) 1))
4 (assert (= 1 (deref x)))
```

The choice of using references instead of introducing variables that are mutable through the `set!` construct of Scheme is to ensure that programs written in λ_τ can profit from shared mutable state that can be passed along function calls, thereby modeling a call-by-reference mechanism. This is not possible with `set!`, which cannot mutate variables outside of their scope. Doing so in Scheme requires for example to box mutable values in a cons cell, and to use `set-car!`. References introduced with `ref` avoid this unpleasant necessity. Moreover, this has the advantage of syntactically denoting mutable state, thereby improving program comprehension: variables are always immutable, and references are always mutable.

Note that the names may remind of *refs* in Clojure, but this is a different concept. References in our language are simply mutable boxes, while in Clojure they are part of an implementation of software transactional memory (Hickey, 2008).

Locks

Locks are used to protect critical sections of a program from race conditions. A lock can be created through the `new-lock` construct, which takes no argument. A lock is acquired by a thread through the `acquire` construct unless the lock is currently held by another thread. If a thread uses `acquire` to acquire a lock, and the lock is already held by another thread, the first thread will block until the lock is released by the second one and becomes available. A lock can be released through the `release` construct by the thread that acquired it.

Preventing Race Conditions with Locks

Consider the situation represented in Figure 2.40, which represents two possible interleavings of a program in which two threads perform the `(ref-set! x (+ (deref x) 1))` operation concurrently, with the goal of incrementing the value contained in reference `x` twice. Suppose the initial value of `x` is 0. If thread t_1 reads the contents of reference `x` first, and thread t_2 then reads the contents of reference `x` (left-hand side of the depicted situation), both threads see the same value of the reference, and the update to the reference from within one thread will overwrite the update from within the other thread. On the other hand, and this is the desired behavior, if thread t_1 reads from and writes to the reference before thread t_2 reads the contents of the reference (right-hand side), the contents of `x` is incremented twice.

Locks make it possible to avoid race conditions by delineating a critical section in the code. The idea is to acquire a lock before entering the critical section and releasing it afterwards. For example, no more than one thread can execute line 5 of the following snippet at any given time. Replacing uses of `(ref-set! x (+ (deref x) 1))` by calls to `inc` in the previous example eliminates the race condition.

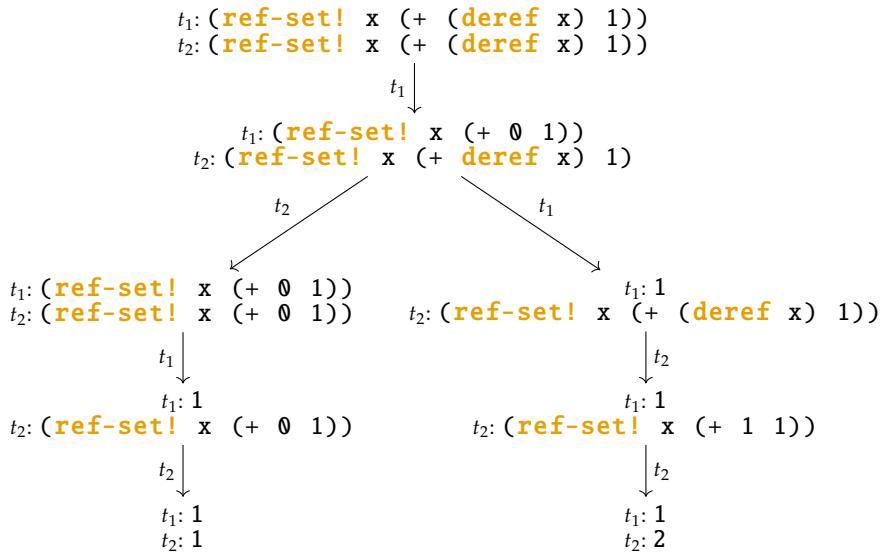


Figure 2.40.: Race condition caused by two concurrent reads and writes to a mutable reference shared between two threads. Nodes denote program states, showing the expression evaluated by the threads ($t_i : e$), or the value reached by the threads ($t_i : v$). Edges denote threads transitioning, and are annotated with the identifier of the thread that transitions.

```
1 (define lock (new-lock))
2 (define x (ref 0))
3 (define (inc)
4   (acquire lock)
5   (ref-set! x (+ (deref x) 1))
6   (release lock))
```

Example Multi-Threaded Program

We give a more involved example multi-threaded program for the computation of a factorial computation in Listing 2.2. The number of threads created in this program depends on the input given by the user. Consider a concrete execution when 3 is given as input.

- The main thread creates a thread to compute `(fact-thread 1 3)` and joins this new thread (line 10).
- The thread executing `(fact-thread 1 3)` creates two threads: one executing `(fact-thread 1 1)` and one executing `(fact-thread 2 3)`, and joins these created threads to multiply their results (lines 6 to 8).
- The thread executing `(fact-thread 1 1)` returns 1 (line 3).
- The thread executing `(fact-thread 2 3)` returns 6 (line 4).

- The blocking join operation can now return the value of the threads, and `(fact-thread 1 3)` returns 6, which is printed to the screen.

The topology resulting from this concrete run of the program is represented in Figure 2.41.

```
1 (define (fact-thread from to)
2   (case (- to from)
3     ((0)  from)
4     ((1)  (* from to))
5     (else (let ((middle (+ from (/ (- to from) 2)))
6                 (t1 (spawn (fact-thread from (- middle 1))))
7                 (t2 (spawn (fact-thread middle to))))
8             (* (join t1) (join t2))))))
9 (define (fact n)
10   (printf "fact(~a) = ~a~n" n (join (spawn (fact-thread 1 n)))))
11 (fact (read-integer))
```

Listing 2.2: Concurrent computation of the factorial of a number resulting from user input.

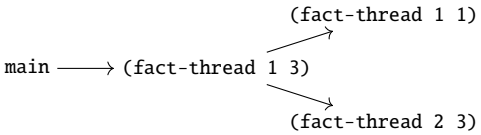


Figure 2.41.: Representation of a concrete execution of Listing 2.2 with 3 as input.

2.3.2. Syntax of λ_τ

Figure 2.42 depicts the syntax of λ_τ. The semantics of λ_τ extend λ_0 with support for the following shared-memory multi-threading concepts.

1. **Thread creation and joining**. Threads can be created to compute an expression in a different process through the `spawn` construct, and a thread can *join* another thread to obtain the final value of the computation performed by this other thread through the `join` construct. The latter is a blocking operation.

2. **Mutable references**. We introduce mutable references into the language. References can be created through the `ref` construct, which wraps its argument value into a reference. References hold a value that can be read through the `deref` construct or that can be modified through the `ref-set!` construct.

3. **Locks**. We introduce locks to delineate critical sections in multi-threaded programs so that race conditions can be avoided. Locks can be created through the `new-lock` construct, returning a new lock. A lock can be acquired by a thread through the `acquire` construct, and released through the `release` construct. Only the thread that acquired a lock can release it. If a thread tries to acquire a lock already held by another thread, the requesting thread blocks until the lock is released.

λ_τ

$$
\begin{aligned}
e \in \mathit{Exp} ::= \ & \ldots \\
& \mid \texttt{(spawn } e\texttt{)} \mid \texttt{(join } ae\texttt{)} \\
& \mid \texttt{(ref } ae\texttt{)} \mid \texttt{(deref } ae\texttt{)} \mid \texttt{(ref-set! } ae\ ae\texttt{)} \\
& \mid \texttt{(new-lock)} \mid \texttt{(acquire } ae\texttt{)} \mid \texttt{(release } ae\texttt{)}
\end{aligned}
$$

Figure 2.42.: Syntax of λ_τ.

2.3.3. Concrete Semantics of λ_τ

We provide the concrete semantics of λ_τ as an extension to the concrete semantics of λ_0.

State Space

The state space for λ_τ is depicted in Figure 2.43. States of λ_τ processes (ς) are identical to states for λ_0 programs: a pair of a control component and a continuation address. Process maps (π) map process identifiers to process states. For thread management, process identifiers are first-class values (**pid**) returned by `spawn`, and accepted by `join`. References and locks are also first-class values. A reference $\mathbf{ref}(a)$ wraps an address a at which the value of the reference resides in the store. A lock $\mathbf{lock}(a)$ also wraps an address. The value pointed to by the address is a lock value, which can either correspond to an acquired lock, containing the process identifier of the process that holds the lock ($\mathbf{locked}(p)$), or to a released lock (**unlocked**). These lock values (**locked** and **unlocked**) are not first-class values as they are not returned by any construct, but are used internally when accessing locks.

λ_τ

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathit{Control} \times \mathit{KAddr} \\
\pi \in \Pi &= \mathit{PID} \to \Sigma \\
v \in \mathit{Val} ::= & \ldots \\
& \mid \mathbf{pid}(p) \mid \mathbf{ref}(a) \\
& \mid \mathbf{lock}(a) \mid \mathbf{locked}(p) \mid \mathbf{unlocked} \\
p \in \mathit{PID} & \text{ a set of process identifiers}
\end{aligned}
$$

Figure 2.43.: State space for λ_τ.

Communication Effects

Multi-threaded λ_τ programs can generate three kinds of communication effects, depicted in Figure 2.44: process-related effects, memory access effects, and lock access effects. Process-related effects comprise the creation of a process ($\mathbf{c}(p, \varsigma)$) with process

identifier p and initial state ς, and the join on a process p which has terminated its execution with value v ($\mathbf{j}(p, v)$). Memory access effects are either read accesses ($\mathbf{r}(a)$), or write accesses ($\mathbf{w}(a)$), and act on a specific address a. Similarly, lock access effects are either acquire accesses ($\mathbf{acq}(p, a)$) or release accesses ($\mathbf{rel}(p, a)$), act on a specific lock address a, and include the process identifier p of the thread performing the access.

λ_τ

$$\begin{aligned} \mathit{eff} \in \mathit{Effect} ::= \; & \mathbf{c}(p, \varsigma) \;\mid\; \mathbf{j}(p, v) \\ \mid \; & \mathbf{r}(a) \;\mid\; \mathbf{w}(a) \\ \mid \; & \mathbf{acq}(p, a) \;\mid\; \mathbf{rel}(p, a) \end{aligned}$$

Figure 2.44.: Communication effects for λ_τ.

Transition Relation

The semantics of λ_τ programs is defined by the concurrent transition relation ($\rightsquigarrow_p$), annotated with the process identifier p of the thread performing the transition. This transition relation acts on process maps, value stores and continuation stores, and a transition is denoted as $\pi, \sigma, \Xi \rightsquigarrow_p \pi', \sigma', \Xi'$. The transition may carry a communication effect, written $\overset{\mathit{eff}}{\rightsquigarrow}_p$.

Sequential transitions. Figure 2.45 depicts the transition relation rule for sequential transitions for λ_τ. Just as for λ_α, λ_τ relies on the sequential transition of λ_0 to encode the semantics of the sequential subset of the language through the Seq rule. No effects are generated for this transition rule.

λ_τ

$$\frac{\pi(p) = \varsigma \qquad \varsigma, \sigma, \Xi \hookrightarrow \varsigma', \sigma', \Xi'}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \varsigma'], \sigma', \Xi'} \textsc{Seq}$$

Figure 2.45.: Transition relation rule for sequential transitions for λ_τ.

Thread management. Figure 2.46 depicts the rules for thread management transitions for λ_τ. For the `spawn` construct, the Spawn transition rule creates a new process state ς' that is added to the process map, generates a $\mathbf{c}(p, \varsigma)$ communication effect where p is the process identifier of the thread created, and ς is the initial state of the created thread. The thread performing the spawn operation proceeds to a value state with the process identifier of the created thread as value. When a thread joins another thread p', it can only advance its execution when thread p' has reached the end of its own execution, i.e., when p' has reached a state with a value for its control component and k_0 for its

continuation address. If this is the case, the thread performing the join can proceed and reaches a value state wrapping the value of the terminated thread (rule Join).

λ_τ

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(spawn } e\texttt{)}, \rho), k\rangle \qquad \varsigma' = \langle \mathbf{ev}(e, \rho), k_0\rangle \qquad p' = \mathit{palloc}(\varsigma', \pi)}{\pi, \sigma, \Xi \overset{\mathbf{c}(p', \varsigma')}{\rightsquigarrow}_p \pi[p \mapsto \langle \mathbf{val}(\mathbf{pid}(p')), k\rangle, p' \mapsto \varsigma'], \sigma, \Xi} \textsc{Spawn}$$

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(join } ae\texttt{)}, \rho), k\rangle \qquad \rho, \sigma \vdash ae \Downarrow \mathbf{pid}(p') \qquad \pi(p') = \langle \mathbf{val}(v), k_0\rangle}{\pi, \sigma, \Xi \overset{\mathbf{j}(p', v)}{\rightsquigarrow}_p \pi[p \mapsto \langle \mathbf{val}(v), k\rangle], \sigma, \Xi} \textsc{Join}$$

Figure 2.46.: Transition relation rules for thread management for λ_τ.

References. Figure 2.47 depicts the rules for references for λ_τ. The transition rule for the `ref` construct allocates a new address using the *alloc* allocation function and returns a reference bound to that address (rule Ref). The transition rule for the `deref` construct looks up the value in the store at the address a wrapped by the reference, and generates a communication effect $\mathbf{r}(a)$ (rule Deref). The transition rule for the `ref-set!` construct updates the value in the store residing at address a wrapped by the reference, and generates a communication effect $\mathbf{w}(a)$ (rule RefSet).

λ_τ

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(ref } ae\texttt{)}, \rho), k\rangle \qquad a = \mathit{alloc}(ae, \sigma) \qquad \rho, \sigma \vdash ae \Downarrow v}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\mathbf{ref}(a)), k\rangle], \sigma[a \mapsto v], \Xi} \textsc{Ref}$$

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(deref } ae\texttt{)}, \rho), k\rangle \qquad \rho, \sigma \vdash ae \Downarrow \mathbf{ref}(a) \qquad v = \sigma(a)}{\pi, \sigma, \Xi \overset{\mathbf{r}(a)}{\rightsquigarrow}_p \pi[p \mapsto \langle \mathbf{val}(v), k\rangle], \sigma, \Xi} \textsc{Deref}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(ref-set! } ae\ ae'\texttt{)}, \rho), k\rangle \\ \rho, \sigma \vdash ae \Downarrow \mathbf{ref}(a) \qquad \rho, \sigma \vdash ae' \Downarrow v\end{array}}{\pi, \sigma, \Xi \overset{\mathbf{w}(a)}{\rightsquigarrow}_p \pi[p \mapsto \langle \mathbf{val}(v), k\rangle], \sigma[a \mapsto v], \Xi} \textsc{RefSet}$$

Figure 2.47.: Transition relation rules for references for λ_τ.

Locks. Figure 2.48 depicts the rules for locks for λ_τ. Rule NewLock creates a new lock by allocating a new address in the store and associating it to the **unlocked** value. Rule Acquire sets an unlocked lock to a **locked**(p) value where p is the process identifier of the thread acquiring the lock. This operation generates an **acq**(p, a) communication

effect. Rule RELEASE changes the value of a lock to **unlocked**, provided the lock was acquired by the same process p, and generates a $\mathbf{rel}(p, a)$ communication effect.

λ_τ

$$\frac{\pi(p) = \langle \mathbf{ev}(\texttt{(new-lock)}, \rho), k\rangle \qquad a = alloc(\texttt{(new-lock)}, \sigma)}{\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \langle \mathbf{val}(\mathbf{lock}(a)), k\rangle], \sigma[a \mapsto \mathbf{unlocked}], \Xi}\ \textsc{NewLock}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(acquire } ae\texttt{)}, \rho), k\rangle \\ \rho, \sigma \vdash ae \Downarrow \mathbf{lock}(a) \qquad \sigma(a) = \mathbf{unlocked}\end{array}}{\pi, k, \sigma, \Xi \overset{\mathbf{acq}(p,a)}{\rightsquigarrow}_p \pi[p \mapsto \langle \mathbf{val}(\mathbf{lock}(a)), k\rangle], \sigma[a \mapsto \mathbf{locked}(p)], \Xi}\ \textsc{Acquire}$$

$$\frac{\begin{array}{c}\pi(p) = \langle \mathbf{ev}(\texttt{(release } ae\texttt{)}, \rho), k\rangle \\ \rho, \sigma \vdash ae \Downarrow \mathbf{lock}(a) \qquad \sigma(a) = \mathbf{locked}(p)\end{array}}{\pi, \sigma, \Xi \overset{\mathbf{rel}(p,a)}{\rightsquigarrow}_p \pi[p \mapsto \langle \mathbf{val}(\mathbf{lock}(a)), k\rangle], \sigma[a \mapsto \mathbf{unlocked}], \Xi}\ \textsc{Release}$$

Figure 2.48.: Transition relation rules for locks for λ_τ.

Injection Function

The injection function for λ_τ programs, depicted in Figure 2.49, injects the program to evaluate in a process map containing an initial process with process identifier p_0, that evaluates the expression e corresponding to the program.

λ_τ

$$\mathcal{I}(e) = [p_0 \mapsto \langle \mathbf{ev}(e, []), k_0\rangle]$$

Figure 2.49.: Injection function for λ_τ.

Process Identifiers

Process identifiers, depicted in Figure 2.50, use the same concrete definitions as for λ_α programs. They are represented by integers, and the next available process identifier is returned for the concrete semantics. Note that the size of the process map only increases during the program execution, as terminated process are not removed from the process map.

λ_τ

$$p \in PID = \mathbb{N} \qquad palloc(\varsigma, \pi) = |\pi|$$

Figure 2.50.: Process identifiers for λ_τ.

Collecting Semantics

The collecting semantics is defined similarly to the collecting semantics for λ_0 (Figure 2.8) and λ_α (Figure 2.28), as the fixed point of a transfer function $\mathcal{F}_e^{\lambda_\tau} : \mathcal{P}(\Pi \times \mathit{Store} \times \mathit{KStore}) \to \mathcal{P}(\Pi \times \mathit{Store} \times \mathit{KStore})$, depicted in Figure 2.51. This transfer function acts on tuples consisting of a process map, a value store and a continuation store. The initial injected state is reachable (1), and any state that can be reached in one step on any process from a reachable state is itself reachable (2). We use $\rightsquigarrow_p$ here as notation for a transition that may generate any effect or no effect.

λ_τ

$$\mathcal{F}_e^{\lambda_\tau}(S) = \{(\mathcal{I}(e), [], [k_0 \mapsto \epsilon])\} \quad (1)$$

$$\cup \bigcup_{\substack{(\pi,\sigma,\Xi)\in S \\ p\in\mathrm{dom}(\pi) \\ \pi,\sigma,\Xi \rightsquigarrow_p \pi',\sigma',\Xi'}} (\pi', \sigma', \Xi') \quad (2)$$

Figure 2.51.: Transfer function for λ_τ.

2.3.4. Abstract Semantics of λ_τ

We apply the AAM design method to the concrete semantics of λ_τ, as we did for λ_α. Process identifiers are rendered finite, as are value addresses and continuation addresses. These changes propagate through the abstract state space and transition relation.

Abstract State Space

Figure 2.52 depicts the abstract state space for λ_τ. The abstraction for the state space renders the sets of abstract process identifiers and abstract addresses finite. The range of abstract process maps $\hat{\pi}$ becomes a powerset of states to account for the possibility of multiple process states being mapped to the same process identifier under abstraction. With the exception of the stores which are abstracted similarly to the abstraction in λ_0 (see Section 2.1.3), the remainder of the state space remains structurally the same.

λ_τ

$$\hat{\pi} \in \widehat{\Pi} = \widehat{PID} \to \mathcal{P}(\widehat{\Sigma})$$
$$\hat{v} \in \widehat{Val} ::= \ldots \mid \mathbf{pid}(\hat{p}) \mid \mathbf{ref}(\hat{a}) \mid \mathbf{lock}(\hat{a}) \mid \mathbf{locked}(\hat{p}) \mid \mathbf{unlocked}$$
$$\hat{p} \in \widehat{PID} \text{ a finite set of process identifiers}$$

Figure 2.52.: Abstract state space for λ_τ.

Abstract Communication Effects

Figure 2.53 depicts the abstract communication effects for λ_τ. The communication effects preserve their structure under abstraction, but now contain abstract elements instead of concrete ones.

λ_τ

$$\widehat{eff} \in \widehat{Effect} ::= \mathbf{c}(\hat{p}, \hat{\varsigma}) \mid \mathbf{j}(\hat{p}, \hat{v}) \mid \mathbf{r}(\hat{a}) \mid \mathbf{w}(\hat{a}) \mid \mathbf{acq}(\hat{p}, \hat{a}) \mid \mathbf{rel}(\hat{p}, \hat{a})$$

Figure 2.53.: Abstract communication effects for λ_τ.

Abstract Atomic Evaluation

The atomic evaluation relation is abstracted just as for the abstract interpretation of λ_0. As the rules are identical to the ones depicted in Figure 2.11, we do not repeat them here.

Abstract Transition Relation

Adapting the transition relation rules to the abstract state space requires changing how the process map is accessed and updated. Abstract process maps now map to sets of process states. Accessing the state of a thread with process identifier p is therefore changed from $\varsigma = \pi(p)$ to $\hat{\varsigma} \in \hat{\pi}(\hat{p})$. This is similar to accesses to the process map in the semantics of λ_α (Figures 2.33 to 2.35). For integrating new information into the process map, a join operation ($\sqcup$) is used rather than an in-place update. This to ensure soundness. Joins are also used on value stores and continuation stores for the same reason.

Sequential transitions. The abstract sequential transition, depicted in Figure 2.54, follows the changes made to the abstract state space. Note the use of a membership

operation when extracting a process state from the process map, as more than one process state may be associated with a single abstract process identifier. The use of the join operation on the process map to update a process state is of note too.

λ_τ

$$\frac{\hat{\pi}(\hat{p}) \ni \hat{\varsigma} \qquad \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \hookrightarrow \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \{\hat{\varsigma}'\}], \hat{\sigma}', \widehat{\Xi}'} \textsc{Seq}$$

Figure 2.54.: Abstract rules for sequential transitions for λ_τ.

Thread management. The rules for thread management, depicted in Figure 2.55, similarly follow the changes to the state space induced by abstraction.

λ_τ

$$\frac{\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(spawn } e\texttt{)}, \hat{\rho}), \widehat{k} \rangle \qquad \hat{\varsigma}' = \langle \mathbf{ev}(e, \hat{\rho}), \widehat{k}_0 \rangle \qquad \hat{p}' = \widehat{palloc}(\hat{\varsigma}', \hat{\pi})}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{c}(\hat{p}', \hat{\varsigma}')}{\rightsquigarrow}_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \left\{ \langle \mathbf{val}(\mathbf{pid}(\hat{p}')), \widehat{k} \rangle \right\}, \hat{p}' \mapsto \{\hat{\varsigma}'\}], \hat{\sigma}, \widehat{\Xi}} \textsc{Spawn}$$

$$\frac{\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(join } ae\texttt{)}, \hat{\rho}), \widehat{k} \rangle \qquad \hat{\rho}, \hat{\sigma} \vdash ae \Downarrow \mathbf{pid}(\hat{p}') \qquad \hat{\pi}(\hat{p}') \ni \langle \mathbf{val}(\hat{v}), \widehat{k}_0 \rangle}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{j}(\hat{p}, \hat{v})}{\rightsquigarrow}_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \left\{ \langle \mathbf{val}(\hat{v}), \widehat{k} \rangle \right\}], \hat{\sigma}, \widehat{\Xi}} \textsc{Join}$$

Figure 2.55.: Abstract rules for thread management transitions for λ_τ.

References. Rules for references, depicted in Figure 2.56, similarly follow the changes to the state space induced by abstraction.

λ_τ

$$\frac{\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(ref } ae\texttt{)}, \hat{\rho}), \widehat{k}\rangle \qquad \hat{a} = \widehat{alloc}(ae, \hat{\sigma}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \hat{v}}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \left\{\langle \mathbf{val}(\mathbf{ref}(\hat{a})), \widehat{k}\rangle\right\}], \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{v}], \widehat{\Xi}} \textsc{Ref}$$

$$\frac{\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(deref } ae\texttt{)}, \hat{\rho}), \widehat{k}\rangle \qquad \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{ref}(\hat{a}) \qquad \hat{v} \in \hat{\sigma}(\hat{a})}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{r}(\hat{a})}{\rightsquigarrow}_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \langle \mathbf{val}(\hat{v}), \widehat{k}\rangle], \hat{\sigma}, \widehat{\Xi}} \textsc{Deref}$$

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(ref-set! } ae\ ae'\texttt{)}, \hat{\rho}), \widehat{k}\rangle \\ \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{ref}(\hat{a}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v}\end{array}}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{w}(\hat{a})}{\rightsquigarrow}_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \langle \mathbf{val}(\hat{v}), \widehat{k}\rangle], \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{v}], \widehat{\Xi}} \textsc{RefSet}$$

Figure 2.56.: Abstract rules for references transitions for λ_τ.

Locks. Rules for locks, depicted in Figure 2.57, similarly follow the changes to the state space induced by abstraction.

λ_τ

$$\frac{\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(new-lock)}, \hat{\rho}), \widehat{k}\rangle \qquad \hat{a} = \widehat{alloc}(\texttt{(new-lock)}, \hat{\sigma})}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \langle \mathbf{val}(\mathbf{lock}(\hat{a})), \widehat{k}\rangle], \hat{\sigma} \sqcup [\hat{a} \mapsto \mathbf{unlocked}], \widehat{\Xi}} \textsc{NewLock}$$

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(acquire } ae\texttt{)}, \hat{\rho}), \widehat{k}\rangle \\ \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{lock}(\hat{a}) \qquad \hat{\sigma}(\hat{a}) \ni \mathbf{unlocked}\end{array}}{\hat{\pi}, \widehat{k}, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{acq}(\hat{p}, \hat{a})}{\rightsquigarrow}_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \langle \mathbf{val}(\mathbf{lock}(\hat{a})), \widehat{k}\rangle], \hat{\sigma} \sqcup [\hat{a} \mapsto \mathbf{locked}(\hat{p})], \widehat{\Xi}} \textsc{Acquire}$$

$$\frac{\begin{array}{c}\hat{\pi}(\hat{p}) \ni \langle \mathbf{ev}(\texttt{(release } ae\texttt{)}, \hat{\rho}), \widehat{k}\rangle \\ \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{lock}(\hat{a}) \qquad \hat{\sigma}(\hat{a}) \ni \mathbf{locked}(\hat{p})\end{array}}{\hat{\pi}, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{rel}(\hat{p}, \hat{a})}{\rightsquigarrow}_{\hat{p}} \hat{\pi} \sqcup [\hat{p} \mapsto \langle \mathbf{val}(\mathbf{lock}(\hat{a})), \widehat{k}\rangle], \hat{\sigma} \sqcup [\hat{a} \mapsto \mathbf{unlocked}], \widehat{\Xi}} \textsc{Release}$$

Figure 2.57.: Abstract rules for locks transitions for λ_τ.

Abstract Process Identifiers

As for the abstract semantics of λ_α programs, we use a context-insensitive abstraction for process identifiers. Figure 2.58 depicts abstract process identifiers, which are abstracted to the control component of the initial process state. Allocating a process identifier therefore amounts to extracting the control component from the given process state.

Figure 2.59 represents the abstract topology inferred with such an allocation strategy for the factorial computation of Listing 2.2. Exploring more precise process identifier allocation strategies is an interesting avenue identified as future work, and discussed in Chapter 7. A context-sensitive allocation strategy could distinguish between threads evaluating the same expression but created at different points in the execution of a multi-threaded program.

λ_τ

$$p \in PID = Control \qquad palloc(\langle c, \_ \rangle, \pi) = c$$

Figure 2.58.: Abstract process identifiers for λ_τ.

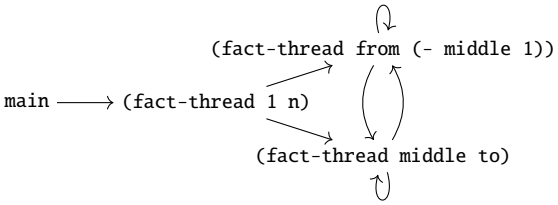


Figure 2.59.: Abstraction of the concrete executions of Listing 2.1 where process identifiers are abstracted as the expression executed within the thread.

Abstract Injection Function

The abstract injection function, depicted in Figure 2.60 accounts for the fact that the range of the abstract process map is now a powerset of process states.

λ_τ

$$\widehat{\mathcal{I}}(e) = [\hat{p}_0 \mapsto \left\{ \langle \mathbf{ev}(e, []) \rangle, \widehat{k}_0 \right\}]$$

Figure 2.60.: Abstract injection function for λ_τ.

Abstract Collecting Semantics

The abstract collecting semantics is derived from the concrete collecting semantics by substituting abstract for concrete components, and is the result of computing the fixed point of the transfer function $\widehat{\mathcal{F}}_e^{\lambda_\tau} : \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}) \to \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore})$, depicted in Figure 2.61. The abstract collecting semantics provides a sound over-approximation of the set of all reachable states of a multi-threaded program e, under all possible thread interleavings. We use $\rightsquigarrow_{\hat{p}}$ here as notation for transitions that may generate any effect or no effect. Our implementation performs a further store widening abstraction (Van

Horn and Might, 2010), using $\mathcal{P}(\widehat{\Pi}) \times \widehat{Store} \times \widehat{KStore}$ as domain for the transfer function. As for λ_α, the height of the lattice $\mathcal{P}(\widehat{\Pi}) \times \widehat{Store} \times \widehat{KStore}$ is exponential, and the analysis therefore exhibits an exponential worst-case time complexity, i.e., $\mathcal{O}(2^{|Exp|})$.

λ_τ

$$\widehat{\mathcal{F}}_e^{\lambda_\tau}(S) = \left\{ \left(\left\{ \widehat{\mathcal{I}}(e) \right\}, [], [\widehat{k}_0 \mapsto \epsilon] \right) \right\} \tag{1}$$

$$\cup \bigcup_{\substack{(\hat{\pi},\hat{\sigma},\widehat{\Xi}) \in S \\ \hat{p} \in \mathrm{dom}(\hat{\pi}) \\ \hat{\pi},\hat{\sigma},\widehat{\Xi} \rightsquigarrow_{\hat{p}} \hat{\pi}',\hat{\sigma}',\widehat{\Xi}'}} (\hat{\pi}', \hat{\sigma}', \widehat{\Xi}') \tag{2}$$

Figure 2.61.: Abstract transfer function for λ_τ.

2.3.5. Soundness and Termination

Theorems 5 and 6 state that the analysis described by the abstract collecting semantics terminates and is sound, two crucial properties for a static analysis.

Theorem 5 (Soundness)**.** $\mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_\tau})$ *is a sound over-approximation of* $\mathrm{lfp}(\mathcal{F}_e^{\lambda_\tau})$.

Proof. The proof is detailed in Appendix B.1.3 and performs a case analysis on the transition rules. It relies on Theorem 1 to show that the rule Seq is sound, and shows that the concurrent transitions are sound. □

Theorem 6 (Termination)**.** *The computation of* $\mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_\tau})$ *always terminates.*

Proof. The proof is detailed in Appendix B.1.3 and follows the same structure as the proof for Theorem 2: $\widehat{\mathcal{F}}_e^{\lambda_\tau}$ is a monotone transfer function, $(\mathcal{P}(\widehat{\Sigma} \times \widehat{Store} \times \widehat{KStore}), \sqsubseteq)$ is a finite lattice, hence by Tarski's fixed-point theorem (Tarski, 1955), the analysis always terminates. □

2.4. Soundness Testing and Evaluation of Running Time, Precision, and Scalability

We implemented the analyses resulting from a naive application of AAM described in this chapter. To empirically evaluate these analyses, we composed a list of 56 benchmark programs consisting of 28 λ_α programs and 28 λ_τ programs. We evaluate the running time of our implementation of the analyses presented in this chapter on our set of benchmark programs.

2.4.1. Implementation

All of the analyses presented in this dissertation have been implemented on top of the Scala-AM static analysis framework (Stiévenart *et al.*, 2016b; Stiévenart *et al.*, 2016a) and are available online along with the benchmark programs used to evaluate them[1]. The base sequential language supported by our implementation is not the λ-calculus variant λ_0 in ANF, but rather a large subset of Scheme in direct style with support for data types such as integers, strings, cons-cells, vectors, and with a number of primitive operations on these data types (e.g., `string-length`, `cons`, `car`, `cdr`, `make-vector`, etc.). Moreover, the actor constructs introduced in λ_α are not restricted to messages with one argument nor to actors with a single state variable, but support any number of message arguments and actor state variables. Constructs for λ_τ match the constructs described in this chapter, but are in direct style rather than in ANF.

Each running time reported in this dissertation is an average of 20 analysis runs after an initial 10 warmup runs. The warmup runs eliminate the initialization overhead of the JVM and its JIT compiler, as our static analysis framework is implemented in Scala and executed on the JVM. We used Scala 2.12.3 and Java 1.8.0_151 on a Linux server with an Intel Xeon CPU processor running at 3.5GHz and 256GB of RAM, 8GB of which were allocated for the heap of the JVM, and 200MB of which were allocated for its stack. We used the same setup for all empirical evaluations in this dissertation.

We also implemented concrete interpreters for concrete versions of λ_α and λ_τ, which record the communication effects that arise during the execution of a program. This enables comparing the result of an analysis that infers the communication effects with the run-time communication effects that have been recorded, with the aim of demonstrating the soundness of our implementation, and of measuring its precision empirically.

2.4.2. Benchmark Suite

To assess the precision and performance of our implementation, we used two benchmark suites of 28 benchmark programs each, one for actor-based programs and one for multi-threaded programs. The full list of benchmark programs is given in Table 2.1, with their length in lines of code (LOCs), the number of actor creation sites for actor benchmarks, i.e., the number of expressions which are calls to `create` (C), and the number of thread spawning sites, i.e., the number of expressions which are calls to `spawn` (S). Note that at run time, each process creation site may be executed an arbitrary number of times. Most of the benchmarks actually create a dynamic number of processes depending on a parameter, not known in advance. These benchmarks therefore expose dynamic process creation where the number of process created is not known in advance. A static analysis has to account for all possible values of that parameter. We use these benchmarks

[1] `https://github.com/acieroid/scala-am` on the branch `modularactors` for analyses of actors, and on the branch `modularthreads` for analyses of shared-memory multi-threading. Benchmark programs for actors reside in the `actor/savina` directory, and benchmark programs for shared-memory multi-threading reside in the `threads/suite` directory. Benchmark programs used to evaluate the precision in Chapter 5 are reside in the `actors/soter` directory.

throughout the dissertation to evaluate the analyses presented.

Actors						Threads					
Bench.	LOCs	C	Bench.	LOCs	C	Bench.	LOCs	S	Bench.	LOCs	S
PP	27	2	BTX	61	2	ABP	81	2	TRAPR	78	1
COUNT	29	2	RSORT	60	3	COUNT	55	2	ATOMS	71	1
FJT	38	1	FBANK	143	13	DEKKER	56	2	STM	175	1
FJC	17	1	SIEVE	37	3	FACT	73	4	NBODY	174	1
THR	43	1	UCT	145	4	MATMUL	123	4	SIEVE	71	1
CHAM	81	2	OFL	293	3	MCARLO	38	2	CRYPT	224	1
BIG	52	2	TRAPR	72	2	MSORT	49	2	MCEVAL	145	1
CDICT	67	3	PIPREC	74	2	PC	51	2	QSORT	85	2
CSLL	61	3	RMM	113	3	PHIL	52	1	TSP	153	3
PCBB	98	3	QSORT	69	3	PHILD	66	1	BCHAIN	116	1
PHIL	58	3	APSP	188	1	PP	52	1	LIFE	172	1
SBAR	77	4	SOR	201	3	RINGBUF	90	2	PPS	108	2
CIG	49	2	ASTAR	92	2	RNG	30	1	MINIMAX	133	1
LOGM	106	3	NQN	106	2	SUDOKU	96	29	ACTORS	127	1

Table 2.1.: Benchmark programs used in the evaluation of the analyses.

Actor Benchmarks

The Savina benchmark suite (Imam and Sarkar, 2014) is a set of 28 benchmark programs written in Scala, designed to empirically evaluate different implementations of the actor model. The original Savina programs range from 102 to 616 lines of Scala code, with fragments written in Java. To evaluate our analyses for λ_α, we translated all of the Savina benchmarks to λ_α. After translation, the benchmarks range from 17 to 293 lines of code. This difference in code size can be attributed to the conciseness of λ_α compared to Scala and Java in general, and to λ_α's more concise actor definitions which do not involve defining abstract methods inherited from a trait.

The resulting benchmark suite includes a number of programs intended to benchmark specific aspects of actor implementations such as throughput (FJT) or actor creation time (FJC). There are also λ_α implementations of well-known concurrency problems such as the dining philosophers (PHIL), the sleeping barbers (SBAR), and the cigarette smokers (CIG). Also included are implementations of concurrent sorting algorithms such as bitonic sort (BTX), radix sort (RSORT), and quicksort (QSORT), as well as concurrent versions of trapezoidal approximation (TRAPR), all-pair shortest path (APSP), the A^* algorithm (ASTAR), and the n queens problem (NQN).

The majority of benchmarks feature dynamic process creation, creating a number of processes determined by integer parameters specific to each benchmark. For concrete runs of the benchmark programs, these parameters are chosen randomly at every program run. This is because these parameters influence the behavior of the programs, and we use the concrete program runs to measure the precision of our analyses, hence it is desirable to explore as many different concrete runs as possible. For the analysis of the benchmark programs, the parameters are approximated by the top integer value. This

sound approximation ensures that any concrete run, with any possible value for the parameter, is accounted for by the analysis.

Thread Benchmarks

In order to evaluate the analyses for λ_τ, we composed a set of 28 multi-threaded programs. These benchmarks exhibit dynamic process creation as well as the use of higher-order features. The benchmark suite includes common multi-threaded algorithms such as the alternating bit protocol (ABP), the Dekker algorithm (DEKKER), the producer-consumer problem (PC), and two implementations of the dining philosophers (PHIL, PHILD). It also includes a number of multi-threaded implementations of common computer science problems, such as matrix multiplication (MATMUL), factorial computation (FACT), merge sort (MSORT), a Sudoku solution checker (SUDOKU), as well as implementations of concurrent models on top of threads and shared memory: software transactional memory (STM), the actor model (ACTORS), and a meta-circular evaluator with support for threads (MCEVAL).

2.4.3. Soundness Testing

We have proven that a naive application of the AAM design method to λ_α and λ_τ is sound (Theorems 3 and 5). In addition, we provide empirical evidence for the soundness of our implementation through *soundness testing* (Andreasen *et al.*, 2017). To this end, we verified that all information recorded during 1000 concrete runs of each benchmark program is soundly over-approximated by the analysis of the same program. No unsound results were reported for any of the supported benchmarks, i.e., the analysis implementation over-approximated every value that was observed during the concrete executions. Note that only 6 of the benchmark programs are analyzed within the given time budget of 30 minutes (see next section), but we repeat the soundness testing of our implementation in Chapters 4 and 6.

2.4.4. Running Time

To evaluate the efficiency of the analysis, we analyzed each of the benchmark program 20 times, after 10 warmup runs, and report on the average time required to analyze each program. We set a time-out of 30 minutes, after which running time is denoted as infinite (∞). The results are listed in Table 2.2. It is clear from these results that a naive application of AAM for static analysis of concurrent program lacks in scalability: only 6 of the 56 programs are analyzed within the time limit, and of those only 3 are analyzed in less than a minute. These program either feature a constant set of processes known in advance (PP, COUNT, ABP, DEKKER), or an dynamic set of actors that only receives messages without sending any message (FJT, FJC).

As only few benchmark programs can be analyzed within the time budget of 30 minutes, and because the analyses resulting from application of the MacroConc design

method described in Chapter 4 yields the same precision as the analyses developed in this chapter, we defer the discussion of the precision of the analyses to Chapter 4.

Actors				Threads			
Bench.	Time (ms)	Bench.	Time (ms)	Bench.	Time (ms)	Bench.	Time (ms)
PP	2876	BTX	∞	ABP	92163	TRAPR	∞
COUNT	920	RSORT	∞	COUNT	∞	ATOMS	∞
FJT	435812	FBANK	∞	DEKKER	20675	STM	∞
FJC	528242	SIEVE	∞	FACT	∞	NBODY	∞
THR	∞	UCT	∞	MATMUL	∞	SIEVE	∞
CHAM	∞	OFL	∞	MCARLO	∞	CRYPT	∞
BIG	∞	TRAPR	∞	MSORT	∞	MCEVAL	∞
CDICT	∞	PIPREC	∞	PC	∞	QSORT	∞
CSLL	∞	RMM	∞	PHIL	∞	TSP	∞
PCBB	∞	QSORT	∞	PHILD	∞	BCHAIN	∞
PHIL	∞	APSP	∞	PP	∞	LIFE	∞
SBAR	∞	SOR	∞	RINGBUF	∞	PPS	∞
CIG	∞	ASTAR	∞	RNG	∞	MINIMAX	∞
LOGM	∞	NQN	∞	SUDOKU	∞	ACTORS	∞

Table 2.2.: Running times of the analyses presented in this chapter on our benchmark programs. The times are expressed in milliseconds and represent the average of 20 runs after 10 warmup runs. ∞ denotes that the analysis timed out after 30 minutes.

2.5. Conclusion

In this chapter, we introduced the two concurrent languages that are used throughout this dissertation: a concurrent actor programming language λ_α, and a shared-memory multi-threaded language λ_τ. Both languages are based on the same sequential subset of Scheme formalized by the λ_0 language. We provided the concrete semantics of each language, and performed a naive application of the AAM design method to obtain static analyses for concurrent programs written in λ_α and λ_τ. This naive application does not perform any optimization with respect to scalability, explicitly representing all possible interleavings of the transition relation. We formally proved that the resulting analyses are sound and terminate, and empirically evaluated their running time. While the analyses are sound, they fail to analyze most benchmarks within a budget of 30 minutes. Only 6 of our 56 benchmark programs can be analyzed within this time budget, demonstrating the scalability issues of these analyses. However, the analyses do exhibit almost all of the desirable properties identified in Chapter 3: they are automated, sound and support dynamic process creation. We defer the evaluation of their precision to Chapter 4. The main limitation of these analyses are their scalability. In the following chapters, we present analysis design methods inspired by this naive application of AAM to concurrent language, that improve the scalability of the resulting analysis.

3

STATE OF THE ART IN STATIC ANALYSIS OF CONCURRENT PROGRAMS

In this chapter, we review existing analyses for concurrent programs, observing that no existing analysis combines automation, soundness, scalability, precision and support for dynamic process creation (Section 3.1). Through this review, we identify two predominant approaches to improve the scalability of static analyses for concurrent programs (Section 3.2): state space reduction methods that mitigate the state explosion problem within an analysis, and process-modular analyses designs that overcome the problem from the ground up. As such, both approaches represent potential avenues to address the scalability issues of the analyses described in Chapter 2.

3.1. Static Analyses of Concurrent Programs

We review here the existing analysis methods that target concurrent programs and we identify their shortcomings. We observe that no existing method features all the desired properties explained in Section 1.2: automation, soundness, scalability, precision, and support for dynamic process creation.

Sound static analyses may report defects that will never arise at run time, called *false positives*, which stem from imprecision, often due to a too coarse abstraction. Whether a detected defect is a false positive or a true defect has to be investigated by the user of the analysis, which becomes a burden if the number of false positives is high (Cousot, 2005). Static analysis is an undecidable problem and therefore cannot be both sound (over-approximates all behaviors) and maximally precise at the same time. Some analyses sacrifice soundness in order to achieve maximal precision. Such analyses are maximally precise in the sense that any detected error is an error that will arise under specific conditions (no false positive), but such analyses are not sound in the sense that they may miss errors (potential false negatives). Avoiding false positives is desirable to reduce the burden on the user of an analysis, as developers are prone to ignore the results of imprecise analyses that report too many false positives (Johnson *et al.*, 2013; Bessey *et al.*, 2010). However, false negatives result in an analysis that cannot be trusted: if the analysis reports no error, the program may still be erroneous. Other analyses are neither sound nor maximally precise, and may therefore produce false positives and miss defects, but do so with the goal of achieving a high precision. Dynamic analysis approaches also sacrifice soundness because they cannot reason about all possible behaviors of the program under analysis and will miss potential errors. The dynamic approaches supporting concurrency aim at providing a precise under-approximation of all the behaviors of the program for precise defect detection, rather than at providing a sound over-approximation for proving program properties.

We aim for static analyses that, in contrast, are sound: the analyses account for all possible program inputs and interleavings and do not produce false negatives. If such an analysis does not report a defect, it therefore proves that the given program is free of that defect. Sacrificing soundness to improve precision is not compatible with provably verifying or inferring properties of a concurrent program.

3.1.1. Bug Finding

We discuss existing bug finding approaches targeted at concurrent programs as this gives a better picture of the current state of concurrent program analyses. We defer the discussion of model checking to Section 3.1.4 as it is a large field of research that deserves its own section, even though most of the model-checking related work is maximally precise but only sound under specific assumptions, and therefore fall under the bug finding methods.

Bug Finding Tools for Actor-Based Programs

The dCUTE (Sen and Agha, 2006b), Basset (Lauterburg *et al.*, 2009) and Concuerror (Christakis *et al.*, 2013) tools perform automated testing of actor-based programs, and are therefore dynamic approaches. They rely on concolic testing (Sen, 2007) and incorporate partial-order reduction. Partial-order reduction methods reduce the state space that has to be explored by an analysis, by identifying equivalent process interleavings, thereby enabling the analysis to explore only one interleaving among a group of equivalent interleavings. We present partial-order reduction methods in more details in Section 3.2.1. dCUTE and Concuerror rely on dynamic partial-order reduction (Flanagan and Godefroid, 2005), while Basset includes both dynamic partial-order reduction and an actor-specific state comparison reduction. As these analyses rely on concrete execution of the program, they are not prone to false positives, but are not sound and only terminate if the program itself terminates.

The Erlang static analysis tool Dialyzer (Sagonas, 2007) incorporates multiple analyses geared towards concurrent programming bugs. The race condition analysis of Dialyzer (Christakis and Sagonas, 2010) detects race conditions that may appear between actors that make use of unsafe components of Erlang where memory is shared, such as the Erlang term storage. This analysis first detects all static locations where potentially interfering calls to unsafe components are made, and then filters out combinations of interfering calls that are safe according to unsound but precise conditions. The deadlock analysis of Dialyzer (Christakis and Sagonas, 2011) detects deadlocks that may appear due to incorrect assumptions made by the programmer on the ordering of messages. This analysis is prone to false positives *and* to false negatives, but tries to strike a balance between precision and scalability. These two analyses are based on the escape analysis of Carlsson *et al.* (2003), which aims at detecting functions that can escape their defining function in Erlang. As only escaping functions can be used to spawn new processes in Erlang, this reduces the number of functions to investigate when an unknown process is created. This is particularly efficient for Erlang programs where higher-order functions are used more scarcely than in other languages such as Scheme (Carlsson *et al.*, 2003).

Bug Finding Tools for Multi-Threaded Programs

jCUTE (Sen and Agha, 2006c; Sen and Agha, 2006d; Sen and Agha, 2006a) is a concolic tester for multi-threaded Java programs supporting dynamic process creation. Any error found by jCUTE is a true error as, being a concolic tester, it performs concrete executions of the program under analysis. However, it relies on an oracle to solve path constraints, and unless this oracle can solve all path constraints discovered in the program under test, which is not the case in general, it is not sound. It is also unsound if the program under test has an infinite number of possible paths of execution or a possibly infinitely long executions.

FindBugs (Hovemeyer and Pugh, 2004) performs syntactic checks on Java programs to detect shared-memory concurrency bugs. Among the 424 syntactic checks that are performed by FindBugs, 46 of them fall under the *multi-threaded correctness* category. These

lightweight syntactic checks aim to detect bugs that are specific to the Java concurrency model. FindBugs is neither sound nor maximally precise, as it is prone to both false positives and false negatives.

A large body of work is dedicated to the detection of race conditions in multi-threaded programs. The survey of Hong and Kim (2015) classifies the different types of race detections that have been studied, and surveys 43 race detectors. All the surveyed detectors focus on bug detection rather than on sound analyses.

Various other analyses focus on deadlock detection, either based on testing (Fonseca *et al.*, 2011) or by analyzing lock usage statically (Artho and Biere, 2001; Williams *et al.*, 2005; Engler and Ashcraft, 2003; Naik *et al.*, 2009). However, these analyses are not sound and are closely tied to the concurrency model they analyze.

Bug Finding for Other Concurrency Models

Albert *et al.* (2016) propose a testing method for deadlock detection in programs using futures (Baker and Hewitt, 1977). This is performed through a combination of a testing strategy and a static analysis. The static analysis is based on a may-happen-in-parallel deadlock analysis (Flores-Montoya *et al.*, 2013) that detects potential deadlock cycles. This information is then used to guide the testing phase to find actual deadlocks. This method is limited to programs where the number of futures and their location is known a priori. The testing strategy features macro-stepping semantics (Agha *et al.*, 1997) to reduce the state space that the analysis has to explore, similarly to the static analyses presented in Chapter 4, but Albert *et al.* (2016) use it only in a dynamic analysis setting.

Carver and Lei (2004) present a general model to perform *reachability testing* for diverse concurrency models (asynchronous and synchronous message-passing, as well as shared memory protected through semaphores, locks, and monitors). Reachability testing combines non-deterministic testing, which executes a program with a given input but does not enforce a specific process interleaving, with deterministic testing, which executes a program with a given input *and* a fixed process interleaving. This combination runs a test case deterministically up to a pre-determined point identified as the prefix of a potential error, after which the testing proceeds in a non-deterministic manner in order to find the error. This testing method is not prone to false positives but is unsound.

3.1.2. Abstract Interpretation

Abstract interpretation is the method used by the analyses described in Chapter 2, which form the baseline for this dissertation. In contrast to bug finding approaches of the previous section, abstract interpretation aims at providing a sound foundation for static analysis through sound abstractions of the semantics of the analyzed programming language. There exists a number of applications of abstract interpretation to concurrent languages.

Abstract Interpretation of Actor-Based Programs

Huch (1999) represents some of the earliest work on static analysis of actor-based programs through abstract interpretation. This work identifies four sources of unboundedness that render analyzing actor programs challenging: data unboundedness, stack unboundedness, mailbox unboundedness, and unboundedness resulting from dynamic process creation. Huch (1999) shows how abstract interpretation can be used to solve the first two sources of unboundedness, and mitigates the other two by framing the analysis in the context of programs that have finite mailboxes and a finite number of processes known a priori. However, actor programs tend to contain unbounded mailboxes as well as an unbounded number of processes created at run time.

Support for unbounded mailboxes and unbounded dynamic process creation has been developed by the Soter tool (D'Osualdo *et al.*, 2012; D'Osualdo *et al.*, 2013), which abstracts mailboxes to sets of messages in order to ensure their finiteness. Soter also maps multiple concrete actors to a single abstract actor to deal with unbounded process creation. In this sense, Soter is close to the actor analysis we present in Chapter 2 by explicitly representing interleavings of different abstract actors. However, one major difference with our work is that Soter verifies properties in a two-step fashion: first a model of the program is constructed, then this model is used to perform model checking of user-annotated properties. The analysis presented in Chapter 2 enables direct verification on the model without requiring further model checking. Another difference is that Soter performs coarse abstractions which are later refined in the model checking phase, enabling Soter to support more programs than the actor analysis of Chapter 2, bringing it closer in terms of efficiency to the actor analysis developed in Chapter 4. We demonstrate this in Chapter 4.

Garoche *et al.* (2006) present an abstract interpretation approach to verify properties of an actor calculus. The focus is on abstractions that enable reasoning about the number of actors bound to a process identifier. This analysis is later extended by Garoche (2008) to detect orphan messages in actor programs, using a *vector addition system*. Similarly to D'Osualdo *et al.* (2013), the verification is performed in two phases: first a model is constructed through abstract interpretation, and model checking is then performed on the constructed model. This approach is sound by construction, but the largest program this approach has been applied to is 17 lines long, which took 52 seconds to analyze, raising questions about its scalability.

Abstract Interpretation of Multi-Threaded Programs

Might and Van Horn (2011) apply the *abstracting abstract machines* (Van Horn and Might, 2010) design method in a shared-memory concurrency setting. The resulting analysis is a non-modular analysis of threads supporting dynamic thread creation and shared memory through atomics, where shared memory can be mutated through a *compare-and-swap* operation. This is close to the analysis discussed in Chapter 2, with the difference that our shared-memory multi-threading model relies on locks to protect shared memory rather than atomics. Analyses resulting from a naive application of AAM to concurrent

programs fail to scale beyond synthetic programs as demonstrated in Chapter 2. Might and Van Horn (2011) hint at a possible solution to obtain polynomial complexity by performing a further abstraction that joins all the states explored during the analysis into a single state. This results in an improved complexity at the cost of precision. This further abstraction is used in the Soter tool (D'Osualdo *et al.*, 2012), against which we compare throughout this dissertation.

Another approach to abstract interpretation of concurrent higher-order programs is the work of Jagannathan and Weeks (Weeks *et al.*, 1994; Jagannathan and Weeks, 1994; Jagannathan, 1994). These analyses target compiler optimizations, and support both dynamic process creation and higher-order functions. The abstract interpreter in Jagannathan and Weeks (1994) is written in a concurrent language and its scalability is improved by analyzing the different processes concurrently. The soundness of this line of work is not demonstrated, nor is its ability to analyze more than synthetic benchmarks.

Miné (2014) introduces a process-modular abstract interpretation that reasons about values of variables in a concurrent, shared-memory setting. Its incarnation in the AstréeA tool (Miné, 2012) has been able to analyze commercial programs of up to 1.7 million lines of code with 15 pre-defined running threads in a few days. However, the abstract interpreter is limited to programs with a constant number of running processes known a priori.

Predicate abstraction (Flanagan and Qadeer, 2002) is a form of abstract interpretation where the abstract domain is constructed from a set of predicates that are generated from the source code of the program under analysis. Predicate abstraction has been adapted to a multi-threaded context (Gupta *et al.*, 2011) based on rely-guarantee reasoning (Jones, 1983). This approach can either infer modular or non-modular proofs for multi-threaded programs. However, it is limited to programs with a constant set of processes known a priori.

Abstract Interpretation of Other Concurrency Models

A number of static analyses have been developed for channels concurrency (Hoare, 1978). Midtgaard *et al.* (2016b) introduce a process-modular analysis for synchronous message-passing programs, but the analysis is limited to programs composed of two processes. Other analyses explore all process interleavings explicitly and are either limited to programs with a fixed set of processes (Mercouroff, 1991; Ng and Yoshida, 2016; Stadtmüller *et al.*, 2016), make the simplifying assumption on the concurrency model that communication occurs only through explicitly identified channels (Colby, 1995; Martel and Gengler, 2000), or assume that senders and receivers are statically determined (Ladkin and Simons, 1992).

Software transactional memory (Shavit and Touitou, 1997) has seen static analysis support dedicated to specific compiler optimizations (Afek *et al.*, 2010), but lacks support from general-purpose static analyses.

3.1.3. Type Systems

Static type systems aim at restricting the number of valid programs so that only programs that satisfy the properties expressed by the types can be executed. Soundness in type systems is a crucial property. Type systems are modular by nature, scale well, and support dynamic process creation. While verifying and inferring context-insensitive and flow-insensitive properties is well-supported by type systems, supporting stronger properties remains challenging and renders the type system closer to a proof system, which are discussed in Section 3.1.5

Type Systems for Actor-Based Programs

Several type systems have been proposed for actor programs. Early work focuses on detecting type errors in the sequential subset of the language (Lindgren, 1996; Marlow and Wadler, 1997), without dedicated support for concurrency. While useful to prevent errors in the sequential subset, this approach cannot reason about concurrent properties.

Dagnat and Pantel (2002) introduce a type-driven static analysis that infers interfaces for actors in a subset of Erlang with the goal of detecting orphan messages. Orphan messages are messages that are sent to an actor but that are never handled by the receiving actor. The approach supports dynamic process creation, but is not shown sound nor scalable.

Pony is a language that combines actors and shared memory in a type-safe, memory-safe, data-race free and deadlock-free manner (Clebsch *et al.*, 2015), thanks to its use of capabilities (Miller *et al.*, 2003). Such a language facilitates writing safe concurrent actor programs. Our approach is however not of extending the analyzed language with new features but rather providing support for analysis for existing languages.

Session types enable verifying whether a program adheres to a protocol. Mostrous and Vasconcelos (2011) present session types for Erlang capable of ensuring that sent messages have the expected types. In multi-party asynchronous session types (Honda *et al.*, 2008; Neykova and Yoshida, 2014; Honda *et al.*, 2016; Scalas *et al.*, 2017), the protocol is expressed as a *global type* that describes the behavior of multiple parties of the system under analysis. However, multi-party session types rely on channels used by at most two entities at a time, while in the actor model any number of actors can send messages to an actor. Moreover, relying on a global type to specify the protocol requires advanced knowledge about the topology of the system, while actor systems are inherently dynamic. Finally, while session types are expressive, their static verification is complex and state of the art approaches fall back to dynamic checking of session types when static checking fails (Neykova and Yoshida, 2014)

Type Systems for Multi-Threaded Programs

Several type systems for multi-threaded programs have been proposed to ensure the absence of concurrency errors such as race conditions (Boyapati and Rinard, 2001; Boyapati *et al.*, 2002; Flanagan and Freund, 2000) and deadlocks (Boyapati *et al.*, 2002; Flanagan

and Abadi, 1999), to ensure determinacy of the concurrent executions (Bocchino Jr. *et al.*, 2009), or to verify method atomicity (Flanagan and Qadeer, 2003a). The resulting programs tend to require heavy type annotations before they are accepted by the type checker. Further work has shown that some of the required annotations can be inferred automatically (Abadi *et al.*, 2006; Sasturkar *et al.*, 2005).

Type Systems for Other Concurrency Models

Nielson *et al.* (1999) introduce a type and effect system that performs a communication analysis of typed channels, inferring which values are communicated over the channels, while recording information about the temporal orders of effects. It remains unclear how such an analysis could be adapted to an untyped dynamic setting.

Haskell support for STM (Jones *et al.*, 1996) ensures that transactional memory is used in a safe manner through statically-typed monadic operations. The static verification relies heavily on the type system of Haskell and is not applicable to untyped dynamic languages such as our extensions to λ_0 introduced in Chapter 2.

3.1.4. Model Checking

Model checking entails verifying that a property, generally expressed in a property specification language, holds for a program. The program under analysis is sometimes expressed as a model in a model specification language. This model may also be derived directly from the source code of the program (Godefroid, 1997; Corbett *et al.*, 2000; Havelund and Pressburger, 2000; Fredlund and Svensson, 2007). Conventional model checkers perform explicit state exploration: all reachable states in the execution of a program are explored, and the property is verified for all those states.

The state space is explored either in a stateful or in a stateless manner, respectively leading to *stateful model checking* or *stateless model checking*. In stateful model checking, all encountered states are stored to prevent visited states from being visited again. The main limitation of this approach is that program states, which may contain a large amount of information, have to be encoded in a memory-efficient manner. Stateless model checking avoids this limitation by not storing the set of explored states, hence removing the need to encode program states. However, this comes at the cost that cycles in the execution of a program (e.g., infinite loops) cannot be detected. Therefore, stateless model checking does not terminate in the general case, and is sound only when it terminates.

A number of state space reductions have been applied in the context of model checking, of which we defer the technical discussions to Section 3.2. Such reductions come either in the form of static partial-order reductions (Godefroid, 1996) or dynamic partial-order reductions (Flanagan and Godefroid, 2005), and are limited with respect to supporting dynamic process creation.

Model Checking of Actor-Based Programs

Dam and Fredlund (1998) introduce a specification logic and proof system for Core Erlang programs, a subset of Erlang used by the BEAM Erlang virtual machine. This specification logic and proof system can be used to perform model checking on Erlang programs. This work has been integrated in the Erlang Verification Tool (Arts *et al.*, 1998), later extended to support specific constructs of the widely-used OTP[1] framework (Arts and Noll, 2000). It supports verifying that an implementation satisfies a given specification, but lacks automation and is limited to a client-server setting where the notion of dynamic process creation is not present.

The analyses present in dCUTE (Sen and Agha, 2006b), Basset (Lauterburg *et al.*, 2009), and Concuerror (Christakis *et al.*, 2013) are concolic, but can also be described as stateless model checking. They rely on a variant of dynamic partial-order reduction (Sen and Agha, 2006a) that supports dynamic process creation.

TransDPOR (Tasharofi *et al.*, 2012) performs stateless model checking and relies on an extended version of dynamic partial-order reduction, taking advantage of the fact that the dependency relation describing dependencies among transitions in actor systems is transitive. This enables further reductions of the number of program paths that have to be explored. It is limited to acyclic state spaces like the original dynamic partial-order reduction (Flanagan and Godefroid, 2005) and to programs with a constant set of processes that is known a priori.

McErlang (Fredlund and Svensson, 2007) introduces a model checker for a distributed version of Erlang, but is also limited to acyclic state spaces and to programs with a constant set of processes that is known a priori.

Lauterburg *et al.* (2010) evaluate the impact of different ordering heuristics on the reduction performed by dynamic partial-order reduction when analyzing actor programs. It is shown that analyses are highly sensitive to the order of exploration, as differences of two orders of magnitude can be observed between different heuristics.

Model Checking of Multi-Threaded Programs

While model checking initially focused on communication protocols, it has also been applied to multi-threaded programs (Godefroid, 1997; Holzmann, 2004; Flanagan and Qadeer, 2003b; Flanagan *et al.*, 2002). The initial description of partial-order reduction (Godefroid, 1996) and dynamic partial-order reduction (Flanagan and Godefroid, 2005) rely on a multi-threaded model. Kokologiannakis *et al.* (2018) present an alternative for dynamic partial-order reduction in stateless model checking of applications for weak memory models. However, these approaches are all framed in a context of a constant set of threads that is known a priori.

Java PathFinder (Havelund, 1999) supports verifying multi-threaded Java programs (Ujma and Shafiei, 2012), by reducing the size of the states by abstracting instances of classes from the `java.util.concurrent` package to integers, and relying on the JVM performing the model checking to map from integers to actual class instances. This does not

[1] OTP, the *Open Telecom Platform*, is a set of libraries and tools for designing Erlang applications.

solve the state explosion problem as the complexity of the analysis remains exponential, but significantly reduces the verification time on benchmarks for concurrent data structures from an hour to a few minutes (Ujma and Shafiei, 2012).

Khurshid *et al.* (2003) extend model checking to handle unbounded data input. Relying on Java PathFinder to perform the model checking, this analysis supports multi-threaded applications written in Java and profits from the mitigation of Java PathFinder for reducing the size of the states, but it is also subject to the state explosion problem.

Model Checking of Other Concurrency Models

Jensen *et al.* (2015) apply stateless model checking featuring dynamic partial-order reduction to event-driven applications. Event-driven applications exhibit similar behavior as concurrent programs, even though they are executed using a single thread. These applications also exhibit nondeterministic behavior and contain errors that may manifest only under specific interleavings of events.

Kastenberg and Rensink (2008) introduce a novel dynamic partial-order reduction that supports dynamic enabling and disabling of transitions in a transition system. This is framed in the context of analyzing graph transformation systems, but is also applicable to concurrent programs. This brings dynamic partial-order reduction to systems where processes may be dynamically enabled or disabled but lacks support for dynamic process creation where the behavior of the created processes is not known statically.

3.1.5. Proof Systems

Proof systems can be used to prove challenging properties of concurrent programs, but fall short with respect to automation. Proof systems generally require expert knowledge from the user to express theorems and to fill in the proof details. This requires a significant effort from the developer for complex and for larger programs. This is why in this dissertation we aim for automated static analysis instead, which requires minimal intervention from the user of the analysis.

Proof Systems for Actor-Based Programs

Proof systems for Erlang such as Rebeca (Sirjani *et al.*, 2005; Sirjani and Jaghoori, 2011), the Erlang Verification Tool (Arts *et al.*, 1998; Arts and Noll, 2000), and the work by Dam and Fredlund (1998) all require proof system expertise to prove the correctness of a program.

Proof Systems for Multi-Threaded Programs

Concurrent separation logic (O'Hearn, 2007) enables reasoning about thread-based concurrent systems in a modular way, focusing on resource usage. Automated analyses that infer logic formulas for non-concurrent separation logic have been designed (Calcagno *et al.*, 2009), but an automated inference for concurrent separation logic is yet to be designed. Rely-guarantee reasoning (Jones, 1983) is similar to concurrent separation

logic but tends to lead to longer proofs as it explicitly models non-interferences, while concurrent separation logic models them implicitly. Sergey *et al.* (2015) provide a proof system for verifying concurrent programs that make use of the compare-and-swap primitive, and prove a number of small concurrent programs correct. This proof system is a Coq library that relies on *fine-grained concurrent separation logic* (Nanevski *et al.*, 2014) to encode properties of concurrent programs. This version of concurrent separation logic enables compositional proofs for concurrent programs featuring higher-order functions and dynamic thread creation. The size of the programs verified in Sergey *et al.* (2015) varies from 27 to 305 lines of code and the size of the proofs from 26 to 1744 lines of code, indicating that the effort invested in the proofs may be significantly higher than the effort to write the programs themselves.

3.1.6. Overview

Having reviewed the state of the art in analyses for concurrent programs, Table 3.1 gives an overview of the existing analyses with respect to our desired properties: automation enabling users without tool expertise to rely on the analysis, soundness enabling users to trust the analysis for proving certain program properties, scalability enabling the analysis to support large programs, precision to minimize the number of false positives detected by the analysis, and support for dynamic process creation enabling the analysis to support modern concurrent programs We group analyses that share the same aspects, and list analyses based on abstract interpretation separately as they exhibit different ranges of features. No existing analysis exhibits at the same time all these desirable properties.

Bug finding

Bug finding analyses are often maximally precise at the cost of soundness, and are therefore subject to false negatives (undetected bugs). Such analyses may scale well because of the soundness sacrifice performed, and support dynamic process creation. They are generally automated as their goal is to help developers with a minimal investment cost.

Abstract interpretation

Abstract interpretation analyses are automated, sound, and precise but differ in terms of scalability and support for dynamic process creation. The work of Miné (2014) has been demonstrated to scale to large programs (Miné and Delmas, 2015), but lacks support for dynamic process creation. Conversely, several existing abstract interpretations (D'Osualdo *et al.*, 2013; Garoche *et al.*, 2006; Might and Van Horn, 2011; Jagannathan and Weeks, 1994) support dynamic process creation but are limited in terms of scalability.

Type systems

Type systems are generally sound and scalable by design, and support dynamic process creation. Type systems are automated and require no intervention from the user, except for annotating the program with types, but are limited in their

analysis to reasoning about relatively straightforward properties. Type systems that can express more complex properties fall into the proof systems category, and they require significantly more user intervention. Type systems reject programs that are deemed unsafe, and tend to sacrifice precision, therefore rejecting safe programs that the type system cannot detect as safe.

Model checking

Model checkers are sound under the assumption that the program under analysis always terminates. They are maximally precise and can provide execution traces that lead to a detected error, but they fail to scale despite their use of mitigations to the state explosion problem, such as partial-order reduction. Moreover, model checkers perform verification of a program against a specification expressed in a specification language, which has to be devised by the user of the tool, and therefore fall short with respect to automation. The program analyzed also sometimes has to be expressed in a modeling language, although research automating the construction of the model exists.

Proof systems

Proof systems are sound and can be very precise, but lack automation, leaving a significant amount of work to the user of the system. Once the proof has been encoded in the proof system, running the proof can be performed in a scalable manner.

Method	Automated	Sound	Scalable	Precise	Dynamic process creation
Bug finding	✓	✗	✓	✓	✓
Abstract interpretation					
Huch (1999)	✓	✓	✗	✓	✗
D'Osualdo *et al.* (2013)	✓	✓	✗	✓	✓
Garoche *et al.* (2006)	✓	✓	✗	✓	✓
Might and Van Horn (2011)	✓	✓	✗	✓	✓
Jagannathan and Weeks (1994)	✓	✓	✗	✓	✓
Miné (2014)	✓	✓	✓	✓	✗
Type systems	✓	✓	✓	✗	✓
Model checking	✓	~	~	✓	✗
Proof systems	✗	✓	✓	✓	✓

Table 3.1.: Overview of properties of static analyses for concurrent programs. A check mark (✓) indicates that a given approach *may* support the property of interest. For example, while most model checkers are not automated as they require encoding a model and a specification by hand, there exists work on automating the model construction and there exist pre-defined specifications for specific classes of bugs, hence a check mark is used. A cross mark (✗) is used when no existing work in the field brings support for the specific properties. A tilde (~) is used for analyses that support a property partially, for example by making specific assumptions about the program under analysis.

3.2. Research Approach: Towards Scalable Analyses

The application of AAM to concurrent programs described in Chapter 2 supports modern concurrent programs that feature dynamic process creation. It is sound and automated, but lacks scalability as it fails to analyze most of our benchmark programs. This is a result of the inherent non-determinism in concurrent programs, which the analyses resulting from this naive application account for explicitly by exploring all process interleavings. Having reviewed the state of the art, we now discuss the two main approaches to remedying the scalability issues. State space reduction mitigates the state explosion problem without sacrificing precision, while a process-modular analysis design eliminates the state explosion problem entirely at the cost of precision.

3.2.1. State Space Reduction

The number of possible process interleavings in concurrent programs is typically high, but different interleavings may be equivalent with respect to some properties. The idea behind state space reduction is to only explore one interleaving among each group of equivalent interleavings, thereby still accounting for all possible behaviors of a program without exploring all possible interleavings explicitly.

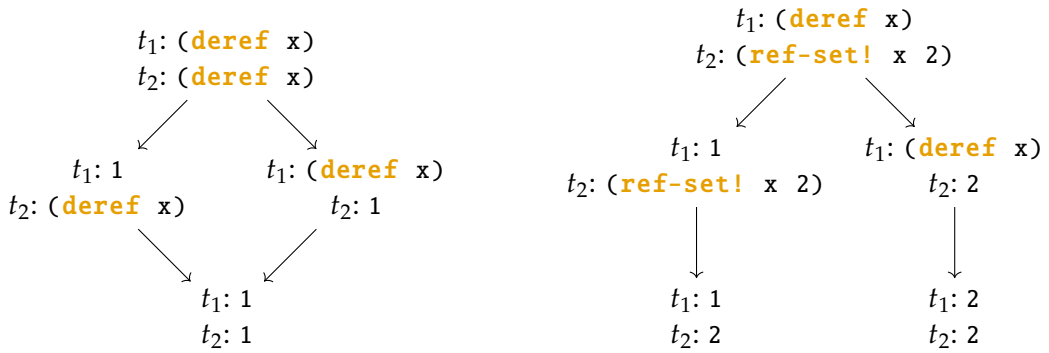
We review here the most prevalent partial-order reductions based on *persistent sets* (Godefroid, 1996). In a program composed of concurrent processes, processes transition between states. From a state, a number of transitions can be taken at any point in time, resulting in non-determinism. The set of transitions that can be taken from a given state is called the set of *enabled transitions*. Analyses featuring partial-order reduction explore only a subset of enabled transitions at every state. The subset of transitions that can be explored from a given state while being sufficient to verify a property is called a *persistent set*. Starting from the initial state, analyses incorporating partial-order reduction always explore a persistent set of the enabled transitions at each discovered state. This results in sound analyses that explore possibly fewer than all the possible interleavings, but still account for them.

Consider a program with two threads, t_1 and t_2, where each thread is ready to perform a state transition.

(a) If the operation performed by each transition does not influence the applicability or the outcome of the transition of the other thread, having t_1 transition first and then only t_2 or having t_2 transition first and then only t_1 does not influence the result of the program. This is for example the case when both transitions read from, but do not write to, the contents of a reference. This situation is depicted in Figure 3.1a. In this case there are two persistent sets from the initial state: the set containing only the transition of t_1, and the set containing only the transition of t_2. An analysis may explore only one of the two interleavings, and still remain sound.

(b) If the transitions do influence each other, then the end result of the program will depend on the interleaving, and an analysis has to take into account both interleavings. This is for example the case if t_1 modifies a reference and at the

same time t_2 accesses the same reference. This situation is depicted in Figure 3.1b. In this case, there is only one persistent set at the initial state, containing both enabled transitions. This means that if t_1 transitions first, the result of the program may be different from the result where t_2 transitions first. For an analysis to be sound, it needs to account for both interleavings in its results.



(a) Non-interfering processes. **(b)** Interfering processes.

Figure 3.1.: Representation of the concurrent execution of two threads in an all-interleavings analysis. Each program state represents the current expression evaluated or the value reached by each thread (written $t_i : e$ or $t_i : v$ for each thread). Edges represent transitions performed in the execution of the program.

Persistent sets can be computed in a number of ways. Godefroid (1996) compares three different algorithms to compute persistent sets statically, all performing an analysis on the static structure of the code of the program under analysis to infer the possible future operations that may be performed after an enabled transition has been taken.

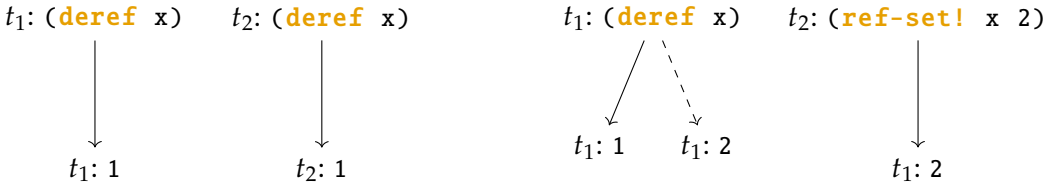
Adapting such reductions to dynamic programs is problematic. First, because static partial-order reduction requires a static analysis to reason about the possible future transitions performed in the execution of a process in order to construct the persistent sets. While this may be feasible for programs with a relatively static control-flow, it is not adapted for programs in dynamic languages such as our extensions to λ_0 considered in Chapter 2. Reasoning about the possible future executions of such program requires a full-fledged static analysis, which is also subject to the state explosion problem. Dynamic partial-order reduction (Flanagan and Godefroid, 2005), in contrast, records information at run time about the current execution of the program to infer the persistent sets. It perform stateless model checking to this end, and is therefore limited to reasoning about programs that terminate. Dynamic partial-order reduction has been transposed to stateful model checking by Yi *et al.* (2006) and by Yang *et al.* (2008), but none of these extensions support dynamic process creation or possibly infinite state spaces.

Neither static nor dynamic partial-order reduction can therefore be incorporated as such in the analyses resulting from the naive application of AAM presented in Chapter 2. However, the general idea of only exploring a subset of all possible process interleavings is applicable. This is what we propose in Chapter 4, where we base ourselves on the

notion of a *macro step* instead. Agha *et al.* (1997) introduced the concept of macro-stepping to facilitate reasoning about concrete semantics of actor programs. A macro step is defined as multiple small steps performed within a single actor between the reception of two messages. This concept has been used by dynamic tools (Sen and Agha, 2006b; Lauterburg *et al.*, 2009; Albert *et al.*, 2016) to reduce the state space explored during dynamic analysis, but has not yet been applied in an abstract setting. We revisit this concept in a static analysis setting as part of our MacroConc analysis design method in Chapter 4.

3.2.2. Process-Modular Analysis Design

While state space reductions mitigate the state explosion problem, they do not reduce the worst-case time complexity of an analysis, which remains exponential (see Section 4.5). A more thorough solution to the problem is to adopt a process-modular analysis design following from the modular analysis notion of Cousot and Cousot (2002). This is the approach taken by Miné (2014), by type systems, and by proof systems. Instead of reducing the number of interleavings to explore, process-modular analyses account for all interleavings through over-approximation, without explicitly exploring every interleaving separately. This is done by analyzing the program on a per-process basis. Each process is analyzed in isolation, and every interleaving of the analyzed process with other processes is deemed possible. Processes that may conflict need to be analyzed more than once to account for possible conflicts, hence a process-modular analysis will iterate over multiple analyses of the set of processes. However, it can be ensured that the maximal number of iterations does not grow exponentially. This comes at a cost in terms of precision, as interleavings are not explicitly represented or explored but are over-approximated.



(a) Non-interfering threads. **(b)** Interfering threads.

Figure 3.2.: Representation of the concurrent execution of two threads in a process-modular analysis. Each node denotes the current expression evaluated by a thread or the value reached by this thread. Edges represent transitions performed during the analysis of a thread. Plain edges are transitions explored during the first iteration of a process-modular analysis, and dashed edges are explored during the second iteration.

We revisit our previous example in the setting of a process-modular analysis. There are two threads to analyze: t_1 and t_2, and each thread is analyzed in isolation.

(a) In the case of non-interfering transitions, each thread is analyzed just as any sequential program, and the complexity of the analysis becomes the complexity

of a sequential analysis multiplied by the number of threads. This is depicted in Figure 3.2a. Note that the interleavings of the different threads are not explicitly represented, but rather the result of the analysis is a graph per thread describing the evolution of each thread separately.

(b) In the case of interfering transitions, a first iteration of the analysis analyzes each thread in separation until completion. This is depicted by the plain edges in Figure 3.2b. Thread t_2 performs an operation conflicting with thread t_1, and therefore thread t_1 is analyzed again to account for the changes that occur from the execution of thread t_2. The dashed transition of Figure 3.2b is therefore also explored.

Although this is a very synthetic example, we can already see that the number of transitions explored is reduced compared to an analysis that explores all interleavings or to an analysis that performs state space reduction. Indeed, in the case of the non-interfering threads, an analysis with state space reduction and a process-modular analysis only explore two transitions instead of the four possible transitions. In the case of interfering processes, an analysis with state space reduction explores all four transitions, while a process-modular analysis only explores three transitions. As the number of processes and the number of transitions grow, this difference in the number of transitions and the number of states that have to be explored by each analysis grows as well.

The concept of modular analysis has been formalized by Cousot and Cousot (2002), which present a general-purpose method to design modular analyses. These ideas have been applied in the context of thread-based programs by using concepts from either assume-guarantee reasoning (Flanagan *et al.*, 2002; Henzinger *et al.*, 2003), rely-guarantee reasoning (Miné, 2014; Monat and Miné, 2017), or separation logic (Gotsman *et al.*, 2007), and this in a setting limited to a statically known number of executed threads. Recent developments (Midtgaard *et al.*, 2016b; Midtgaard *et al.*, 2016a) propose process-modular analyses for synchronous message-passing programs, but again limited to programs composed of a constant set of processes that is known a priori.

Process-modular analyses may fare very well in terms of scalability (Miné and Delmas, 2015), as they are not subject to the state explosion problem. However, they sacrifice precision for this improved scalability, as they over-approximate the interleavings of the different processes: the process interleavings are not represented, and all possible interleavings between the analyzed processes are deemed possible. The main challenge compared to existing work is that dynamic creation of processes is inherent to modern concurrent programs and must be supported by the analysis.

3.3. Conclusion

In this chapter, we presented a thorough review of the state of the art in static analyses for concurrent programs, and identified two approaches to improve the scalability of the analyses introduced in Chapter 2. These analyses resulted from a naive application of AAM, and are automated, sound, precise and support dynamic process creation, but

do not scale well. The identified approaches, namely state space reduction and a process-modular analysis design, are at the core of the analysis design methods presented in Chapters 4 and 6.

4

MACROCONC: DESIGNING MACRO-STEPPING ANALYSES

In this chapter, we present MacroConc, a design method that improves upon a naive application of AAM to concurrent programs by incorporating macro-stepping in the resulting abstract semantics. The naive applications of AAM for concurrent programs presented in Chapter 2 resulted in analyses that explore all reachable states within the collecting semantics of the analyzed programs, explicitly considering all process interleavings. These analyses suffer from scalability issue, as demonstrated in our empirical evaluation. This is due to the exponential growth of the number of interleavings to explore with the size of the program under analysis.

As a mitigation for this issue, we introduce MacroConc, an analysis design method relying on a macro-stepping variant of the concrete semantics of concurrent programs. This design method reduces the state space that has to be explored by analyses resulting from its application. At the core of this design method lies the notion of *macro step*, introduced by Agha *et al.* (1997) and used in dynamic analyses for concurrent programs (Sen and Agha, 2006b; Lauterburg *et al.*, 2009; Albert *et al.*, 2016), but which has not yet been applied in an abstract setting. We describe the MacroConc design method (Section 4.1), and discuss the properties of the analyses resulting from its application (Section 4.2). We apply MacroConc to concurrent actor programs (Section 4.3) and to shared-memory multi-threaded programs (Section 4.4). We formally prove the soundness of the resulting analyses and prove that their precision remains identical to naive applications of the AAM design method. We empirically evaluate the resulting analyses in terms of running time, precision and scalability (Section 4.5), and observe the following.

- There is a improvement of up to four orders of magnitude in running time.
- MacroConc renders the resulting analyses able to analyze more than the few benchmark programs supported by the analyses developed in Chapter 2 within the same time budget.
- We observe a high precision of 96% on our benchmark suite for an analysis that infers communication effects performed by concurrent processes.
- The scalability of the resulting analyses remains however limited, as their worst-case time complexity remains exponential.

4.1. Macro-Stepping Abstract Interpretation of Concurrent Programs

We propose macro-stepping as a way to reduce the number of interleavings explored by static analyses of concurrent programs. The concept of macro-stepping originates from Agha *et al.* (1997), who define the concrete semantics of actor programs in terms of macro steps to aid manual reasoning. It has also been used to optimize the search process in automated testing of concurrent programs (Sen and Agha, 2006b; Lauterburg *et al.*, 2009; Albert *et al.*, 2016), but has not yet been applied in an abstract setting. Inspired by Agha *et al.* (1997), we formalize the concrete semantics of concurrent programs as a macro-stepping semantics. We adapt macro-stepping to abstract semantics as part of the MacroConc analysis design method. MacroConc does not suffer from the limitations of partial-order reduction methods: it does not require a static analysis to reason about the future of the program's execution, it is not limited to acyclic state spaces, and it supports infinite state spaces through abstraction.

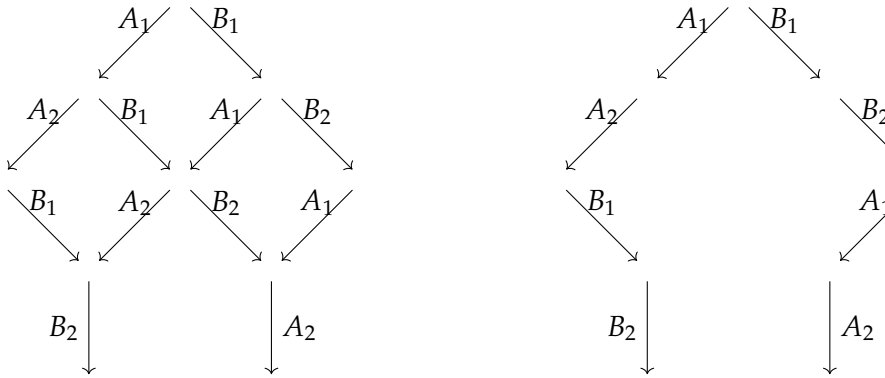
We illustrate the concepts of macro-stepping through an example. Consider a program composed of two concurrent processes A and B, where one process performs operations A_1 and A_2, and the other process performs operations B_1 and B_2. All the possible interleavings of these processes are represented in Figure 4.1a. We assume that the operations A_1 and B_1 are safe (e.g., arithmetic computations), while the operations A_2 and B_2 are acting on the shared process state and deemed potentially unsafe (e.g., writing to a shared reference or sending a message). In that case, performing A_2 before B_2 may produce a different result from performing B_2 before A_2, and it is important for an analysis to account for both interleavings. However, the interleavings of A operations alone are not important for the end result of the program, and may be ignored by an analysis.

Under macro-stepping semantics, each process is either executed until completion or up to a state where at most one potentially interfering operation has been performed. This is a macro step. After the completion of a macro step, another macro step is performed on all running processes. For our example, this is represented in Figure 4.1b: either the first process performs a macro step, executing A_1 and then A_2, followed by the second process, executing B_1 and then B_2, or vice-versa. Both final states of the programs are explored, while equivalent interleavings of the different processes are ignored. Macro-stepping semantics enables analyses to scale significantly better than naive all-interleavings analyses.

The design of an analysis with MacroConc follows four steps, which we detail in the remainder of this section.

1. The specification of an *operational semantics* for the input language featuring concurrency, defined by a transition relation annotated with communication effects. This corresponds to the semantics given for λ_α (actors) and for λ_τ (threads) in Chapter 2.

2. The specification of a *macro-stepping transfer function* that computes the set of states



(a) Interleavings considered under an all-interleavings semantics.

(b) Interleavings considered under a macro-stepping semantics.

Figure 4.1.: Representation of the executions of two concurrent processes, where one process performs operations A_1 and then A_2, while the other process performs operations B_1 and then B_2, where operations A_1 and B_1 are safe while operations A_2 and B_2 are potentially unsafe.

reachable within a single macro-step of a given process starting at a given state.

3. The specification of a *global transfer function* that drives the fixed-point computations of the macro-stepping transfer function and infers the set of reachable states under macro-stepping collecting semantics.

4. The abstraction of the macro-stepping collecting semantics, in order to obtain a finite static analysis.

4.1.1. Step 1: Definition of the Operational Semantics for the Input Language

We frame our presentation of MacroConc in the context of a concrete semantics for concurrent programs as described in Chapter 2. In such a context, a concrete transition relation $(\rightsquigarrow_p)$, parameterized by a process identifier p, defines the small-step operational semantics of concurrent programs. Transitions may be annotated with communication effects: $s \overset{eff}{\rightsquigarrow}_p s'$ indicates that the program may transition from state s to state s', generating a communication effect *eff*. The semantics for λ_α programs and for λ_τ programs have both been formalized through a transition relation fitting this description (see Sections 2.2 and 2.3 respectively).

4.1.2. Step 2: Definition of the Macro-Stepping Transfer Function

Macro steps are defined by a transfer function $\mathit{Macro}\text{-}\mathcal{G}_{s_0,p}$ that explores all reachable states in the macro step of a single process p from a given initial state s_0. We provide a generic formulation of this transfer function in Figure 4.2, relying on an *effect restriction function* $f : \mathit{Effect} \to \mathcal{P}(\mathit{Effect})$. This effect restriction function returns the set of effects

that are not allowed in a macro step after a transition generating the given effect has been taken. The transfer function itself acts on tuples $\langle S, F, E\rangle$ consisting of the set of states explored as part of the macro step (S), the set of states from which no more transition can be explored either because the process finished its execution or because that would break the macro step (F), and the set of effects that are not allowed in the remainder of the macro step (E).

$$
\begin{aligned}
\mathit{Macro}\text{-}\mathcal{G}_{s_0,p}(\langle S, F, E\rangle) = {} & \langle \{s_0\}, \varnothing, \varnothing\rangle && (1)\\
& \sqcup \bigsqcup_{\substack{s\in S \\ s \overset{\mathit{eff}}{\rightsquigarrow}_p s' \\ \mathit{eff}\notin E}} \langle \{s'\}, \varnothing, f(\mathit{eff})\rangle && (2)\\
& \sqcup \bigsqcup_{\substack{s\in S \\ s \rightsquigarrow_p s'}} \langle \{s'\}, \varnothing, \varnothing\rangle && (3)\\
& \sqcup \bigsqcup_{\substack{s\in S \\ s \overset{\mathit{eff}}{\rightsquigarrow}_{p\_} \\ \mathit{eff}\in E}} \langle \varnothing, \{s\}, \varnothing\rangle && (4)\\
& \sqcup \bigsqcup_{\substack{s\in S \\ \nexists s', s \overset{\mathit{eff}}{\rightsquigarrow}_p s'}} \langle \varnothing, \{s\}, \varnothing\rangle && (5)
\end{aligned}
$$

Figure 4.2.: Generic formulation of a macro-stepping transfer function.

Figure 4.2 defines the macro-stepping transfer function $\mathit{Macro}\text{-}\mathcal{G}_{s_0,p} : \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \times \mathcal{P}(\mathit{Effect}) \to \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \times \mathcal{P}(\mathit{Effect})$ in five parts.

1. The initial state s is part of the set of reachable states.
2. Any state s' reachable from a state s in the set of reachable states S, through a transition that generates an allowed effect, is also reachable. The set of effects that are not allowed is updated according to the effect restriction function f.
3. Any state s' reachable from a state s in the set of reachable states S, through a transition that does not generate any effect, is also reachable.
4. States s from which a transition can be performed, but that would generate an effect that is not allowed, are added to the set of final states.
5. States s from which no transition can be performed for the process p are added to the set of final states.

The fixed point of this transfer function, $\mathrm{lfp}(\mathit{Macro}\text{-}\mathcal{G}_{s_0,p})$, is a tuple $\langle S, F, E\rangle$ where S is the set of states reachable within the macro step, F is the set of final states from

which the current macro step can no longer proceed, and E is the set of effects that are not allowed. The joining of tuples is defined as a component-wise union, that is $\langle S, F, E\rangle \sqcup \langle S', F', E'\rangle = \langle S \cup S', F \cup F', E \cup E'\rangle$. The big join operator ($\bigsqcup$) joins all the elements described to its right (see Appendix A).

4.1.3. Step 3: Definition of the Global Transfer Function

The semantics of a concurrent program e is defined by a global transfer function *Global* - $\mathcal{G}_e$ that explores the entire set of reachable program states using the macro-stepping transfer function. The global transfer function *Global* - $\mathcal{G}_e : \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$ acts on tuples of sets of states $\langle S, F\rangle$, where S is the set of reachable program states, including all intermediary states visited as part of macro steps, and F is the set of final states of previously computed macro steps, from which the global transfer function starts new macro steps. We provide a generic formulation of this transfer function in Figure 4.3, where s_0 is the initial injected state of program e, with the initial stores as defined in Section 2.1.2, and where $\mathit{processes}(s)$ extracts the process identifiers of all running processes in state s.

$$\mathit{Global}\text{-}\mathcal{G}_e(\langle S, F\rangle) = \langle \{s_0\}, \{s_0\}\rangle \tag{1}$$

$$\sqcup \bigsqcup_{\substack{s \in F \\ p \in \mathit{processes}(s) \\ \langle S', F', \_\rangle = \mathrm{lfp}(\mathit{Macro}\text{-}\mathcal{G}_{s,p})}} \langle S', F'\rangle \tag{2}$$

Figure 4.3.: Generic formulation of a global transfer function in terms of a macro-stepping transfer function.

The global transfer function *Global* - $\mathcal{G}_e$ is described in two parts.

1. The initial state is added to the set of reachable states as well as the set of final states, as only final states are considered to initiate further macro steps.
2. For each final state, a macro step is performed by computing the fixed point of the macro-stepping transfer function. The resulting set of reachable states and set of final states are added to the global analysis state.

The fixed point of the global transfer function, $\mathrm{lfp}(\mathit{Global}\text{-}\mathcal{G}_e)$, is a tuple $\langle S, F\rangle$ where S contains all the reachable states of program e under macro-stepping semantics, and F contains all states at which a macro step ended, and is therefore a subset of the set S.

4.1.4. Step 4: Abstraction of the Macro-Stepping Collecting Semantics

The transfer functions described in steps 2 and 3 are concrete, and the computation of their fixed points may not terminate. To ensure termination, the state space used in the definition of the concrete operational semantics (step 1) needs to be abstracted

using the same method as in Chapter 2. The changes to the state space propagate to the definition of the abstract transition relation ($\rightsquigarrow_{\hat{p}}$), and to the definition of the abstract transfer functions (*Macro* - $\widehat{\mathcal{G}}_{s_0,\hat{p}}$ and *Global* - $\widehat{\mathcal{G}}_e$). The computations of the fixed points of the abstracted transfer functions terminate and provide a sound over-approximation of the concrete macro-stepping semantics.

4.2. Properties of a Macro-Stepping Analysis

We discuss here important properties that analyses resulting from the application of MacroConc should exhibit, and explain how this is ensured.

4.2.1. Termination

Termination is ensured by the abstraction performed in step 4 of the MacroConc design method. By abstracting the possibly infinite concrete state space to a finite abstract one, and by demonstrating that the abstract transfer functions are monotone, termination is ensured by Tarski's fixed-point theorem (Tarski, 1955). We provide termination proofs for the MacroConc analyses for λ_α and λ_τ in Sections 4.3.6 and 4.4.5 respectively.

4.2.2. Soundness

As macro-stepping semantics reduces the number of interleavings expressed, it describes *less* information than all-interleavings semantics, which might be seen as problematic for soundness. However, it can be shown that for verifying particular program properties, relying on macro-stepping semantics is equivalent to relying on all-interleavings semantics. Performing a sound abstraction on the macro-stepping semantics therefore results in a sound analysis with respect to these properties.

For example, the application of MacroConc to λ_α and λ_τ is proven sound for local process properties (Theorems 7 and 9): all process states (ς) explored by an all-interleavings analysis are also explored by an analysis featuring macro-stepping. This suffices to derive sound analyses such as, among others, communication topology analyses (Colby, 1995; Martel and Gengler, 2000). Similarly to other state explosion mitigation techniques (Godefroid, 1996), properties expressed over the concurrent state of multiple processes are not expressible in a sound manner anymore without reestablishing all-process interleavings into the semantics. This is for example the case for a may-happen-in-parallel analysis that infers expressions that can be evaluated in parallel on multiple processes (Duesterwald and Soffa, 1991).

4.2.3. Complexity

In the worst case, every macro step degenerates to a single regular step, and the number of interleavings explored by a macro-stepping analysis remains the same as the number of interleavings explored by an all-interleavings analysis. As the number of processes created and of communication effects performed in a concurrent program increases,

the number of interleavings that have to be explored increases exponentially. Analyses resulting from the application of MacroConc therefore have an exponential worst-case time complexity, i.e., $\mathcal{O}(2^{|Exp|})$, like the analyses presented in Chapter 2. However, in practice, the running time of a macro-stepping analysis is orders of magnitude better because most macro steps perform more than one small step, therefore reducing the number of interleavings that the analysis has to explore. However, an increasing number of communication effects in the program under analysis will result in an increased running time. This is because the more communication effects there are, the less transitions can be made within the macro steps, which results in an increase in non-determinism. We demonstrate this in our empirical evaluation (see Section 4.5).

4.2.4. Precision

The precision of an analysis resulting from the application of MacroConc remains the same as the precision of an analysis resulting from the naive application of AAM. This is because an analysis relying on macro-stepping will neither compute *less* information nor *more* information. When proving soundness, one proves that every state that matters for the soundness of an analysis is accounted for by the macro-stepping semantics, and one proves that even though the macro-stepping analysis computes less information than an all-interleavings analysis, all observable information remains identical, hence precision is not improved. Moreover, an analysis relying on macro-stepping semantics cannot produce more information than an analysis relying on all-interleavings semantics, as it only reduces the number of interleavings expressed in the semantics. Hence, the precision of an analysis under macro-stepping semantics is not worse than under all-interleavings semantics. Therefore, precision is preserved in analyses resulting from the application of MacroConc.

4.3. Application of MacroConc to λ_α

We apply the MacroConc design method to λ_α. We start by demonstrating that for an actor model in which the messages in a mailbox are ordered, macro-stepping semantics as originally described by Agha *et al.* (1997) may miss important process interleavings. We therefore instantiate two macro-stepping semantics: *ordered macro-stepping* for actor models with ordered mailboxes, and *unordered macro-stepping* for actor models with unordered mailboxes. Actor models with unordered mailboxes exist in formal models (Agha *et al.*, 1997; Garoche *et al.*, 2006), but are uncommon in real-world implementations of actors, which typically provide some guarantees about the ordering of messages. We then provide the definition of the macro-stepping transfer function and of the global transfer function for λ_α, and their abstractions.

4.3.1. The Importance of Order

Listing 4.1 illustrates how the order of message sends may be impacted by macro-stepping semantics. Lines 2 and 7 each define an actor behavior. The first behavior `beh1`

(line 2) has no state variables, and handles three different messages (lines 3–5). The second behavior `beh2` (line 7) has one state variable, `target`, and upon reception of a message `start` (line 8) sends two messages to this target (lines 9 and 10). The main process then creates actor `t` (line 12) with behavior `beh1`, and actor `a` with behavior `beh2` (line 13), specifying actor `t` as its target. The process then sends message `start` to actor `a` (line 14), followed by message `m3` to actor `t`.

```
1 (define beh1
2   (actor ()
3     (m1 () (become beh1))
4     (m2 () (become beh1))
5     (m3 () (become beh1))))
6 (define beh2
7   (actor (target)
8     (start ()
9       (send target m1)
10      (send target m2)
11      (become beh2 target))))
12 (define t (create beh1))
13 (define a (create beh2 t))
14 (send a start)
15 (send t m3)
```

Listing 4.1: Example program motivating the need for static analyses to revisit macro-stepping for actor models with ordered-message mailboxes.

In a concrete execution, actor `t` can receive messages in its mailbox in any of the following orders.

- `m1`, `m2`, `m3`: actor `a` sends its messages `m1` and `m2`, after which the main process is scheduled for execution and sends message `m3`.
- `m3`, `m1`, `m2`: the main process sends message `m3`, after which actor `a` sends messages `m1` and `m2`.
- `m1`, `m3`, `m2`: actor `a` sends a first message `m1`, the main process is scheduled and sends message `m3`, after which actor `a` sends its second message `m2`.

For a static analysis to be sound for an actor model in which mailboxes preserve the ordering of their messages, it should account for all important interleavings. Consider an analysis that reasons about the order in which messages are received at actor `t`. For such an analysis, a mailbox abstraction preserving ordering information is required, and we study such abstractions in Chapter 5. An analysis abstracting all-interleavings semantics will account for all of these orderings.

An analysis abstracting Agha's original macro-stepping semantics (Agha *et al.*, 1997) however no longer includes the third interleaving in its over-approximation of the program's runtime behavior. This is because the analysis will not interleave the main process with actor `a`'s processing of the `start` message. The original macro-stepping semantics defines a macro step as multiple small steps between the reception of a message and the reception of the next message. Therefore, under such macro-stepping semantics, actor `a` will always send both `m1` and `m2` without interruptions, and the results of an analysis abstracting these semantics will not be sound.

This is why we propose a finer-grained variant of macro-stepping semantics which we call *ordered* macro-stepping semantics. During an ordered macro step, each actor

is allowed to process a message and to send at most a single message. The ordered macro step ends right before a second message is sent, as message sends can introduce other important interleavings. Two ordered macro steps (instead of one regular macro step) are therefore required for actor `a` to process the `start` message. The first macro step ends before actor `a` sends the second message, allowing the main actor to send its message before `a` sends message `m2`. The difference to regular macro-stepping is small, but ensures that analyses relying on such abstractions of ordered macro-stepping semantics correctly account for interleavings at message sends.

This example illustrates that regular macro-stepping semantics, while useful to reduce non-determinism, needs to be adapted for actor models with ordered-message mailboxes. Otherwise, important message interleavings might be discarded, rendering an analysis abstracting macro-stepping semantics unsound. For unordered-message mailboxes, the original notion of macro-stepping suffices because messages can be reordered arbitrarily in the mailbox.

4.3.2. Step 1 of MacroConc for λ_α: Definition of the Operational Semantics

The first step of MacroConc consists of defining the operational semantics of the language. We defined this semantics for λ_α in Chapter 2 in terms of a concurrent transition relation $\rightsquigarrow$, and do not redefine it here as it remains identical. This transition relation includes rules for sequential transitions (Figure 2.23), rules for actor management (Figure 2.24), and rules for sending and processing messages (Figure 2.25).

4.3.3. Step 2 of MacroConc for λ_α: Definition of the Macro-Stepping Transfer Function

Figure 4.4 instantiates our generic formulation of the macro-stepping transfer function from Figure 4.2 for λ_α, as the transfer function $\mathit{Macro}\text{-}\mathcal{G}^{\lambda_\alpha}_{p,\pi_0,\sigma_0,\Xi_0}$. The domain of this transfer function is defined as tuples containing a process map π, a value store σ and a continuation store Ξ, which are defined in Figure 2.19. We discuss the macro-stepping strategy defined by the restriction function f^{λ_α} below.

Instantiation of the Macro-Stepping Transfer Function for λ_α

As the original macro-stepping semantics of Agha *et al.* (1997) does not describe a sound subset of the all-interleavings semantics for actor programs with an ordered mailbox model, we introduce two macro-stepping transfer functions: one that follows the definition of Agha *et al.* (1997), and is suitable for actor models with unordered mailboxes only, and one that is suitable for ordered-mailbox actor models and therefore for λ_α.

Unordered macro-stepping

Using $f^{\lambda_\alpha}(\mathit{eff}) = \{\mathbf{prc}(t, v) \mid t \in \mathit{Tag}, v \in \mathit{Val}\}$ as restriction function in the definition of $\mathit{Macro}\text{-}\mathcal{G}^{\lambda_\alpha}_{p,\varsigma_0,\sigma_0,\Xi_0}$ gives rise to the unordered macro-stepping semantics of

Agha *et al.* (1997). This restriction disallows processing more than one message in the same macro step. A macro step on an actor therefore executes a message handler up to completion, which is called a *turn* in the actor terminology (De Koster *et al.*, 2016).

Ordered macro-stepping

Ordered macro-stepping semantics arises from the definition of f^{λ_α} given below.

$$f^{\lambda_\alpha}(\mathbf{snd}(p,t,v)) = \{\mathbf{prc}(t',v'), \mathbf{snd}(p',t',v') \mid t' \in Tag, p' \in PID, v' \in Val\}$$
$$f^{\lambda_\alpha}(\mathit{eff}) = \{\mathbf{prc}(t,v) \mid t \in Tag, v \in Val\} \text{ if } \mathit{eff} \neq \mathbf{snd}(\_,\_,\_)$$

This restriction disallows actors from processing more than one message and from sending more than one message within the same macro step. Processing or sending a second message requires a further macro step.

λ_α

$$\mathit{Macro}\text{-}\mathcal{G}^{\lambda_\alpha}_{p,\pi_0,\sigma_0,\Xi_0}(\langle S,F,E\rangle) = \langle\{\langle\pi_0,\sigma_0,\Xi_0\rangle\},\varnothing,\varnothing\rangle \tag{1}$$
$$\sqcup \bigsqcup_{\langle\pi,\sigma,\Xi\rangle\in S} \langle\{\langle\pi',\sigma',\Xi'\rangle\},\varnothing,f^{\lambda_\alpha}(\mathit{eff})\rangle \tag{2}$$
$$\pi,\sigma,\Xi \overset{\mathit{eff}}{\rightsquigarrow}_p \pi',\sigma',\Xi' \quad \mathit{eff}\notin E$$
$$\sqcup \bigsqcup_{\langle\pi,\sigma,\Xi\rangle\in S} \langle\{\langle\pi',\sigma',\Xi'\rangle\},\varnothing,\varnothing\rangle \tag{3}$$
$$\pi,\sigma,\Xi \rightsquigarrow_p \pi',\sigma',\Xi'$$
$$\sqcup \bigsqcup_{\langle\pi,\sigma,\Xi\rangle\in S} \langle\varnothing,\{\langle\pi,\sigma,\Xi\rangle\},\varnothing\rangle \tag{4}$$
$$\pi,\sigma,\Xi' \overset{\mathit{eff}}{\rightsquigarrow}_{p,\_,\_,\_} \quad \mathit{eff}\in E$$
$$\sqcup \bigsqcup_{\langle\pi,\sigma,\Xi\rangle\in S} \langle\varnothing,\{\langle\pi,\sigma,\Xi\rangle\},\varnothing\rangle \tag{5}$$
$$\nexists\langle\pi',\sigma',\Xi'\rangle,\pi,\sigma,\Xi \overset{\mathit{eff}}{\rightsquigarrow}_p \pi',\sigma',\Xi'$$

Figure 4.4.: Macro-stepping transfer function for λ_α.

4.3.4. Step 3 of MacroConc for λ_α: Definition of the Global Transfer Function

Figure 4.5 depicts the instantiation of the generic formulation of the global transfer function from Figure 4.3 for λ_α. The initial state s_0 is obtained by injecting the program under analysis with the injection function $\mathcal{I}$ defined in Figure 2.27, and using the initial

value store and continuation store. Identifiers of running processes correspond to the domain of the process map of a state. The global transfer function explores the initial state (1), and any state that can be reached in a macro step from a state of the set of final states F (2).

λ_α

$$\mathit{Global}\text{-}\mathcal{G}_e^{\lambda_\alpha}(\langle S, F\rangle) = \langle \{s_0\}, \{s_0\}\rangle \quad (1)$$

$$\sqcup \bigsqcup_{\substack{\langle \pi,\sigma,\Xi\rangle \in F \\ p \in \mathrm{dom}(\pi) \\ \langle S',F',\_\rangle = \mathrm{lfp}(\mathit{Macro}\text{-}\mathcal{G}^{\lambda_\alpha}_{p,\pi,\sigma,\Xi})}} \langle S', F'\rangle \quad (2)$$

$$\text{where } s_0 = \langle \mathcal{I}(e), [], [k_0 \mapsto \epsilon]\rangle$$

Figure 4.5.: Global transfer function for λ_α.

4.3.5. Step 4 of MacroConc for λ_α: Abstraction of the Macro-Stepping Collecting Semantics

The transfer functions are abstracted by incorporating the abstract state space and the abstract transition relation defined in Chapter 2 instead of the concrete ones. The abstract transfer functions $\mathit{Macro}\text{-}\widehat{\mathcal{G}}^{\lambda_\alpha}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\hat{\Xi}_0} : \mathcal{P}(\widehat{\Sigma}) \times \mathcal{P}(\widehat{\Sigma}) \times \mathcal{P}(\widehat{\mathit{Effect}}) \to \mathcal{P}(\widehat{\Sigma}) \times \mathcal{P}(\widehat{\Sigma}) \times \mathcal{P}(\widehat{\mathit{Effect}})$ and $\mathit{Global}\text{-}\widehat{\mathcal{G}}^{\lambda_\alpha}_e : \mathcal{P}(\widehat{\Sigma}) \times \mathcal{P}(\widehat{\Sigma}) \to \mathcal{P}(\widehat{\Sigma}) \times \mathcal{P}(\widehat{\Sigma})$ preserve the same structure as the concrete ones from Figures 4.4 and 4.5. We therefore do not provide their definition for the sake of brevity.

4.3.6. Soundness and Termination

Theorems 7 and 8 state that the analysis described by the fixed point of the abstract global transfer function is sound and terminates, two crucial properties for a static analysis.

Theorem 7 (Soundness)**.** *Static analyses for a concurrent actor program that soundly abstract macro-stepping semantics are sound with respect to local process properties.*

Proof. The proof is detailed in Appendix B.2.1. The idea of this proof is the following. We introduce a function $\mathit{local} : \mathcal{P}(\Pi \times \mathit{Store} \times \mathit{KStore}) \to (\mathit{PID} \to \mathcal{P}(\Sigma \times \mathit{Store} \times \mathit{KStore}))$, which defines the meaning of a *local process property*. A local process property is a property that can be expressed using the result of this function when applied to the fixed point of the global transfer function, i.e., a property that only concerns the states of processes in isolation. We show that the local process view of all states reachable under all-interleavings semantics is equivalent to the local process view of states reachable using macro-stepping, and that the restriction function for ordered macro-stepping allows at most a single transition that act on the global state of the analysis per macro

step. Performing a sound abstraction of the concrete macro-stepping semantics results in a sound static analysis for local process properties. □

Theorem 8 (Termination)**.** *The computation of* $\mathrm{lfp}(\mathit{Global}\text{-}\widehat{\mathcal{G}}_e^{\lambda_\alpha})$ *always terminates.*

Proof. The proof is detailed in Appendix B.2.1 and follows the same structure as previous termination proofs. By proving that the abstract state space is finite, and that both the abstract macro-stepping transfer function $\mathit{Macro}\text{-}\widehat{\mathcal{G}}^{\lambda_\alpha}_{\hat{\pi}_0,\hat{\sigma}_0,\hat{\Xi}_0,\hat{p}}$ and the global transfer function $\mathit{Global}\text{-}\widehat{\mathcal{G}}_e^{\lambda_\alpha}$ are monotone, termination is ensured by Tarski's fixed-point theorem (Tarski, 1955). □

4.4. Application of MacroConc to λ_τ

We apply the MacroConc design method to the λ_τ language introduced in Section 2.3. We present this application in the same way as for λ_α: we provide the definition of the operational semantics, the definition of the macro-stepping transfer function, the definition of the global transfer function and the abstractions of these transfer functions. Applying the macro-stepping analysis design method to λ_τ results in exactly the same formulation of the transfer functions as for λ_α.

4.4.1. Step 1 of MacroConc for λ_α: Definition of the Operational Semantics

We defined the semantics for λ_τ in Chapter 2 in terms of a concurrent transition relation $\rightsquigarrow$, and do not redefine it here as it remains identical. This transition relation includes rules for sequential transitions (Figure 2.45), rules for thread creation and joining (Figure 2.46), rules for creating, accessing and modifying references (Figure 2.47), and rules for creating, acquiring and releasing locks (Figure 2.48).

4.4.2. Step 2 of MacroConc for λ_τ: Definition of the Macro-Stepping Transfer Function

Figure 4.6 instantiates the generic macro-stepping transfer function from Figure 4.2 for λ_τ programs, as the function $\mathit{Macro}\text{-}\mathcal{G}^{\lambda_\tau}_{p,\pi_0,\sigma_0,\Xi_0}$. The domain of this transfer function is defined as tuples containing a process map π, a value store σ and a continuation store Ξ, which are defined in Figure 2.43. It remains structurally equivalent to the generic formulation and only differs in the definition of the effect restriction function f^{λ_τ}.

Instantiation of the Macro-Stepping Transfer Function for λ_τ

The effect restriction function for λ_τ has to break macro steps when possibly interfering effects are performed. A straightforward yet effective definition of this restriction function is to only allow for at most one of the following operations in a macro step: spawning a thread, reading from a reference, writing to a reference, acquiring a lock, releasing a lock, or joining another process. This results in defining the restriction function as

$f^{\lambda_\tau}(\mathit{eff}) = \mathit{Effect}$, meaning that all possible effects are forbidden once a communication effect has been performed (*Effect* is the set of all effects). As shown in our evaluation (Section 4.5), this restriction function results in an analysis that performs well both in terms of running time and precision. Refinements of this restriction function include allowing a thread to spawn multiple threads or taking into account the addresses read from or written to. We discuss possible refinements of this restriction function as future work in Chapter 7.

λ_τ

$$\mathit{Macro}\text{-}\mathcal{G}^{\lambda_\tau}_{p,\pi_0,\sigma_0,\Xi_0}(\langle S, F, E\rangle) = \langle\{\langle\pi_0,\sigma_0,\Xi_0\rangle\},\varnothing,\varnothing\rangle \tag{1}$$

$$\sqcup \bigsqcup_{\langle\pi,\sigma,\Xi\rangle\in S} \langle\{\langle\pi',\sigma',\Xi'\rangle\},\varnothing,f^{\lambda_\tau}(\mathit{eff})\rangle \tag{2}$$
$$\pi,\sigma,\Xi \overset{\mathit{eff}}{\rightsquigarrow}_p \pi',\sigma',\Xi',\ \mathit{eff}\notin E$$

$$\sqcup \bigsqcup_{\langle\pi,\sigma,\Xi\rangle\in S} \langle\{\langle\pi',\sigma',\Xi'\rangle\},\varnothing,\varnothing\rangle \tag{3}$$
$$\pi,\sigma,\Xi \rightsquigarrow_p \pi',\sigma',\Xi'$$

$$\sqcup \bigsqcup_{\langle\pi,\sigma,\Xi\rangle\in S} \langle\varnothing,\{\langle\pi,\sigma,\Xi\rangle\},\varnothing\rangle \tag{4}$$
$$\pi,\sigma,\Xi' \overset{\mathit{eff}}{\rightsquigarrow}_{p\_,\_,\_},\ \mathit{eff}\in E$$

$$\sqcup \bigsqcup_{\langle\pi,\sigma,\Xi\rangle\in S} \langle\varnothing,\{\langle\pi,\sigma,\Xi\rangle\},\varnothing\rangle \tag{5}$$
$$\nexists\langle\pi',\sigma',\Xi'\rangle,\pi,\sigma,\Xi \overset{\mathit{eff}}{\rightsquigarrow}_p \pi',\sigma',\Xi'$$

Figure 4.6.: Macro-stepping transfer function for λ_τ.

4.4.3. Step 3 of MacroConc for λ_τ: Definition of the Global Transfer Function

Figure 4.7 instantiates the generic global transfer function from Figure 4.3 for λ_τ programs. The initial state s_0 is obtained by injecting the program to analyze with the injection function $\mathcal{I}$ defined in Figure 2.49, and the set of running processes can be extracted from the domain of the process map ($\mathrm{dom}(\pi)$). It preserves the same structure as the generic formulation.

λ_τ

$$\textit{Global}\text{-}\mathcal{G}_e^{\lambda_\tau}(\langle S, F\rangle) = \langle\{s_0\}, \{s_0\}\rangle \sqcup \bigsqcup_{\substack{\langle\pi,\sigma,\Xi\rangle\in F \\ p\in\mathrm{dom}(\pi) \\ \langle S',F',\_\rangle=\mathrm{lfp}(\textit{Macro}\text{-}\mathcal{G}^{\lambda_\tau}_{p,\pi,\sigma,\Xi})}} \langle S', F'\rangle$$
$$\text{where } s_0 = \langle \mathcal{I}(e), [], [k_0 \mapsto \epsilon]\rangle$$

Figure 4.7.: Global transfer function for λ_τ.

4.4.4. Step 4 of MacroConc for λ_τ: Abstraction of the Macro-Stepping Collecting Semantics

The transfer functions of the macro-stepping collecting semantics for λ_τ are abstracted analogously to those for λ_α, by plugging in the abstract domains in place of the concrete ones, without requiring structural modifications. The abstract macro-stepping transfer function $\textit{Macro}\text{-}\widehat{\mathcal{G}}^{\lambda_\tau}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0} : \mathcal{P}(\widehat{\Sigma}) \times \mathcal{P}(\widehat{\Sigma}) \times \mathcal{P}(\widehat{\textit{Effect}}) \to \mathcal{P}(\widehat{\Sigma}) \times \mathcal{P}(\widehat{\textit{Effect}})$ therefore acts on sets of abstract states and on sets abstract effects, and the abstract global transfer function $\textit{Global}\text{-}\widehat{\mathcal{G}}^{\lambda_\tau}_e : \mathcal{P}(\widehat{\Sigma}) \times \mathcal{P}(\widehat{\Sigma}) \to \mathcal{P}(\widehat{\Sigma}) \times \mathcal{P}(\widehat{\Sigma})$ acts on sets of abstract states. We do not provide these abstract definitions for the sake of brevity, as again, only hats need to be placed on the concrete elements to perform abstraction.

4.4.5. Soundness and Termination

Theorems 9 and 10 state that the analysis described by the fixed point of the abstract global transfer function is sound and terminates, two crucial properties for a static analysis.

Theorem 9 (Soundness)**.** *Static analyses for a multi-threaded program that soundly abstract macro-stepping semantics are sound with respect to local process properties.*

Proof. The proof is detailed in Appendix B.2.2 and follows the same reasoning as the proof for Theorem 7: we show the equivalence of macro-stepping semantics to all-interleavings semantics in the concrete case for local process properties, and a further sound abstraction results in a sound static analysis for local process properties. □

Theorem 10 (Termination)**.** *The computation of* $\mathrm{lfp}(\textit{Global}\text{-}\widehat{\mathcal{G}}^{\lambda_\tau}_e)$ *always terminates.*

Proof. The proof is detailed in Appendix B.2.2 and follows the same structure as the proof for Theorem 8: both transfer functions are monotone and act on finite state spaces, hence the analysis always terminates. □

4.5. Soundness Testing and Evaluation of Running Time, Precision, and Scalability on a Benchmark Suite

We implemented the analyses resulting from the application of MacroConc described in this chapter using our Scala-AM static analysis framework, presented in Section 2.4.1. We empirically evaluate these analyses in terms of running time, precision, and scalability on the set of benchmark programs introduced in Section 2.4.2.

4.5.1. Soundness Testing

We have proven that the application of MacroConc to λ_α and λ_τ results in sound analyses (Theorems 7 and 9). We also provide empirical evidence for the soundness of our implementation through soundness testing (Andreasen *et al.*, 2017) as described in Section 2.4.3. In brief, we compare the information inferred by each analysis to the information recorded during 1000 concrete runs of each benchmark, and check that all concrete information is soundly over-approximated by the analysis. No unsound results were discovered.

4.5.2. Running Times

We report the average time required to analyze each benchmark program with the same setup used in the evaluation of Section 2.4: each program is analyzed 20 times after 10 warmup runs, with a time budget of 30 minutes. The results are given in Table 4.1. Of the 56 benchmark programs, 30 are analyzed within the given time budget of 30 minutes, with analysis times varying between 15 milliseconds and 25 minutes.

Actors				Threads			
Bench.	Time (ms)	Bench.	Time (ms)	Bench.	Time (ms)	Bench. (ms)	Time (ms)
PP	494	BTX	2567	ABP	1014	TRAPR	559
COUNT	284	RSORT	12913	COUNT	48003	ATOMS	8710
FJT	112	FBANK	619849	DEKKER	275	STM	∞
FJC	15	SIEVE	79	FACT	∞	NBODY	∞
THR	233744	UCT	∞	MATMUL	∞	SIEVE	1026
CHAM	1499941	OFL	∞	MCARLO	1134673	CRYPT	∞
BIG	∞	TRAPR	10301	MSORT	∞	MCEVAL	∞
CDICT	21246	PIPREC	387	PC	12852	QSORT	∞
CSLL	∞	RMM	∞	PHIL	463	TSP	∞
PCBB	99630	QSORT	∞	PHILD	5168	BCHAIN	74711
PHIL	∞	APSP	∞	PP	657	LIFE	663712
SBAR	∞	SOR	∞	RINGBUF	∞	PPS	56669
CIG	1337	ASTAR	∞	RNG	711	MINIMAX	∞
LOGM	∞	NQN	∞	SUDOKU	∞	ACTORS	∞

Table 4.1.: Running times of the analyses presented in this chapter on our benchmark programs. The timings are expressed in milliseconds and represent the average of 20 runs after 10 warmup runs. ∞ denotes that the analysis exceeded the allocated time budget of 30 minutes.

We observe that the running time of the analysis correlates with the following characteristics of the program under analysis.

- When each abstract process is mapped to a single concrete process, because each creation site is executed a single time, the analysis time is low. This is the case for the PP and COUNT actor programs, and for the ABP, DEKKER and PP multi-threaded programs, which are all analyzed in a second or less. This is because when each abstract process is mapped to a single concrete process, the use of abstract counting reduces the non-determinism on the process map π of the analysis.

- For actor programs, the fewer messages are sent in a message handler, the lower the analysis time. For example, programs FJT and FJC contain actors that receive messages but do not send any message, and programs BTX, TRAPR and PIPREC contain actors that send at most one message per message handler. These benchmarks are all analyzed in 10 seconds or less.

  If more than one message are sent from a message handler, the macro step will have to be broken before the end of the actor's turn. The RMM program illustrates this well: a message handler of one of the actors sends more than 8 messages upon the reception of a single message. To analyze this message handler, more than 8 macro steps are therefore required.

- For multi-threaded programs, the fewer threads perform joins, the lower the analysis time. For example, programs ABP, COUNT, DEKKER, PC, PHIL, PP, and RNG all contain calls to `join` only within the main thread, never in a spawned thread, and are analyzed in under a minute. As more threads perform join operations, more macro steps are broken and more non-determinism is introduced, resulting in an increased running time for the analysis.

- For the multi-threaded programs, the fewer potential conflicts through access to shared-memory there are, the lower the analysis time. For example, programs PHIL and TRAPR do not make use of shared-memory constructs and are analyzed in less than a second. Program PHILD program is similar to the PHIL program, but uses shared memory and takes ten times the time required for the PHIL program. As more potential conflicts are introduced, more macro steps have to be broken and analysis time increases.

We confirm that all these factors have an impact on the analysis time in Section 4.5.4. Another factor—which we do not detail further as it is orthogonal to our discussion—is the sequential complexity of the program under analysis. For example, while the MCEVAL program does not use many concurrent features (only `spawn` and `join`), it is a meta-circular interpreter that manipulates cons cells and relies on higher-order functions such as `map`, and therefore requires more time to analyze than a program like RNG that, even though it accesses and modifies shared state, performs simple arithmetic computations.

Comparison to Naive Applications of AAM

We observe a substantial improvement in running time over the analyses presented in Chapter 2, which can only analyze 6 of the 56 benchmark programs within the time budget. Table 4.2 compares running times for benchmarks that can be analyzed within the time budget by both analyses. The application of MacroConc rather than a naive application of AAM results in analyses that are up to four orders of magnitude faster, with a speedup factor (i.e., the running time of analyses presented in Chapter 2 over the running time of analyses resulting from the application of MacroConc) ranging from 3 to 35216. The speedup is more important for actor programs that contain actors of which message handlers do not send messages, such as programs FJT and FJC.

Actors				Threads			
Bench.	Naive	Macro	Speedup	Bench.	Naive	Macro	Speedup
PP	2876	494	÷5.82	ABP	92163	1014	÷90.89
COUNT	920	284	÷3.24	DEKKER	20675	275	÷75.18
FJT	435812	112	÷3891.18				
FJC	528242	15	÷35216.13				

Table 4.2.: Running time comparison between the analyses resulting from the naive application of AAM (column *Naive*) and the analyses resulting from the application of MacroConc presented in this chapter (column *Macro*), for programs analyzed by both analyses in under 30 minutes. Columns *Naive* and *Macro* contain timings in milliseconds. Columns *Speedup* represent the speedup factor of the running time of analyses resulting from the application of AAM over the running time of analyses resulting from the application of MacroConc.

Comparison to Related Work

We also compare the analysis for actors presented in this chapter against its closest related work, the Soter tool (D'Osualdo *et al.*, 2012). Soter performs verification of actor programs written in Erlang in two phases: a first phase constructs a model of the program under analysis, and a second phase performs model checking using this model and user-provided annotations to verify that the properties expressed in the annotations hold for every execution of the program. We only compare to the running times of the first phase. To perform the comparison, we translated each of the actor programs from our benchmark suite presented in Section 2.4.2 faithfully from λ_α to Erlang. As Soter is closed source and only available through a web interface with a two minute time limit, the resulting running times are not directly comparable to the running times of our analysis. We therefore do not compare the approaches in terms of raw timings, but rather in terms of the number of benchmarks that can be analyzed within the time limit.

The results are given in Table 4.3, where we cap the running times of our analysis at two minutes to match Soter's time budget. We observe that both analyses share similar results: Soter analyzes 11 out of the 28 benchmarks under the two-minutes timeout, while our macro-stepping analysis analyzes 12 out of the 28 benchmarks within the

Bench.	Macro	Soter	Bench.	Macro	Soter	Bench.	Macro	Soter	Bench.	Macro	Soter
PP	494	150	CDICT	21246	∞	BTX	2567	∞	PIPREC	387	∞
COUNT	284	310	CSLL	∞	∞	RSORT	12913	∞	RMM	∞	∞
FJT	112	470	PCBB	99630	∞	FBANK	∞	∞	QSORT	∞	∞
FJC	15	110	PHIL	∞	29840	SIEVE	79	4240	APSP	∞	∞
THR	∞	1130	SBAR	∞	1710	UCT	∞	∞	SOR	∞	∞
CHAM	∞	1860	CIG	1337	∞	OFL	∞	∞	ASTAR	∞	∞
BIG	∞	∞	LOGM	∞	31950	TRAPR	10301	∞	NQN	∞	29090

Table 4.3.: Comparison between the analysis for concurrent actors presented in this chapter (column *Macro*) and the analysis for concurrent actors present in Soter (D'Osualdo *et al.*, 2012) (column *Soter*). Each benchmark program is analyzed with a time limit of 2 minutes. Running times are given in milliseconds.

same time budget. We also see that 5 benchmark programs can be analyzed by both approaches within the time budget, that 6 benchmarks can be analyzed within the time budget only by Soter while our approach times out (THR, CHAM, PHIL, SBAR, CIG, NQN), and that Soter times out on 7 benchmarks that are analyzed by our approach in under two minutes (CDICT, PCBB, CIG, LOGM, BTX, OFL, TRAPR).

4.5.3. Precision

As explained in Section 4.2.4, the precision of analyses resulting from the application of MacroConc remains the same as the analyses presented in Chapter 2. We did not discuss precision in Chapter 2 as only 6 benchmarks can be analyzed within the time budget with the analyses presented in that chapter. The precision of the analyses resulting from the application of MacroConc being the same, we discuss precision of all analyses together here.

To measure precision, we compare the information inferred from running the analyses presented in this chapter on each benchmark program with the corresponding concrete information resulting from running each benchmark 1000 times, first aggregating and then abstracting the observed results. The abstracted aggregated results from the concrete runs represent an under-approximation of the maximally precise analysis results. This is an under-approximation because we cannot ensure that all possible program paths and process interleavings have been explored in the 1000 different runs. However, we expect the abstraction of the aggregated concrete results to be close to the results that a maximally precise analysis would compute. By comparing these abstracted aggregated results to the information computed by a static analysis, we can quantify the precision of the analysis by counting potential *spurious* abstract elements. Resulting from over-approximation, a spurious abstract element lacks corresponding concrete elements in actual runs of the program. The more spurious elements, the less precise the results of the analysis.

We record the following elements at each run of the λ_α programs:

1. each execution of a `become` statement, with the arguments given,
2. each creation of an actor, with the state given as argument, and

3. each reception of a message, with the tag and arguments of the message.

We record the following elements at each run of the λ_τ programs:

1. each value returned by each thread,
2. each memory address read from and written to by each thread,
3. each lock acquired and released by each thread.

We compare the observed concrete elements with the results of a communication effects analysis in order to detect spurious elements. We sum the number of spurious elements computed by the analysis for each benchmark and report on the results in Table 4.4. Note that these results are an upper bound on the number of spurious elements, as it might be the case that none of the 1000 runs of each benchmark program explored a specific path that leads to one of the element detected as spurious, which therefore is not spurious. We nonetheless carefully manually inspected the recorded elements and could not find missing elements that could arise in other runs of the programs.

Actors						Threads					
Bench.	Obs.	Spu.	Bench.	Obs.	Spu.	Bench.	Obs.	Spu.	Bench.	Obs.	Spu.
PP	9	0	BTX	9	0	ABP	20	0	TRAPR	2	0
COUNT	8	0	RSORT	10	0	COUNT	6	4	ATOMS	6	0
FJT	3	0	FBANK	38	0	DEKKER	16	5	STM	11	–
FJC	2	0	SIEVE	8	0	FACT	21	–	NBODY	25	–
THR	5	2	UCT	20	–	MATMUL	40	–	SIEVE	4	2
CHAM	10	0	OFL	13	–	MCARLO	12	0	CRYPT	2	–
BIG	10	–	TRAPR	7	0	MSORT	12	–	MCEVAL	2	–
CDICT	13	0	PIPREC	8	0	PC	15	0	QSORT	18	–
CSLL	16	–	RMM	14	–	PHIL	4	0	TSP	12	–
PCBB	13	0	QSORT	6	–	PHILD	8	0	BCHAIN	7	0
PHIL	13	–	APSP	5	–	PP	6	0	LIFE	16	0
SBAR	19	–	SOR	12	–	RINGBUF	19	–	PPS	10	0
CIG	9	0	ASTAR	11	–	RNG	6	0	MINIMAX	4	–
LOGM	15	–	NQN	11	–	SUDOKU	58	–	ACTORS	18	–

Table 4.4.: Precision evaluation. Column *Obs.* lists the number of observed elements among 1000 concrete runs of each benchmark program. Column *Spu.* lists the number of potential spurious elements inferred by an analysis, i.e., abstract elements that have no corresponding element observed in any concrete run. A dash (–) is used to denote benchmarks for which the analysis exceeds the time budget of 30 minutes.

We see that the analyses achieve full precision on most benchmarks: out of the 30 benchmark programs analyzed within the time limit, 26 are analyzed with full precision. On the 4 other benchmarks, a few potential spurious elements are detected by our evaluation method. In total, on the programs analyzed within the time budget, 13 potential spurious elements have been reported, and 290 elements observed at run time have been correctly inferred by the analysis. This results in a precision of 96%[1]. We

[1] We have 290 observed elements (true positives) and 13 spurious elements (false positives), therefore the precision is $\frac{290}{290+13} = 0.96$.

therefore conclude that an analysis resulting from the application of MacroConc yields a very high precision on our benchmark suite.

4.5.4. Scalability

We identified the worst-case time complexity of analyses resulting from the application of MacroConc as exponential in the number of running processes and communication effects (Section 4.2.3). We also evaluate the scalability of the analyses resulting from the application of MacroConc empirically. To that end, we generate a number of synthetic benchmark programs of increasing complexity for which we record the running time of a macro-stepping analysis. Each benchmark family studied here has an increasing number of created processes or communication effects, and demonstrate how processes created and communication effects performed impact the running time of the analyses presented in this chapter. In each benchmark family, we ensured that the number of expression remains constant to measure precisely the overhead incurred on analysis time by the increasing value of the parameter. To that end, some benchmark families include dummy function calls to the + primitive, that are replaced to concurrent primitives (e.g., `spawn`) when the value of the parameter increases.

For λ_α, a first family of benchmark programs depicted in Figure 4.8 creates a single actor with a fixed number of message handlers (10), and sends the same number of messages to that actor. The varying parameter for this benchmark family is the number of different messages being sent, varying from a single message (the actor receives 10 times the same message), to all different messages (the actor receives 10 different messages). A second family of benchmark programs depicted in Figure 4.9 creates a fixed number of actors (7), with a number of different behaviors varying from a single behavior (all 7 actors have the same behavior) to 7 different behaviors (all actors have a different behavior). Due to the process allocation strategy presented in Figure 2.26, multiple actors that are created with the same behavior are mapped to a single abstract actor, while actors that are created with different behaviors are mapped to different abstract actors.

For λ_τ, a first family of benchmark programs depicted in Figure 4.10 contains a fixed number of nested `let` bindings (10), and gradually evaluates the expression for the bound value in a separate thread. We run each of the programs from this benchmark family from 1 thread created (one expression is evaluated in a new thread, the others are evaluated sequentially) to 10 threads created (all bound values are evaluated in a new thread). A second family of benchmark programs depicted in Figure 4.11 spawns a number of threads (12), each either evaluating an expression sequentially, or joining the result of another thread. We run each of the programs from this benchmark family from 1 joins (all but one threads evaluate an arithmetic expression) to 9 joins (9 threads perform joins). A third family of benchmark programs depicted in Figure 4.12 spawns a number of threads (10), each evaluating a simple expression or incrementing a shared mutable reference. We run each of the programs from this benchmark family from 0 conflicting threads (all but one threads evaluate an arithmetic expression) to 10 conflicting threads (all threads perform unprotected read and write accesses to the same mutable reference).

```
(letrec ((b (actor ()
              (m0 () (become b))
              ...
              (m9 () (become b))))
         (p (create b)))
  (send p m0)
  ...
  (send p m0))
```

(a) $m = 1$

```
(letrec ((b (actor ()
              (m0 () (become b))
              ...
              (m9 () (become b))))
         (p (create b)))
  (send p m0)
  ...
  (send p m9))
```

(b) $m = 10$

Figure 4.8.: Benchmark family for number of message (m) in λ_α

```
(letrec ((b0 (actor () (m () (become b0))))
         ...
         (b6 (actor () (m () (become b6)))))
  (send (create b0) m)
  ...
  (send (create b0) m))
```

(a) $b = 1$

```
(letrec ((b0 (actor () (m () (become b0))))
         ...
         (b6 (actor () (m () (become b6)))))
  (send (create b0) m)
  ...
  (send (create b6) m))
```

(b) $b = 7$

Figure 4.9.: Benchmark family for number of behaviors (b) in λ_α

```
(letrec ((t0 (spawn 0))
         (t1 (id 1))
         ...
         (t9 (id 9)))
  1)
```

(a) $t = 0$

```
(letrec ((t0 (spawn 0))
         (t1 (spawn 1))
         ...
         (t9 (spawn 9)))
  1)
```

(b) $t = 10$

Figure 4.10.: Benchmark family for number of threads (t) in λ_τ

```
(letrec ((t0 (spawn (+ (join t1) (join t2))))
         (t1 (spawn (+ (id 2) (id 3))))
         ...
         (t9 (spawn (+ (id 10) (id 11))))
         (t10 (spawn (id 11)))
         (t11 (spawn (id 12)))))
  1)
```

(a) $j = 1$

```
(letrec ((t0 (spawn (+ (join t1) (join t2))))
         (t1 (spawn (+ (join t2) (join t3))))
         ...
         (t9 (spawn (+ (join t10) (join t11))))
         (t10 (spawn (id 11)))
         (t11 (spawn (id 12))))
  1)
```

(b) $j = 9$

Figure 4.11.: Benchmark family for number of joins (j) in λ_τ

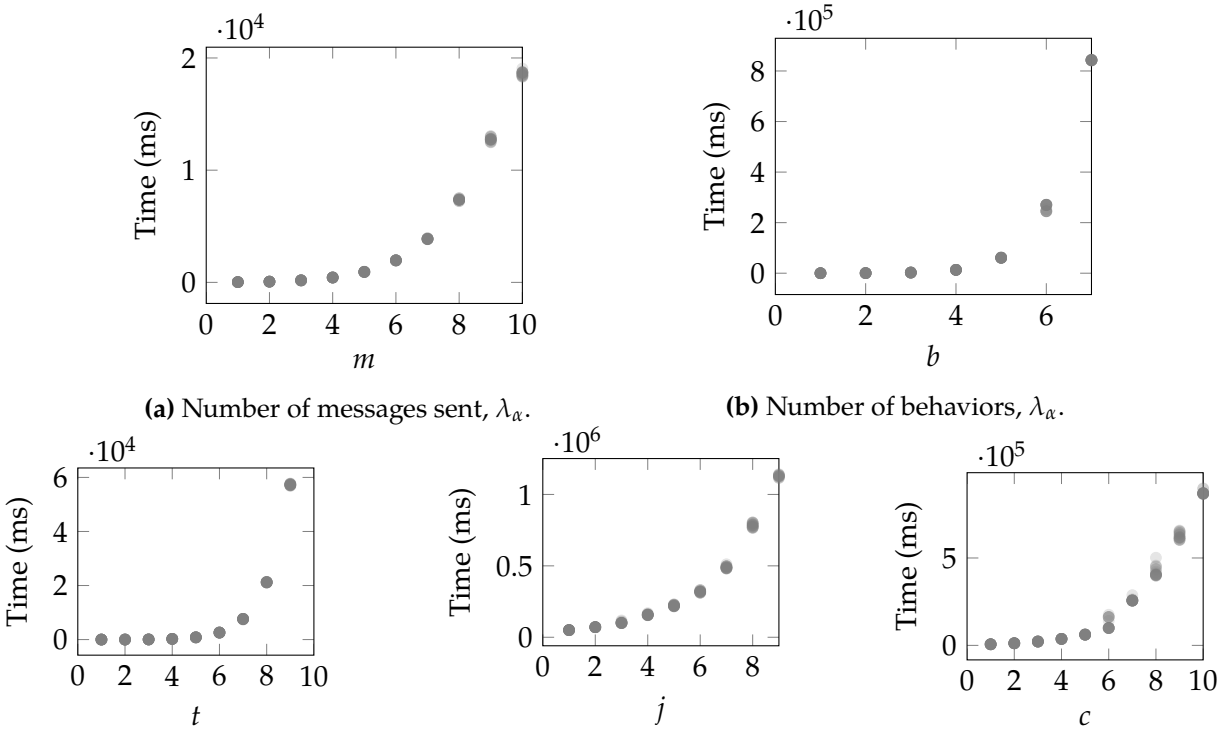
```
(letrec ((x (ref 0))
         (t0 (spawn (ref-set! x
                      (id (deref x 1)))))
         (t1 (spawn (id (id (id 1) 1))))
         ...
         (t9 (spawn (id (id (id 9) 1)))))
  (join t0)
  ...
  (join t9))
```

(a) $c = 1$

```
(letrec ((x (ref 0))
         (t0 (spawn (ref-set! x (+ (deref x 1)))))
         (t1 (spawn (ref-set! x (+ (deref x 1)))))
         ...
         (t9 (spawn (ref-set! x (+ (deref x 1))))))
  (join t0)
  ...
  (join t9))
```

(b) $c = 10$

Figure 4.12.: Benchmark family for number of conflicts (c) in λ_τ



(a) Number of messages sent, λ_α.

(b) Number of behaviors, λ_α.

(c) Number of threads, λ_τ.

(d) Number of joins, λ_τ.

(e) Number of conflicts, λ_τ.

Figure 4.13.: Scalability evaluation of MacroConc. Each graph corresponds to a benchmark which is parameterized by a value (m for number of messages, and b for number of actor behaviors, t for abstract threads, j for join operations, c for conflicts) that increases, and shows the running time of the analysis in function of the value of the given parameter.

For each of the λ_α and λ_τ families of benchmarks, we expect to see an exponential increase in the running time of the analysis as the value of the parameter (messages, behaviors, threads, joins, conflicts) increases. This is because as more communication effects are performed, the macro steps become smaller, and the non-determinism of the analysis increases. We ran each benchmark 20 times after 10 warmup runs, and recorded the timing of each of the 20 runs. We plot the results as one data point per run in Figure 4.13, where we see an exponential trend for each family of benchmarks. Although we note minor variations of the data points, they do not deviate from the exponential curve. This confirms the theoretical results from Section 4.2.3.

4.6. Conclusion

In this chapter, we presented the MacroConc analysis design method that, when applied to operational semantics of a concurrent programming language, results in a static analysis featuring an improved scalability compared to analyses resulting from a naive

application of the AAM design method—thanks to macro-stepping. Analyzing parts of the execution of the processes sequentially, as macro steps, reduces the number of process interleavings that have to be explored. The macro-stepping strategy is defined by a restriction function expressing which communication effects are not allowed once specific communication effects have been generated by a process. Any transition that performs a communication effect that is disallowed interrupts the macro step at which point the analysis investigates other interleavings. The result of a macro step of a process from a given state is expressed as the fixed point of a macro-stepping transfer function expressed in terms of the aforementioned restriction function. The fixed-point computations of the macro-stepping transfer function are driven by a global transfer function that dictates which macro step to explore based on the running processes and the final states of previously explored macro steps.

We have formally proven that the analyses resulting from the application of MacroConc to λ_α and to λ_τ are sound, terminate and exhibit the same precision as the analyses presented in Chapter 2, while exploring potentially fewer interleavings. However, their worst-case time complexity remains exponential. We empirically evaluate the resulting analyses through their implementation on top of our static analysis framework. These analyses scale significantly better than the analyses presented in Chapter 2, analyzing 30 out of the 56 benchmarks within the time budget, compared to only 6 for the analyses presented in Chapter 2. In terms of running time, analyses resulting from the application of MacroConc provide a speedup of up to four orders of magnitude. We compare the analysis for λ_α to its closest related work (D'Osualdo *et al.*, 2012), showing that it behaves similarly: our analysis analyzes 12 out of the 28 actor benchmarks under 2 minutes, Soter analyzes 11 of them. In terms of precision, the analyses resulting from the application of MacroConc achieve a high precision of 96% on our benchmark suite when considering communication effects inferred by the analyses. However, the scalability of the resulting analyses remains limited, as demonstrated empirically through a set of synthetic benchmarks. As the number of communication effects in the program under analysis increases, analysis time increases exponentially.

This motivates the need for analyses that over-approximate the interleavings instead of reducing the number of interleavings to analyze, as in the worst case the number of interleavings remains exponential.

5

A STUDY OF MAILBOX ABSTRACTIONS

The analyses for concurrent actor programs presented in Chapters 2 and 4 abstract the mailboxes of actors to sets. However, this abstraction is not sufficiently expressive for verification of specific properties that require information about the ordering or the multiplicity of messages within a mailbox. Sets do not preserve information about the ordering nor about the multiplicity of their elements. Preserving such information in the abstraction of mailboxes enables the verification of more properties and may increase the precision of existing analyses, as less over-approximations are performed.

In this chapter, we study the impact of different mailbox abstractions on the precision and running time of the analysis for concurrent actor programs presented in Chapter 4. We first justify the importance of having information about the ordering and multiplicity of messages in the abstraction of mailboxes (Section 5.1). We then present five mailbox abstractions, of which we prove the soundness. We categorize these abstractions according to whether they preserve the ordering of messages and according to whether they preserve the multiplicity of messages (Section 5.2). We finally evaluate the impact of these mailbox abstractions on the running time and precision of actor analyses (Section 5.3).

5.1. The Importance of Ordering and Multiplicity

The static analyses introduced in Chapters 2 and 4 abstract mailboxes to sets of abstract messages by default. However, the abstract mailbox domain $\widehat{Mbox}$ is a tunable parameter of the analysis, of which we study possible instantiations in this chapter. We first emphasize the need to study mailbox abstractions, as they have an impact on the precision of analyses for concurrent actor programs. We claim that mailbox abstractions have to be chosen carefully, as illustrated by the following examples.

5.1.1. Verifying Absence of Errors

Listing 5.1 is adapted from Agha (1986) and uses an actor with behavior `stack-node` to implement a stack. Upon receiving a `push` message with a value `v` to be pushed on the stack (line 3), the actor creates a closure capable of restoring its current state (line 5), i.e., the values of `content` and `link`. The actor updates its state variable `content` to the pushed value and updates its state variable `link` to the closure, by becoming the same behavior with the updated state variables. Upon receiving a `pop` message (line 8), the value of `content` is sent to the provided target actor `customer`, and the `link` closure is called to restore the previous state. Should the stack be empty upon a pop (i.e., `link` is `#f`), a stack underflow error is raised (line 15). The main process pushes a value obtained from the user on a stack `act` (line 19), pops one value from this stack (line 20), which will send it (line 11) to a `display` actor (omitted from the example, passed along on line 20) that will print the value received. Although this program contains an error statement on line 15, this error is not reachable in any execution of the program under any input and any interleaving.

```
1 (define stack-node
2   (actor (content link)
3     (push (v)
4       (become stack-node v
5         (lambda ()
6           (become stack-node
7             content link))))
8     (pop (customer)
9       (if link
10        (begin
11          (send customer
12            message content)
13          (link))
14        (begin
15          (error "underflow")
16          (terminate))))))
17 (define display (create display-actor))
18 (define act (create stack-node #f #f))
19 (send act push (read-integer))
20 (send act pop display)
```

Listing 5.1: Actor-based stack implementation adapted from Agha (1986).

No multiplicity information.

If mailboxes are abstracted to sets, as done in Chapters 2 and 4 and in related work (D'Osualdo *et al.*, 2013), this error is deemed reachable. Executing lines 19–20 results in the mailbox of actor `act` being abstracted to the set $\{\texttt{push}(Int), \texttt{pop}(\texttt{display})\}$. To preserve soundness, messages need to be extracted from this abstract mailbox nondeterministically. This is because there is no information about the *multiplicity* of the mes-

sages in the mailbox. A sound static analysis therefore computes not one, but two mailboxes as the result of retrieving the `push` message from this mailbox: $\{\texttt{pop}(\texttt{display})\}$ and $\{\texttt{push}(\mathit{Int}), \texttt{pop}(\texttt{display})\}$. The extraction of the next message from the mailbox $\{\texttt{pop}(\texttt{display})\}$ again yields two mailboxes: $\varnothing$ and $\{\texttt{pop}(\texttt{display})\}$. Through the former case, the analysis accounts for `pop` being present but once and deems the stack underflow error unreachable as a result. Through the latter case, the analysis accounts for `pop` being present more than once. It now deems the stack underflow error reachable as the stack may be empty when a subsequent `pop` message is processed. This false positive results from a loss of precision due to the use of a set abstraction for the mailboxes.

No ordering information.

Another possible abstraction is to abstract mailboxes to multisets (Garoche *et al.*, 2006), where the abstraction preserves information about the number of times a message is present in the abstract mailbox. Multisets are sets that preserve multiplicity but, like sets, are unordered. However, a mailbox abstraction that preserves multiplicity does not suffice either to analyze this program precisely. At the point where the stack actor has received the `push` message followed by the `pop` message, the analysis abstracts the mailbox of actor `act` to $[\texttt{push}(\mathit{Int}) \mapsto 1, \texttt{pop}(\texttt{display}) \mapsto 1]$. This multiset contains the information that both a $\texttt{push}(\mathit{Int})$ message and a $\texttt{pop}(\texttt{display})$ message are present once in the mailbox. Again, the analysis needs to non-deterministically extract the next message to process, giving rise to two possible successor mailboxes: $[\texttt{pop}(\texttt{display}) \mapsto 1]$ and $[\texttt{push}(\mathit{Int}) \mapsto 1]$. The former multiset represents the mailbox of the stack actor after it has processed message $\texttt{push}(\mathit{Int})$. In contrast to the set abstraction, retrieving the next message from this mailbox gives rise to a single mailbox $[]$, because $\texttt{pop}(\texttt{display})$ is present only once, and no stack underflow error can be reached through (spurious) subsequent `pop` messages. However, because ordering information is not preserved, message $\texttt{pop}(\texttt{display})$ might be processed before its corresponding $\texttt{push}(\mathit{Int})$ when dequeuing from the mailbox $[\texttt{push}(\mathit{Int}) \mapsto 1, \texttt{pop}(\texttt{display}) \mapsto 1]$, and the analysis still deems the stack underflow error reachable under a multiset abstraction for the actor's mailbox.

This example motivates the importance of mailbox abstractions that satisfy ordering *and* multiplicity: without one or the other, the analysis cannot automatically infer that the program in Listing 5.1 is free of errors.

5.1.2. Inferring Mailbox Bounds

Consider the example program in Listing 5.2. This is a simplified implementation of the circuit breaker pattern (Kuhn *et al.*, 2017). In this program, an actor `a-service` performs computations that may fail under high load. To avoid high-load situations on that actor, the requests are routed through a *circuit breaker*, implemented by the `a-breaker` actor. This actor buffers requests and dispatches them to the `a-service` actor one at a time, avoiding high load on this actor. A full implementation of the circuit breaker pattern

may perform other tasks, such as identifying when the `a-service` actor is failing, and directly providing a failure notification to the requester, instead of waiting for a reply of the `a-service` actor that might possibly take a long time.

```
1 (define service
2   (actor ()
3     (request (v p)
4       (computation)
5       (send p response)
6       (become a-service))))
7
8 (define breaker-open
9   (actor (target)
10    (request (v)
11      (send target
12        request v self)
13      (become breaker-close
14        target))
15    (response ()
16      (become breaker-open
17        target))))
18 (define breaker-close
19   (actor (target requests)
20     (request (v)
21       (become breaker-close
22         target
23         (append requests (list v))))
24     (response ()
25       (if (empty? requests)
26         (become breaker-open)
27         (begin
28           (send target
29             request (car requests))
30           (become breaker-close
31             target (cdr requests)))))))
32
33 (define a-service (create service))
34 (define a-breaker (create breaker-open a-service))
35 (define (loop i)
36   (send a-breaker request i)
37   (loop (+ i 1)))
38 (loop 0)
```

Listing 5.2: Simplified implementation of the circuit breaker pattern (Kuhn *et al.*, 2017).

The implementation in Listing 5.2 performs the following operations. A request sent to the `a-service` actor takes a value `v` as argument, and a process identifier `p`. Upon processing the request (line 3), the `a-service` actor performs its computation (line 4), after which it sends the response to the requester (line 5). The `a-breaker` actor receives requests as `request` messages with a value `v` as argument. Upon receiving a request, if this actor has the `breaker-open` behavior (line 8), it directly sends the request to the `a-service` actor (line 11), and changes its behavior to `breaker-closed` (line 13). Upon receiving a response from the `a-service` actor, it preserves the `breaker-open` behavior. If the `a-breaker` actor has the `breaker-closed` behavior (line 19), it appends the request to a list of pending requests by updating its behavior (line 22). Upon receiving a response from the `a-service` actor (line 25), either all the requests have been processed, and the `a-breaker` actor changes its behavior back to `breaker-open`, which will directly send the next request to the `a-service` actor. Or, if there are pending requests, the next request is sent to the `a-service` actor (line 29), and is removed from the list of pending requests by updating the behavior (line 31). In this program, an infinite number of requests are dispatched to the `a-breaker` actor (line 37).

One important property of this program is that the `a-service` actor should not be under high load. One can prove that in fact, the mailbox of actor `a-service` will never contain more than one message at any given time. Therefore, the mailbox of this actor is *bounded*, and has a maximal size of 1. Such a property can be statically verified by a mailbox bound analysis (D'Osualdo *et al.*, 2013).

Consider an analysis of this program using a set mailbox abstraction. The mailbox of

actor `a-service` is abstracted to the set $\{\texttt{request}(\mathit{Int}, \texttt{a-breaker})\}$. Sets do not preserve information about the multiplicity of their elements, and the size of this abstract mailbox is therefore approximated to ∞. A static analysis with such a set abstraction cannot prove that the mailbox of `a-service` is bounded. A static analysis that abstracts mailboxes to multisets infers in the abstract mailbox $[\texttt{request}(\mathit{Int}, \texttt{a-breaker}) \mapsto 1]$, and this mailbox has a size of 1. With a multiset mailbox abstraction, it can therefore be inferred that the mailbox of actor `a-service` is bounded to a maximal size of 1. This further motivates the need for mailbox abstractions that do preserve multiplicity information.

5.2. Categorization of Mailbox Abstractions

We now formalize and categorize five abstractions of mailboxes for actors, one of which results in an infinite domain for mailboxes even if the domain of messages $\widehat{\mathit{Message}}$ is finite (*Multiset*) and four of which result in a finite domain for mailboxes is finite when $\widehat{\mathit{Message}}$ is finite (*Set*, *Finite List*, *Finite Multiset*, *Graph*). For completeness, we also categorize the concrete mailbox representation (*List*).

The domain of messages $\widehat{\mathit{Message}}$ is finite if the value domain ($\widehat{\mathit{Val}}$) itself is finite, which is the case when process identifiers ($\widehat{\mathit{PID}}$) are also finite. This is the case for the abstract domains presented in Chapters 2 and 4.

A mailbox abstraction $\widehat{\mathit{Mbox}}$ is defined by instantiating the following functions and elements.

- $\widehat{\mathit{size}} : \widehat{\mathit{Mbox}} \to \mathbb{N} \cup \{\infty\}$ returns the size of a mailbox, and may over-approximate this size to ∞.
- $\widehat{\mathit{empty}} : \widehat{\mathit{Mbox}}$ represents the empty mailbox.
- $\widehat{\mathit{enq}} : \widehat{\mathit{Message}} \times \widehat{\mathit{Mbox}} \to \widehat{\mathit{Mbox}}$ enqueues a message at the back of a mailbox.
- $\widehat{\mathit{deq}} : \widehat{\mathit{Mbox}} \to \mathcal{P}(\widehat{\mathit{Message}} \times \widehat{\mathit{Mbox}})$ dequeues a message from the front of a mailbox. This is a non-deterministic operation. Each element of the resulting set is a tuple containing the message dequeued from the mailbox and the subsequent mailbox.

Table 5.1 depicts a two-dimensional categorization of the mailbox abstractions presented in this chapter, and also contains the concrete representation of mailboxes (*List*). A mailbox abstraction preserves message *ordering* information if it can encode which messages have arrived before others (partially or up to some bound), i.e., there exists a bound n such that $\alpha_n(m_1 : m_2 : mb) \neq \alpha_n(m_2 : m_1 : mb)$. A mailbox abstraction preserves message *multiplicity* if it can encode the number of times a message has been received (up to some bound), i.e., there exists a bound n such that $\alpha_n(m : mb) \neq \alpha_n(m : m : mb)$.

5.2.1. Soundness of Mailbox Abstractions

A mailbox abstraction is sound if the abstraction soundly over-approximates the concrete mailbox. This means that any behavior of a concrete mailbox has to be accounted for by

	Ordering	No ordering
Multiplicity	List (§ 5.2.2)	Multiset (§ 5.2.4)
	Finite List (§ 5.2.6)	Finite Multiset (§ 5.2.5)
No multiplicity	Graph (§ 5.2.7)	Set (§ 5.2.3)

Table 5.1.: Categorization of the concrete *List* mailbox and five abstractions thereof.

its abstraction. For a mailbox abstraction to be sound, the following equations should hold.

- $\forall mb, size(mb) \leq \widehat{size}(\alpha(mb))$, i.e., the abstract size function soundly over-approximates the size of the mailbox.
- $\alpha(empty) \sqsubseteq \widehat{empty}$, i.e., the empty abstract mailbox is a sound over-approximation of the empty concrete mailbox.
- $\forall mb, m, \alpha(enq(m, mb)) \sqsubseteq \widehat{enq}(\alpha(mb), \alpha(m))$, i.e., the abstract enqueuing function soundly over-approximates the concrete enqueuing function.
- $\forall m, mb, mb', (m, mb') \in deq(mb) \implies \exists \widehat{mb}', (\hat{m}, \widehat{mb}') \in \widehat{deq}(\alpha(mb)) \wedge \alpha(mb') \sqsubseteq \widehat{mb}' \wedge \alpha(m) \sqsubseteq \hat{m}$, i.e., for every message that can be dequeued from a concrete mailbox, and for the resulting mailbox, the abstract dequeuing function returns an over-approximating message and over-approximating mailbox.

We provide an abstraction function α and a partial-order relation $\sqsubseteq$ for each mailbox abstraction and prove the soundness of each abstraction in Appendix B (Appendix B.3).

5.2.2. List Representation for Concrete Mailboxes

Figure 5.1 depicts concrete ordered mailboxes represented by lists of messages. We introduced this concrete definition in Chapter 2 and repeat it here. The size of a mailbox is the size of the sequence representing the mailbox. The empty element is the empty sequence. Enqueuing a message in a mailbox adds the message to the front of the mailbox. Dequeuing a message from a mailbox extracts the last message from the mailbox, and results in the singleton set containing a pair of this message with the resulting mailbox. Dequeuing from an empty mailbox results in the empty set.

λ_α

$$mb \in Mbox = Message^* \qquad enq(m, mb) = m : mb$$
$$size(mb) = |mb| \qquad deq(\epsilon) = \varnothing$$
$$empty = \epsilon \qquad deq(mb : m) = \{(m, mb)\}$$

Figure 5.1.: List representation of mailboxes.

This representation is not finite, even if the domain of messages is made finite. It preserves information about both the ordering and the multiplicity of the messages contained.

5.2.3. Set Abstraction

Figure 5.2 depicts the set mailbox abstraction, used in Chapters 2, 4 and 6 and in related work (D'Osualdo *et al.*, 2013). Concrete mailboxes are abstracted to sets that contain abstract messages. We repeat this definition here, and name the domain of abstract mailboxes abstracted to sets as *Set* (in Chapters 2 and 4, we use $\widehat{Mbox} = Set$).

A mailbox is abstracted to a set of abstract messages. The empty mailbox is abstracted to the empty set and it is always empty, hence its size is 0. The size of a mailbox containing messages is not known—because of the lack of multiplicity information—and is therefore approximated to ∞. Enqueuing a message in a mailbox entails joining it into the set representing the mailbox. Dequeuing a message from a mailbox accounts for all possible orderings and multiplicities: any message can be dequeued ($\hat{m} \in \widehat{mb}$), and the resulting mailbox may contain the message to account for the corresponding concrete mailboxes that contain it more than once ($(\hat{m}, \widehat{mb})$), or may not contain the message to account for the corresponding concrete mailboxes that contain it only once ($(\hat{m}, \widehat{mb} \setminus \{\hat{m}\})$).

λ_α

$$\widehat{mb} \in Set = \mathcal{P}(\widehat{Message}) \qquad \widehat{empty}_{Set} = \varnothing$$
$$\widehat{size}_{Set}(\varnothing) = 0 \qquad \widehat{enq}_{Set}(\hat{m}, \widehat{mb}) = \widehat{mb} \cup \{\hat{m}\}$$
$$\widehat{size}_{Set}(\widehat{mb}) = \infty \qquad \widehat{deq}_{Set}(\widehat{mb}) = \left\{ (\hat{m}, \widehat{mb}), (\hat{m}, \widehat{mb} \setminus \{\hat{m}\}) \mid \hat{m} \in \widehat{mb} \right\}$$

Figure 5.2.: Set abstraction for mailboxes.

Though sound (Theorem 11), this coarse abstraction only keeps track of which messages are present in the mailbox, and preserves neither ordering nor multiplicity of messages.

Theorem 11 (Soundness of the set mailbox abstraction)**.** *The set mailbox abstraction is sound.*

Proof. The proof entails showing that the properties given in Section 5.2.1 hold. $\widehat{size}_{Set}$ is proven sound by a case analysis on the concrete mailbox. $\widehat{empty}_{Set}$ is sound by definition. $\widehat{enq}_{Set}$ is proven sound by induction on the concrete mailbox. $\widehat{deq}_{Set}$ is proven sound by showing that it accounts for a message being present once or more than once in the concrete mailbox. This proof is detailed in Appendix B.3.1. □

5.2.4. Multiset Abstraction

Figure 5.3 depicts the multiset mailbox abstraction, used in related work as concrete representation of unordered mailboxes (Agha *et al.*, 1997; Garoche *et al.*, 2006). With this abstraction, a mailbox is abstracted to a multiset that keeps track of the multiplicity of each abstract message within the mailbox, but does not preserve ordering information. A multiset is represented as a mapping from abstract messages to integers representing the multiplicity of the corresponding message. This multiplicity represents *concrete* information: a multiplicity of n associated to an abstract message $\hat{m}$ in a mailbox $\widehat{mb}$ represents that in the concrete, mailboxes abstracted to $\widehat{mb}$ contain n concrete messages that are abstracted to $\hat{m}$, e.g., the same concrete message n times, or different messages that are all abstracted to $\hat{m}$. The abstract size of a mailbox abstracted by a multiset is the sum of the multiplicities of all the messages contained in the mailbox. The empty mailbox is the mapping that maps all messages to zero, as no message is present in the empty mailbox. Enqueuing an element in a mailbox increases its multiplicity by one. The dequeuing operation accounts for all possible orderings ($\hat{m} \in \mathrm{dom}(\widehat{mb})$) of messages that are present in the mailbox ($\widehat{mb}(\hat{m}) \geq 1$), and decreases the count of the dequeued message in the resulting mailbox.

λ_α

$$\widehat{mb} \in MS = \widehat{Message} \to \mathbb{N} \qquad \widehat{enq}_{MS}(\hat{m}, \widehat{mb}) = \widehat{mb}[\hat{m} \mapsto \widehat{mb}(\hat{m}) + 1]$$

$$\widehat{size}_{MS}(\widehat{mb}) = \sum_{\hat{m} \in \mathrm{dom}(\widehat{mb})} \widehat{mb}(\hat{m}) \qquad \widehat{deq}_{MS}(\widehat{mb}) = \Big\{ (\hat{m}, \widehat{mb}[\hat{m} \mapsto \widehat{mb}(\hat{m}) - 1]) \mid \hat{m} \in \mathrm{dom}(\widehat{mb}) \wedge \widehat{mb}(\hat{m}) \geq 1 \Big\}$$

$$\widehat{empty}_{MS} = \lambda \hat{m}.0$$

Figure 5.3.: Multiset abstraction for mailboxes.

The multiset abstraction is sound (Theorem 12) but its domain is infinite: there is no limit on the number of times each message may appear. However, this abstraction can be rendered finite by approximating the multiplicity associated to each message, as done in the next section.

Theorem 12 (Soundness of multiset abstraction)**.** *The multiset mailbox abstraction is sound.*

Proof. The proof entails showing that the properties given in Section 5.2.1 hold. $\widehat{size}_{MS}$ is proven sound by induction on the concrete mailbox. $\widehat{empty}_{MS}$ is sound by definition. $\widehat{enq}_{MS}$ is proven sound by induction on the concrete mailbox. $\widehat{deq}_{MS}$ is proven sound by showing that it accounts for all possible messages present in the abstract mailbox to be returned. This proof is detailed in Appendix B.3.2. □

5.2.5. Finite Multiset Abstraction

Figure 5.4 depicts the finite multiset mailbox abstraction, which renders multisets finite by imposing a bound on the multiplicity of each abstract message. Once this bound is exceeded for an abstract message, the multiplicity of that specific message is abstracted to ∞. The mailbox domain of the finite multiset abstraction maps to the union of bounded integers and the multiplicity ∞ ($\mathbb{N}^{\leq n} \cup \{\infty\}$). The empty mailbox is represented as in the multiset abstraction, and maps all messages to a multiplicity of 0 ($\lambda\hat{m}.0$). The size of the mailbox is also the sum of the multiplicities of all elements contained in the abstract mailbox, and becomes ∞ when at least one of the multiplicities is over-approximated to ∞. Enqueuing a message increases the multiplicity of the enqueued message ($\widehat{mb}(\hat{m}) + 1$), over-approximating it to ∞ when it exceeds the bound n (i.e., when $\widehat{mb}(m) \geq n$). The dequeuing operation accounts for all possible orderings ($\hat{m} \in \mathrm{dom}(\widehat{mb})$) and decreases the multiplicity of the dequeued message if it is not over-approximated. If the multiplicity of the message is over-approximated to ∞, then either the message is contained in the mailbox more than once and its multiplicity remains over-approximated ($(\hat{m}, \widehat{mb})$), or it was contained only once and its multiplicity is set to zero ($(\hat{m}, \widehat{mb}[\hat{m} \mapsto 0])$). Note that its multiplicity could also be set to n, but setting it to zero suffices as a sound over-approximation according to the ordering relation $\sqsubseteq$ defined in Appendix B, and reduces the non-determinism of this abstraction. Function $\widehat{size}_{MS_n}$ would be more precise on specific mailboxes, but this precision would not profit client analyses such as a mailbox bound analysis, as the maximal size of the mailbox on the entire program's execution would remain ∞.

λ_α

$$
\begin{aligned}
\widehat{mb} \in MS_n &= \widehat{Message} \to (\mathbb{N}^{\leq n} \cup \{\infty\}) \\
\widehat{empty}_{MS_n} &= \lambda\hat{m}.0 \\
\widehat{size}_{MS_n}(\widehat{mb}) &= \sum_{\hat{m} \in \mathrm{dom}(\widehat{mb})} \widehat{mb}(\hat{m}) \\
\widehat{enq}_{MS_n}(\hat{m}, \widehat{mb}) &= \widehat{mb}[\hat{m} \mapsto \widehat{mb}(\hat{m}) + 1] \qquad \text{if } \widehat{mb}(\hat{m}) < n \\
&= \widehat{mb}[\hat{m} \mapsto \infty] \qquad \text{otherwise} \\
\widehat{deq}_{MS_n}(\widehat{mb}) &= \left\{ (\hat{m}, \widehat{mb}[\hat{m} \mapsto \widehat{mb}(\hat{m}) - 1]) \mid \hat{m} \in \mathrm{dom}(\widehat{mb}) \wedge 1 \leq \widehat{mb}(\hat{m}) \leq n \right\} \\
&\quad \cup \left\{ (\hat{m}, \widehat{mb}), (\hat{m}, \widehat{mb}[\hat{m} \mapsto 0]) \mid \hat{m} \in \mathrm{dom}(\widehat{mb}) \wedge \widehat{mb}(\hat{m}) = \infty \right\}
\end{aligned}
$$

Figure 5.4.: Finite multiset abstraction for mailboxes.

This mailbox abstraction is sound (Theorem 13) and finite. It does not preserve ordering information but does preserve multiplicity information on a per-message basis: the

multiplicity of each message is preserved up to the bound n.

Theorem 13 (Soundness of finite multiset abstraction)**.** *The finite multiset mailbox abstraction is sound.*

Proof. The proof relies on the soundness of the multiset abstraction (Theorem 12), and entails showing that abstracting the multiplicity of a message to ∞ is sound. This proof is detailed in Appendix B.3.3. □

5.2.6. Finite List Abstraction

Combining the set abstraction with the concrete representation of mailboxes results in the *finite list* abstraction depicted in Figure 5.5 and denoted L_n. We use the notation $\widehat{Message}_n^*$ to denote lists of messages with a length that does not exceed n (see Appendix A). The finite list abstraction represents an abstract mailbox as a list up to the point where the size of the list exceeds a specific bound n, after which the mailbox is abstracted to a set. The size of the mailbox is approximated by the size function of the corresponding domain, i.e., *size* for mailboxes abstracted to lists and $\widehat{size}_{Set}$ for mailboxes abstracted to sets. The empty concrete mailbox is abstracted to an empty list. Enqueuing a message in a mailbox relies on the enqueuing function defined on the corresponding domain, except if the size of the mailbox would exceed its bound n after enqueuing. If this is the case, the mailbox is abstracted to a set using *toSet* (where $toSet(\epsilon) = \varnothing$, and $toSet(\hat{m} : \widehat{mb}) = \{\hat{m}\} \cup toSet(\widehat{mb})$). The dequeuing of an element from a mailbox is defined in terms of the dequeuing function of the corresponding domain: if the mailbox is abstracted to a list, a message is dequeued with function *deq*, and if the mailbox is abstracted to a set, a message is dequeued with function $\widehat{deq}_{Set}$.

λ_α

$$
\begin{aligned}
\widehat{mb} \in L_n &= \widehat{Message}_n^* \cup Set \\
\widehat{size}_{List_n}(\widehat{mb}) &= size(\widehat{mb}) && \text{if } \widehat{mb} \in \widehat{Message}_n^* \\
&= \widehat{size}_{Set}(\widehat{mb}) && \text{if } \widehat{mb} \in Set \\
\widehat{empty}_{L_n} &= \epsilon \\
\widehat{enq}_{L_n}(\hat{m}, \widehat{mb}) &= enq(\hat{m}, \widehat{mb}) && \text{if } \widehat{mb} \in \widehat{Message}_n^* \wedge |\widehat{mb}| < n \\
&= \widehat{enq}_{Set}(\hat{m}, toSet(\widehat{mb})) && \text{if } \widehat{mb} \in \widehat{Message}_n^* \wedge |\widehat{mb}| = n \\
&= \widehat{enq}_{Set}(\hat{m}, \widehat{mb}) && \text{if } \widehat{mb} \in Set \\
\widehat{deq}_{L_n}(\widehat{mb}) &= deq(\widehat{mb}) && \text{if } \widehat{mb} \in \widehat{Message}_n^* \\
&= \widehat{deq}_{Set}(\widehat{mb}) && \text{if } \widehat{mb} \in Set
\end{aligned}
$$

Figure 5.5.: Finite list abstraction for mailboxes.

This finite and sound abstraction (Theorem 14) preserves ordering and multiplicity of the abstract messages in the abstract mailbox up to the point where the bound is reached. Once the number of messages in the mailbox exceeds the bound n, the finite list abstraction behaves like the set abstraction, rendering it finite.

Note that the finite list mailbox abstraction has two values for the empty mailbox: $\epsilon \in \mathit{Mbox}$ and $\varnothing \in \mathit{Set}$. It is safe to replace $\varnothing$ by ϵ when it appears during an analysis as the concrete empty mailbox is the only mailbox abstracted by $\varnothing$. We do so in our implementation, as this provides extra precision to this abstraction.

Theorem 14 (Soundness of finite list abstraction)**.** *The finite list mailbox abstraction is sound.*

Proof. The proof relies on the fact that up to the point where the bound n is reached, it behaves like a concrete mailbox containing abstract messages, and that when abstracted to a set, it is sound according to Theorem 11. This proof is detailed in Appendix B.3.4. □

5.2.7. Graph Abstraction

The graph abstraction described here is a new abstraction that we introduced (Stiévenart *et al.*, 2017), and which resembles the regular expression abstraction introduced by Midtgaard *et al.* (2016a). It results from the observation that ordering relation between abstract messages in a mailbox may be approximated by edges between nodes in a graph. A mailbox is abstracted to a graph of which the nodes correspond to messages and of which the edges express their ordering in the mailbox: an edge from node a to b indicates that message b was enqueued after a in the mailbox. A mailbox abstracted to a graph has a *first* pointer, denoting the front of the mailbox, and a *last* pointer, denoting the back of the mailbox. When a message is enqueued in the back of a mailbox, a node is added in the graph with an directed edge connecting the previous last node of the graph to this new node, which becomes the last node of the graph. When a message is dequeued, the first node of the graph is extracted, and its edge points to the new first node of the graph. This abstraction maintains information about the first and last message in the mailbox. Figure 5.6 depicts the following evolution over time of a mailbox using this abstraction.

(a) Enqueuing message a in the empty mailbox creates a node a, and makes the *first* (f) and *last* (l) pointers point to this node (Figure 5.6a).

(b) Enqueuing message b creates a new node connected to the previous *last* pointer, updates the *last* pointer, but leaves the *first* pointer as is (Figure 5.6b).

(a) Enqueuing message a does not create a new node since the node a is already in the graph, but does add a new edge from b to a, and updates the *last* pointer (Figure 5.6c).

(c) Dequeuing a message yields the message a pointed by the *first* pointer. The resulting mailbox has the same graph, but the *first* pointed is updated to point to the target of its outgoing edge of its current node (Figure 5.6d). If this node has more

than one outgoing edges, this is a non-deterministic operation that yield multiple possible results, one per outgoing edges.

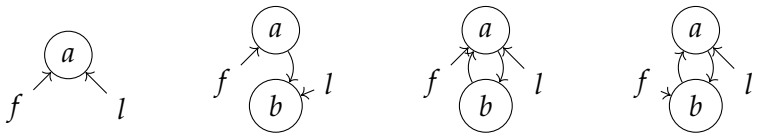


Figure 5.6.: Visual representation of the graph mailbox abstraction. f is the *first* pointer and l is the *last* pointer of the abstract mailbox.

Figure 5.7 depicts the formalization of the graph mailbox abstraction. A graph mailbox abstraction is either bottom ($\bot$), denoting the empty mailbox or a tuple containing the set of nodes in the graph, the set of edges in the graph, the node corresponding to the *first* message, and the node corresponding to the *last* message. The size of the graph is given by the length of the unique path between the first node and the last node, if that path is unique, otherwise it is over-approximated to ∞. This is computed by the *PathLength* function (defined in Appendix B), which returns n if there is a single path between f and l, and this path has length n. For example, there is a single path from the first to the last message in Figures 5.6a and 5.6b, but not in Figure 5.6c nor in Figure 5.6d. Enqueuing a message in the empty mailbox initializes the graph with a single node ($\{\hat{m}\}$), no edges, and the first and last pointers pointing to the enqueued message. Enqueuing a message in a non-empty mailbox adds the enqueued message to the set of nodes ($V \cup \{\hat{m}\}$), adds an edge from the current last node to the enqueued message ($E \cup \{\langle l, \hat{m}\rangle\}$), and sets the last pointer to the node for the enqueued message. Dequeuing a message from the empty mailbox results in the empty set, as no message can be dequeued because no message is contained in the mailbox. Dequeuing from a mailbox containing a single message results in the dequeued message and the empty mailbox. A mailbox contains a single message only if there is no edge starting at its first node ($\nexists f' \mid (f, f') \in E$). Otherwise, the resulting mailbox is updated so that the first node is set to one of the successor of the previous first node. This is a non-deterministic operation, as the first node may have more than one edge to other nodes.

λ_α

$$
\begin{aligned}
\widehat{mb} \in G = (&\mathcal{P}(\widehat{Message}) \times \mathcal{P}(\widehat{Message} \times \widehat{Message}) \\
&\times \widehat{Message} \times \widehat{Message}) \cup \{\bot\} \\
\widehat{empty}_G &= \bot \\
\widehat{size}_G(\bot) &= 0 \\
\widehat{size}_G(\langle V, E, f, l\rangle) &= 1 + \mathit{PathLength}(f, l, \langle V, E\rangle) \\
\widehat{enq}_G(\hat{m}, \bot) &= \langle \{\hat{m}\}, \varnothing, \hat{m}, \hat{m}\rangle \\
\widehat{enq}_G(\hat{m}, \langle V, E, f, l\rangle) &= \langle V \cup \{\hat{m}\}, E \cup \{\langle l, \hat{m}\rangle\}, f, \hat{m}\rangle \\
\widehat{deq}_G(\bot) &= \varnothing \\
\widehat{deq}_G(\langle V, E, f, l\rangle) &= \{(f, \bot)\} \text{ if } \nexists f' \mid (f, f') \in E \\
\widehat{deq}_G(\langle V, E, f, l\rangle) &= \{(f, \langle V, E, f', l\rangle) \mid (f, f') \in E, f' \in V\} \text{ otherwise}
\end{aligned}
$$

Figure 5.7.: Graph abstraction for mailboxes.

This sound abstraction (Theorem 15) preserves ordering information but does not preserve multiplicity according to our definition of preservation of multiplicity. This is because, unlike the finite list abstraction for which a bound n can be found to preserve the multiplicity of any mailbox, the graph abstraction does not have such bound and may not preserve multiplicity information for any mailbox, e.g., the size of a mailbox $a : b : a$ abstracted to a graph is always ∞. The graph abstraction is finite when the domain of messages is finite, as the number of nodes in the graph is bounded by the number of abstract messages.

Theorem 15 (Soundness of graph abstraction)**.** *The graph mailbox abstraction is sound.*

Proof. The proof entails showing that the properties given in Section 5.2.1 hold. $\widehat{size}_G$ is sound because it either over-approximates the size to ∞, or has the maximally precise size when there exists a single path from the first node to the last node in the graph. $\widehat{empty}_G$ is sound by definition. $\widehat{enq}_G$ is proven sound by induction on the concrete mailbox. $\widehat{deq}_G$ is proven by a case analysis on the concrete mailbox. This proof is detailed in Appendix B.3.5. □

5.3. Evaluation of Mailbox Abstractions

We extended the implementation of the analysis resulting from the application of MacroConc to λ_α presented in Chapter 4 to perform verification of absence of errors and inference of mailbox bounds. Both the verification and the inference are performed by a walk through the reachable states computed by the analysis. If no error state is reachable in the analysis results, the program is deemed to be free of errors. The inferred bound for the mailbox of an actor in a program is the maximal mailbox size observed within the

reachable states in the results of the analysis. This is because these results correspond to the program's abstract collecting semantics.

We implemented the different mailbox abstractions presented in this chapter and use this implementation to evaluate the applicability of these abstractions empirically. The benchmark suite introduced in Section 2.4.2 is not well-suited for evaluating the extent to which the analysis supports verification of absence of errors and inference of mailbox bounds, as only a few benchmark programs contain unreachable errors and bounded mailboxes. We therefore introduce a different benchmark suite containing a number of programs that exhibit such properties (Section 5.3.1). We perform verification of absence of errors and inference of mailbox bounds on these programs using our analysis, and using Soter (D'Osualdo *et al.*, 2012). We measure the precision of the analysis with different mailbox abstractions (Section 5.3.2). To assess the applicability of the different mailbox abstractions to real-world programs, we run the analysis with each mailbox abstraction on our full benchmark suite introduced in Section 2.4.2 (Section 5.3.3).

5.3.1. Benchmark Suite for Absence of Errors and Mailbox Bounds

We introduce in Table 5.2 a small benchmark suite containing programs exhibiting unreachable errors and bounded mailboxes. This benchmark suite is different from the one used in the remainder of this dissertation, as well as from the one used in Section 5.3.3. We use this small benchmark suite to evaluate the impact of the different mailbox abstractions on the precision of the analysis in the next section. The use of this different benchmark suite is justified by the fact that it contains programs that exhibit unreachable errors and bounded mailboxes, unlike the benchmark suite used in the rest of this dissertation. This renders this specific benchmark suite suitable to evaluate the precision of client analyses that perform verification of unreachable errors and inference of mailbox bounds, as evaluated in the next section. We translated benchmark programs exhibiting unreachable errors and mailbox bounds from multiple sources to λ_α, remaining as close as possible to their original implementation. We unrolled all loops that create a fixed number of actors, in order to benefit from the additional precision offered by abstract counting.

In order to compare our approach with Soter, which analyzes Erlang programs, we faithfully translated all the benchmarks in Erlang. The correspondence between the λ_α and Erlang versions of the benchmarks is as close as possible. Table 5.2 lists the programs that compose this benchmark suite. These benchmarks are adapted from the Savina benchmark suite (Imam and Sarkar, 2014), from Agha (1986), and from D'Osualdo *et al.* (2012). They feature dynamic process creation, but we did not target benchmark programs with an unbounded number of concrete actors, as this is an orthogonal problem to the points discussed in this chapter. Support for unbounded creation of processes is however at the core of the MacroConc and ModConc analysis design methods (Chapters 4 and 6).

Bench.	LOCs	Bench.	LOCs
PARIKH	22	CELL	13
SAFE-SEND	15	PIPE-SEQ	15
STUTTER	15	STATE-FAC	24
STACK	20	PP	27
COUNT-SEQ	23	FJC-SEQ	13

Table 5.2.: Benchmark programs exhibiting unreachable errors and bounded mailboxes used in the evaluation of the analyses. Note that the line count differs for the listings of this chapter presenting benchmark programs, as we performed formatting change to improve the readability of the code.

5.3.2. Precision

We evaluate the precision of the different mailbox abstractions for verification of absence of errors and inference of mailbox bounds. As part of this precision evaluation, we also compare our analyses of λ_α with Soter (D'Osualdo *et al.*, 2013), because Soter can perform verification of absence of errors and verification of mailbox bounds. We refer to Chapter 4 for a detail description of Soter, but we recall two important differences between our approach and Soter.

1. Soter generates a coarse flow graph as the model of a program, and then performs model checking on this flow graph to verify program properties. Our approach is to construct a more precise flow graph of the program on which the verification can be performed directly, not requiring a separate model checking phase to prove the absence of run-time errors or bounds on mailboxes. To highlight this difference, consider the PARIKH benchmark depicted in Listing 5.3. It defines a `server` actor that expects `init` as a first message, but throws an error if it receives a second `init` message. With a set mailbox abstraction, which does not preserve multiplicity, the error is reachable in the graph generated by Soter. However, it can be proved unreachable by performing an extra model-checking phase. With a suitable mailbox abstraction, our approach benefits from improved precision, resulting in a smaller and more precise flow graph that does *not* contain the error state. No further model checking phase is therefore required.

2. Soter requires the user to specify the properties that need to be verified through code annotations. The model checking part of Soter relies on these annotations to drive the model checker. An important difference with our analysis for the mailbox bounds analysis is that our analysis *infers* the size of the mailbox of each actor in the program, while Soter *verifies* that specific mailboxes do not exceed the size specified in their annotation.

Table 5.3 lists the results of our analysis with the different mailbox abstractions and of Soter on programs that exhibit unreachable errors (Table 5.3a), and on programs with bounded mailboxes (Table 5.3b). We can observe the following.

Set abstraction

```
(define server-init-actor
  (actor ()
    (init (p x)
      (send p ok)
      (become server-actor x))))
(define server-actor
  (actor (x)
    (init (p x)
      (error))
    (set (y)
      (become server-actor y))
    (get (p)
      (send p result x))
    (bye ()
      (terminate))))
(define after-init-actor
  (actor (s)
    (ok ()
      (send s set 'b)
      (send s bye)
      (terminate))))

(define s (create server-init-actor))
(define after-init
  (create after-init-actor s))
(send s init after-init 'a)
```

Listing 5.3: PARIKH benchmark program.

```
(define stutter
  (actor (f)
    (message (x)
      (become unstutter f))))
(define unstutter
  (actor (f)
    (message (x)
      (f x)
      (unstutter))))
(define (dosmt x)
  (if (= x 0)
    (error)
    'ok))

(define p (create stutter dosmt))

(define (sendA p)
  (send p message 0)
  (sendB p))
(define (sendB p)
  (send p message 1)
  (sendA p))

(sendA p)
```

Listing 5.4: STUTTER benchmark program.

```
(define fac-actor
  (actor (p)
    (call (in)
      (send p update self in)
      (become fac-wait-actor p))))
(define fac-wait-actor
  (actor (p)
    (done ()
      (become fac-actor p))
    (call (n)
      (send self call n)
      (become fac-wait-actor p))))
(define state-actor
  (actor (n f)
    (update (p in)
      (send p done)
      (become state-actor m (f n in)))))

(define state
  (create state-actor
    (random) (lambda (x y) (random))))
(define factory (create fac-actor state))

(define (call-loop n f)
  (if (= n 1)
    (send f call (random))
    (begin
      (send f call (random))
      (call-loop (- n 1) f))))

(call-loop (random) factory)
```

Listing 5.5: STATE-FAC benchmark program.

The set mailbox abstraction does not preserve multiplicity information and over-approximates all non-empty mailboxes with a potentially infinite size, rendering it unfit for inference of mailbox bounds. Moreover, this coarse abstraction is not precise enough to verify any of the programs in our benchmark suite as free of errors.

Multiset abstraction

Through its preservation of multiplicity information, the multiset abstraction enables verifying the PARIKH program as free of errors, and enables inferring all mailbox bounds.

List abstraction

Through its preservation of ordering information, the list abstraction enables verifying the rest of the programs except the STUTTER and STATE-FAC benchmark programs. These benchmark programs, respectively depicted in Listings 5.4 and 5.5, exhibit infinitely growing mailboxes in the concrete that follow a specific pattern (respectively, $a : b : a : b : \ldots$ and $a : a : \ldots : a : b$), for which the finite list abstraction loses all precision, as no value of the bound for this abstraction is sufficient to analyze these benchmark programs with maximal precision.

Graph abstraction

Through its preservation of ordering information on specific infinite mailboxes, the graph abstraction can preserve the ordering on the infinite mailboxes of programs STUTTER and STATE-FAC, and therefore enables verifying these benchmarks.

In contrast to our approach, Soter uses a set mailbox abstraction, but is able to verify benchmarks despite this very coarse abstraction, thanks to the second model checking phase included in Soter.

Bench.	*Set*	MS_n		L_n		*G*	Soter
PARIKH	✗	✓	1	✓	1	✓	✓
SAFE-SEND	✗	✗	–	✓	4	✓	✓
STUTTER	✗	✗	–	✗	–	✓	✗
STACK	✗	✗	–	✓	4	✓	✗
COUNT-SEQ	✗	✗	–	✓	2	✓	✗
CELL	✗	✗	–	✓	2	✓	✗

(a) Precision evaluation of mailbox abstractions for verification of absence of errors. For each mailbox abstraction, we indicate the programs for which the absence of errors can be verified (✓) or not (✗), and for the abstractions with a bound parameter, we indicate the value of the parameter.

Bench.	*Set*	MS_n		L_n		*G*	Soter
PIPE-SEQ	✗	✓	1	✓	1	✓	✓
STATE-FAC	✗	✓	1	✗	–	✓	✓
PP	✗	✓	1	✓	1	✓	✓
COUNT-SEQ	✗	✓	1	✓	2	✓	✓
CELL	✗	✓	1	✓	2	✓	✓
FJC-SEQ	✗	✓	1	✓	1	✓	✓

(b) Precision evaluation of mailbox abstractions for inference of mailbox bounds. For each mailbox abstraction, we indicate the programs for which the correct mailbox bound can be inferred (✓) or not (✗), and for the abstractions with a bound parameter, we indicate the value of the parameter.

Table 5.3.: Precision evaluation of mailbox abstractions.

Spurious Mailboxes and Spurious Dequeued Messages

To evaluate the impact of the different mailbox abstractions on the precision of the analysis resulting from the application of MacroConc to λ_α, we analyzed the small benchmark suite introduced in Section 5.3.1 and report on the number of spurious values detected.

Instead of comparing the messages dequeued and the executions of `become` expressions during abstract interpretation and concrete interpretations, as done in Chapter 4, we perform the precision evaluation at a finer-grained level to obtain more insights to the precision of the mailbox abstractions. This finer-grained precision evaluation not only takes into account the messages dequeued from the mailbox, but also takes into account the mailbox from which they are dequeued. We compare abstract mailboxes computed during an analysis and messages dequeued from these abstract mailboxes to the concrete values that can be observed in concrete executions. Resulting from over-approximation, a spurious abstract value lacks corresponding concrete values in actual runs of the program. The more spurious values, the less precise the results of an analysis. We counted the following spurious values in the analysis results and summed the results for all programs listed in Table 5.2, which are depicted in Figure 5.8:

1. spurious mailboxes, i.e., mailboxes that results from too much over-approximation and correspond to no concrete mailboxes,
2. spurious messages resulting from spurious mailboxes (*Spurious Messages #1*),
3. spurious messages resulting from non-spurious mailboxes (*Spurious Messages #2*).

Any message dequeued from a spurious mailbox is a spurious message, directly linking the number of such spurious messages to the number of spurious mailboxes. This link is not that direct for spurious messages resulting from non-spurious mailboxes, and at least a different mailbox abstraction is required to decrease the number of spurious messages in this category. For example, a non-empty mailbox will always yield spurious messages if abstracted to a set, no matter the precision of the other abstractions used in the analysis.

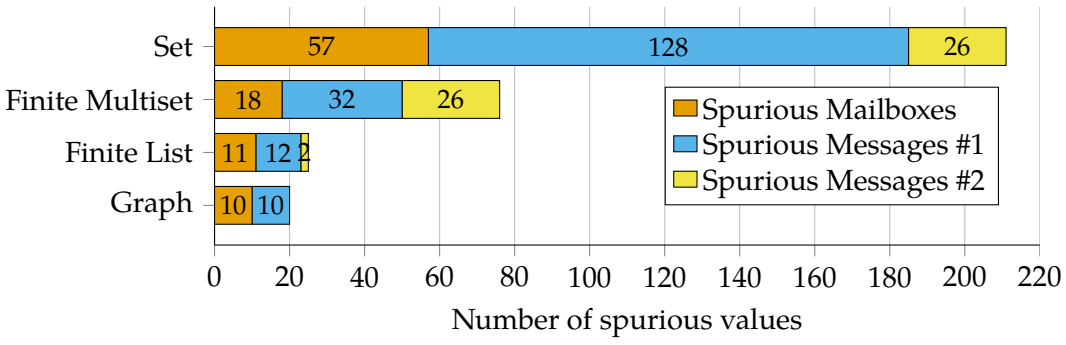


Figure 5.8.: Precision metrics for the different mailbox abstractions (lower is better).

	Set			MS_n			L_n			*G*		
Bench.	A	B	C	A	B	C	A	B	C	A	B	C
PARIKH	4	7	1	0	0	1	0	0	0	0	0	0
SAFE-SEND	21	61	11	3	7	11	0	0	0	0	0	0
STUTTER	2	2	2	4	7	2	4	4	2	0	0	0
STACK	4	9	10	6	13	10	0	0	0	0	0	0
COUNT-SEQ	3	5	1	1	1	1	0	0	0	0	0	0
CELL	0	0	1	0	0	1	0	0	0	0	0	0
PIPE-SEQ	16	30	0	4	4	0	4	4	0	4	4	0
STATE-FAC	2	3	0	0	0	0	3	4	0	6	6	0
PP	5	11	0	0	0	0	0	0	0	0	0	0
FJC-SEQ	0	0	0	0	0	0	0	0	0	0	0	0
Total	57	128	26	18	32	26	11	12	2	10	10	0

Table 5.4.: Precision metrics for the different mailbox abstractions. Column *A* indicates the number of spurious mailboxes that correspond to no concrete mailbox in a concrete run of the benchmark. Column *B* indicates the number of spurious values dequeued from spurious mailboxes. Column *C* indicates the number of spurious values dequeued from non-spurious mailboxes.

The results, detailed in Table 5.4, show that the coarse set abstraction is the most imprecise abstraction, resulting in many spurious values. For example, the SAFE-SEND program exhibits a mailbox with 4 different unique messages in a specific order, and the set abstraction yields many spurious values due to its loss of precision on ordering and multiplicity information. The multi-set abstraction benefits from higher precision because it preserves multiplicity, therefore resulting in fewer spurious mailboxes. However, because it lacks ordering information, it does not improve over the set abstraction in the number of spurious messages resulting from non-spurious mailboxes. The list and graph abstractions preserve both multiplicity and ordering, which renders them more precise. On benchmarks with an unbounded number of sent messages, both lose some precision. However, when the messages in a possibly infinite mailbox follow a specific pattern, the graph abstraction yields a better precision than the finite list abstraction. This is because the list abstraction reduces to a set once the bound is reached, thereby losing ordering information, while ordering information may be preserved by the graph abstraction. This is the case in the STUTTER benchmark program for example, where messages sent to an actor follow a specific pattern (one, zero, one, zero, etc.). The list abstraction reaches its bound and over-approximates the mailbox by the set $\{0, 1\}$ and results in 10 spurious values, while the graph preserves the ordering information and analyzes this benchmark with full precision (i.e., without spurious values).

The only program where the graph abstraction yields more spurious values than the other abstractions is STATE-FAC, depicted in Listing 5.5. This is because this program contains an actor receiving a specific message an unbounded number of times, as well as a single instance of another message. Due to the former message being received an

unbounded number of times, the graph abstraction does not maintain the multiplicity information that the other message is unique. Using the finite multiset abstraction preserves this multiplicity and yields no spurious values. The set and finite list abstractions do not preserve this multiplicity information, and therefore yield spurious values. However because these abstractions have a smaller domain size, they produce less spurious mailboxes compared to the graph abstraction (2 for the set abstraction, 3 for the finite list abstraction, 6 for the graph abstraction). This is because multiple spurious values in a more precise abstraction may be represented by a single spurious value in a less precise mailbox, e.g., if $a : b$ and $b : a$ are two spurious values for a list abstraction, a single spurious value is present in the set abstraction: $\{a, b\}$.

5.3.3. Running Times on Full Benchmark Suite

To evaluate the impact of the different mailbox abstractions on the efficiency of the analysis resulting from the application of MacroConc to λ_α, we analyzed 20 times each of the programs from the benchmark suite introduced in Section 2.4.2, after 10 warmup runs, and report on the average time required to analyze each program. The results are listed in Table 5.5.

We see that an increase in precision from the mailbox abstractions does not result in a consistent improvement of the scalability. In some cases, the analysis exhibits an improved running time with the more precise mailbox abstractions presented in this chapter. This is the case for programs where either all actors receive a single type of message (FJT, FJC), which is well-supported by the more precise abstractions, or where messages follow a specific pattern well-supported by the more precise abstractions (PP, COUNT). In other cases, the running time increases when the analysis does not profit from the increased precision from the mailbox abstraction, and results in an increased state space size (e.g., RSORT). This is because the analysis loses precision for reasons not linked to the mailbox abstraction (e.g., due to context-insensitivity), and this loss of precision is reflected in the mailbox abstraction which creates more non-determinism due to its more precise representation of mailboxes. A less precise abstraction has less impact on the size of the state space in this case.

With the set mailbox abstraction, 12 out of the 28 benchmarks are analyzed within the time limit. With the finite multiset abstraction, one more program times out (BTX), and the time required to analyze some programs increases by one order of magnitude (CDICT, PCBB, RSORT), while the time decreases on smaller programs (PP, COUNT, FJT). The finite list abstraction supports the 12 same benchmarks as the set abstraction, with relatively similar running times, except on the smaller programs for which the analysis time is reduced, while the BTX program now takes one order of magnitude more time to analyze. Finally, the graph abstraction, the most precise according to our precision evaluation, supports fewer benchmark programs: only 8 compared to the 12 supported by the set abstraction.

These more precise abstractions therefore do not result in improved scalability. However, the cost in scalability to pay for the improved precision is low. We identify as an interesting avenue for future work the combination of different abstractions, where the

set abstraction could be used by default for most actors, yielding a reduced running time, and abstractions with higher precision could be used for other actors based on either user-provided code annotations or heuristics, to profit from an increased precision on these mailboxes.

Bench.	*Set*	MS_n	L_n	*G*	Bench.	*Set*	MS_n	L_n	*G*
PP	494	7	7	7	BTX	2567	∞	14493	∞
COUNT	284	24	24	18	RSORT	12913	153432	147773	133711
FJT	112	58	54	48	FBANK	∞	∞	∞	∞
FJC	15	16	14	14	SIEVE	79	247	103	∞
THR	∞	∞	∞	∞	UCT	∞	∞	∞	∞
CHAM	∞	∞	∞	∞	OFL	∞	∞	∞	∞
BIG	∞	∞	∞	∞	TRAPR	10301	24920	13098	10339
CDICT	21246	211467	19215	∞	PIPREC	387	331	319	212
CSLL	∞	∞	∞	∞	RMM	∞	∞	∞	∞
PCBB	99630	940338	83136	∞	QSORT	∞	∞	∞	∞
PHIL	∞	∞	∞	∞	APSP	∞	∞	∞	∞
SBAR	∞	∞	∞	∞	SOR	∞	∞	∞	∞
CIG	1337	3916	2589	1230	ASTAR	∞	∞	∞	∞
LOGM	∞	∞	∞	∞	NQN	∞	∞	∞	∞

Table 5.5.: Running times of MacroConc analyses with different mailbox abstractions. The timings are expressed in milliseconds and represent the average of 20 runs after 10 warmup runs. ∞ denotes that the analysis exceeded the allocated time budget of 30 minutes.

5.4. Conclusion

In this chapter, we presented five sound abstractions for mailboxes of actor programs. The set mailbox abstraction used in the analyses presented in Chapters 2, 4 and 6 is a coarse abstraction that preserves no information about the ordering nor about the multiplicity of the messages contained in the mailbox. We have shown the importance of preserving such information in order to analyze actor programs for specific properties, such as verification of absence of errors or inference of mailbox bounds. We categorized the abstractions according to whether they preserve ordering information and according to whether they preserve multiplicity information. We formalized each of the abstractions, which are proven sound. We assessed the impact of the different abstractions on the precision and running time of an analysis resulting from the application of MacroConc. Through a benchmark suite that includes programs featuring unreachable errors and bounded mailboxes, we have shown that mailbox abstractions preserving multiplicity and ordering information are necessary in order to verify such benchmarks. We also observed an increased precision at the level of mailboxes for abstractions that preserve this information. The finite list and the graph abstractions yield the highest precision, and the finite multiset abstraction yields a significantly better precision than a coarse set abstraction. To assess the applicability of these abstractions on real-world programs, we measured the running time of the analysis with each of these abstractions on our full benchmark suite. Compared to the coarse set abstraction used in the other

chapters of this dissertation, these new abstractions do not result in more benchmarks being analyzed, but the cost in scalability is low for a high improvement in precision. Small benchmark programs even profit from an improved analysis time.

6

MODCONC: DESIGNING MODULAR ANALYSES

Chapter 4 introduced the MacroConc analysis design method, which incorporates macro-stepping semantics for improved scalability into AAM-style analyses without compromising on their precision. Although the resulting analyses perform orders of magnitude better than the analyses of Chapter 2, they still fail to analyze the entirety of our benchmark suite, both for actor programs and for multi-threaded programs. This is because, even though MacroConc reduces the number of interleavings that an analysis has to consider, it does not solve the inherent state explosion: the number of interleavings to explore remains exponential in the worst case.

In this chapter, we demonstrate that scalability can be achieved through *modular* analysis, as introduced by Cousot and Cousot (2002). We present the ModConc analysis design method, resulting in process-modular analyses for concurrent programs (Section 6.1) and discuss the soundness, complexity and termination of the resulting analyses (Section 6.2). We then apply ModConc to concurrent actor programs (Section 6.3) and to shared-memory multi-threaded programs (Section 6.4). Finally, we evaluate the resulting analyses empirically in terms of running time, scalability and performance (Section 6.5).

6.1. Modular Abstract Interpretation of Concurrent Programs

The analyses presented in Chapter 2 explore all possible process interleavings of the transition relation explicitly, which hampers their scalability. While only exploring a sound subset of the interleavings—as done by the introduction of macro-stepping semantics into the analysis (Chapter 4)—leads to an improvement in running time, the worst-case time complexity of the resulting analyses remains exponential. In practice not all of our benchmark programs can be analyzed within the time limit. We propose process-modular analyses as an alternative, where process interleavings are not modeled explicitly.

The notion of modular analysis was introduced by Cousot and Cousot (2002). A modular analysis treats the behavior of every *component* (in our case, each process) in isolation. The analysis computes the *interferences* (in our case, communication effects) of a component with other components, which are then reconsidered for analysis until a fixed point is reached. To this, we add the notion that the analysis of a component can discover new components that are created dynamically and need to be considered for analysis. The core insight enabling modular analysis for concurrent programs is that the concurrent behavior of a process is defined by its code, known at creation time, and the communication effects involving this process, discovered by analyzing other processes. Other factors such as user input and random values are dealt with by the abstractions on values of the sequential subset.

Our ModConc design method relies on the same transition relation as the analyses of Chapter 2, but consists of two clearly distinct and alternating phases: an intra-process analysis phase and an inter-process analysis phase.

1. In the *intra-process analysis* phase, a single process is analyzed in isolation until a fixed point is reached. This phase infers the processes that are created and the communication effects that are generated by the analyzed process. The intra-process analysis is based on a modified version of the concurrent semantics, replacing concurrent operations by operations that denote the corresponding generated effects but otherwise do not modify the analysis state of other processes in any way. This is in contrast with the analyses presented in Chapters 2 and 4, which keep track of the state of all processes and may step any of the running processes at any given point, immediately reflecting the interferences between the processes.

2. The *inter-process analysis* phase relies on the effects accumulated during the intra-process analysis to update the global analysis state. After computing the set of processes that interfere with the analyzed processes, the inter-process analysis runs additional intra-process analyses for processes that were created or impacted by these state updates. When no new interprocess communication effects are discovered, a sound over-approximation of the behavior of all processes in the program has been reached.

The result is a modular whole-program analysis for concurrent programs that infers the set of all running processes, their reachable states and the communication effects

performed, and this in a scalable manner. The resulting analyses, like any sound all-interleavings analysis, take into account all communication effects that may occur during program execution (no information is "lost") but, unlike all-interleavings analyses, do not explicitly explore all possible process interleavings. This information is over-approximated, but remains implicitly accounted for. For this reason, the resulting analyses scale with respect to process creation and interprocess communication.

We illustrate the notion of a process-modular analysis at a high level. Consider the factorial example implemented with actors, introduced in Section 2.2 and repeated in Listing 6.1. A process-modular analysis analyzing this program performs the following iterations.

1. The first iteration of the process-modular analysis analyzes the main process. This is done by an intra-process analysis that relies on a sequentialized transition relation and that collects the set of processes created and messages sent. In this case, the intra-process analysis detects two created actors: an actor `fact` with behavior `fact-actor` created at line 21, and an actor `displayer` with behavior `displayer-actor` created at line 22. The intra-process analysis also detects that the main process sends a message `compute`$(\mathit{Int}, \texttt{displayer})$ to the actor `fact` at line 23.

2. The second iteration analyzes the `fact` actor, with a mailbox containing the `compute`$(\mathit{Int}, \texttt{displayer})$ message. It detects one created actor `c` with behavior `customer-actor`$(\mathit{Int}, \texttt{displayer})$ created at line 6, and two sent messages: a `result`(Int) message sent to actor `displayer` at line 5, and a `compute`$(\mathit{Int}, \texttt{c})$ sent to actor `fact` at line 8.

3. The third iteration analyzes the `fact` actor with an updated mailbox, and the `displayer` actor. For the `fact` actor, one new created actor is detected: actor `c` with behavior `customer-actor`$(\mathit{Int}, \texttt{c})$ created at line 6 (note the change in its second state variable compared to the created actor detected in the previous iteration, and the circularity in its definition that results from the abstraction of the semantics), and two sent messages are detected: message `result`(Int) is sent to actor `c` at line 5, and message `compute`$(\mathit{Int}, \texttt{c})$ is sent to actor `fact` at line 8. The intra-process analysis of the `displayer` actor analyzes this actor with a mailbox containing the `result`(Int) message, and does not detect created actors nor sent messages.

4. The fourth iteration analyzes the `c` actor with behavior `customer-actor`$(\mathit{Int}, \{\texttt{displayer}, \texttt{c}\})$ (note the approximation of its second state variable). It detect two possible sent messages: the message `result`(Int) can be sent to both actor `displayer` and to actor `c`. At this point, the sent messages that are detected have already been taken into account by the analysis, which has therefore reached a fixed point and terminates.

Unlike MacroConc, ModConc is directly applied to the *abstract semantics* of a concurrent programming language. This is because, unlike the analyses presented in Chapters 2

```
1 (define fact-actor
2   (actor ()
3     (compute (n customer)
4       (if (= n 0)
5         (send customer result 1)
6         (let ((c (create customer-actor
7                          n customer)))
8           (send self compute (- n 1) c)))
9       (become fact-actor))))
10 (define customer-actor
11   (actor (n customer)
12     (result (k)
13       (send customer result (* n k))
14       (become customer-actor n customer))))
15 (define displayer-actor
16   (actor ()
17     (result (v)
18       (display v)
19       (become displayer-actor))))
20
21 (define fact (create fact-actor))
22 (define displayer (create displayer-actor))
23 (send fact compute (read-integer) displayer)
```

Listing 6.1: Program computing the factorial of a user-given number with actors, adapted from Agha (1986).

and 4, the analyses presented here over-approximate the process interleavings. MacroConc abstracts a refinement of the concrete semantics of a concurrent programming language, while ModConc abstracts further over an initial abstraction such as the ones presented in Chapter 2. We prove that applications of this further abstraction to λ_α and to λ_τ are sound (Theorems 16 and 18).

ModConc comes at a cost in terms of precision. Properties that rely on ordering or multiplicity information may not be verifiable using the results of this process-modular analysis. This is the case for the mailbox bounds analysis of Chapter 5. However, local process properties such as communication effects can still be inferred with high precision, as demonstrated in our evaluation of Section 6.5.

Applying ModConc to concurrent programs is a four-step process.

1. Specifying an *operational semantics* for the input language featuring concurrency, modeled by a transition relation. This corresponds to the semantics given in Chapter 2.

2. Modifying the operational semantics into a *sequentialized transition relation* annotated with communication effects.

3. Specifying an *intra-process analysis* that computes states reachable and the communication effects produced by a given process, based on the sequentialized semantics.

4. Specifying an *inter-process analysis* that runs intra-process analyses on newly discovered processes and on processes affected by communication effects, until no new communication effects are inferred.

The remainder of this section details each of these steps.

6.1.1. Step 1: Definition of the Abstract Operational Semantics for the Input Language

In order to apply ModConc, the operational semantics of a concurrent programming language needs to be defined through a transition that specifies how the program state (i.e., the state of its set of processes, its value store, and its continuation store) evolves in a small-step fashion. Sections 2.2 and 2.3 respectively presented such a transition relation for λ_α and for λ_τ. These relations define two kinds of transitions.

1. *Sequential transitions* model sequential operations affecting a single process. These correspond to the transitions defined by on the transition relation of the common base language λ_0 ($\hookrightarrow$). They generate no communication effects.

2. *Concurrent transitions* define the semantics for concurrent operations that affect more than one process or that affect the global state of the system. Concurrent transitions include for example the transitions for the `create` and `send` constructs in λ_α (see Section 2.2), and the transitions for the `spawn` and `join` constructs in λ_τ (see Section 2.3).

6.1.2. Step 2: Definition of the Sequentialized Transition Relation

The second step in an application of ModConc is to construct a sequentialized version of the above semantics that acts on a single process. This semantics honors the existing sequential transitions, but replaces the transitions for concurrent constructs by transitions that generate communication effects that don't apply them to the global analysis state yet. This results in an intra-process transition relation annotated with a set of communication effects, which is not empty for concurrent operations. These effects are defined formally for the actor-based and multi-threaded languages in Sections 2.2 and 2.3.

We denote this intra-process transition relation as $\rightsquigarrow_{\hat{p}}$. Note that it only acts on a single process state $\hat{\varsigma}$, whereas the transition relations in the analyses of Chapters 2 and 4 act on a map of processes π. This intra-process transition relation may also be parameterized by values, written as subscript. This allows a view on the global state, e.g. to retrieve messages from a mailbox or inspect return values of other threads.

6.1.3. Step 3: Definition of the Intra-Process Analysis

The intra-process analysis analyzes a process under given assumptions and infers the communication effects that can be generated by the analyzed process. The assumptions for the intra-process analysis are encoded as parameters and generally include the global value store and global continuation store, as well as the values returned by other threads in multi-threaded programs and the actor mailboxes in actor-based programs.

The intra-process analysis explores all possible behaviors of the process under analysis under the given assumptions, and infers the communication effects as well as potential changes to global program components such as the value store and continuation store.

The intra-process analysis is formalized as the fixed point of a parameterized transfer function *Intra*-$\widehat{\mathcal{H}}$, that applies the transition relation resulting from step 2. The fixed point of this transfer function is a sound over-approximation of all possible executions of the process under analysis, under the given assumptions. It provides the information that needs to be propagated to the global analysis state.

6.1.4. Step 4: Definition of the Inter-Process Analysis

The inter-process analysis collects the communication effects inferred by the intra-process analyses for a set of processes, and computes the next set of processes that require intra-process analysis. These are the newly created processes and the processes that depend on the inferred communication effects.

For example, if a thread t_1 is joining a thread t_2, and the intra-process analysis of thread t_2 has determined a change in the return value of this process, then the intra-process analysis of thread t_1 must be performed again to account for this new information. Similarly, based on the communication effect determined by an intra-process analysis when an actor a_1 sends a message to an actor a_2, the inter-process analysis performs a subsequent intra-process analysis on actor a_2.

As the execution of a concurrent program starts from its main process, the inter-process analysis starts by performing an intra-process analysis of this main process. The inter-process analysis terminates when no new communication effects can be inferred, and results in a sound over-approximation of all possible behaviors of the program under analysis.

The inter-process analysis is formalized as the fixed point of a transfer function *Inter*-$\widehat{\mathcal{H}}_e$, parameterized by the program e under analysis. The domain of the inter-process transfer function contains a global process map $\hat{\pi}$ mapping process identifiers to their initial process state and to a set of reachable states as computed by the intra-process analysis. It also contains global program components such as the value store $\hat{\sigma}$ and continuation store $\widehat{\Xi}$.

6.2. Properties of a Process-Modular Analysis

We discuss here important properties that analyses resulting from the application of ModConc exhibit, and explain how this is ensured.

6.2.1. Termination

In order for an analysis resulting from the application of ModConc to terminate, both the intra-process and inter-process analyses, defined as fixed-point computations, have to terminate. This can be proven by showing that the transfer function of each analysis is monotone, and that these functions act over a finite abstracted state space. Termination of these analyses is then ensured by Tarski's fixed-point theorem (Tarski, 1955). We prove termination of analyses resulting from the application of ModConc to λ_α in Section 6.3.5 and to λ_τ in Section 6.4.5.

6.2.2. Soundness

Analyses resulting from the application of ModConc over-approximate the process interleavings, without exploring them explicitly. All interleavings between the explored processes are deemed reachable. Similarly as for analyses resulting from the application of MacroConc (Section 4.2), it can be shown that for verifying particular program properties, a process-modular analysis soundly approximates the result of analyses relying on all-interleavings semantics. In particular, the application of ModConc to λ_α and to λ_τ is proven sound (Theorems 16 and 18 in Sections 6.3 and 6.4): all process states (ς) explored by an all-interleavings analysis are also accounted for by a process-modular analysis. This suffices to derive sound analyses such as communication topology analyses (Colby, 1995; Martel and Gengler, 2000). We prove the soundness of analyses resulting from the application of ModConc to λ_α in Section 6.3.5 and to λ_τ in Section 6.4.5.

6.2.3. Complexity

A process-modular analysis designed according to the ModConc design method scales *linearly* in terms of the total number of communication effects: for a given program size, the analysis time increases linearly in the number of abstract processes created by the program under analysis and in other communication effects such as the number of abstract messages sent in actor programs or write operations on shared variables in thread programs. This is because the inter-process analysis will at most have to recompute an intra-process analysis a number of times proportional to the number of communication effects. As the number of abstract processes increases, the running time of the analysis increases linearly.

The complexity of the inter-process analysis therefore adds a polynomial factor to the complexity of the sequential analysis. If the sequential analysis is polynomial, the process-modular concurrent analysis remains polynomial, unlike a non-modular analysis which becomes exponential due to the non-determinism in process interleavings. For example, the intra-process analyses presented in this chapter exhibit the same characteristics as the analysis for λ_0, and have a complexity of $\mathcal{O}(|Exp|^3)$. The inter-process analysis will at most run one intra-process analysis for each detected communication effect, i.e., its worst-time complexity is of $\mathcal{O}(|\widehat{Effect}| \times |Exp|^3)$. We demonstrate the scalability of the application of ModConc to λ_α and λ_τ in Section 6.5.4.

6.2.4. Precision

Whereas analyses resulting from the application of the MacroConc design method (Chapter 4) preserve the same precision as the analyses resulting from a naive application of AAM to concurrent programs (Chapter 2), process-modular analyses resulting from the application of ModConc trade off precision to gain in scalability. Information about the order in which the communication effects occur and about their multiplicity is lost, because they are stored in sets by the intra-process analysis. ModConc therefore cannot be used for analyses such as the mailbox bound analysis of Chapter 4. We leave

open the question of whether this information may be preserved, identifying it as future work in Chapter 7. We study the precision of analyses resulting from the application of ModConc in Section 6.5.3.

6.3. Application of ModConc to λ_α

Next, we apply the ModConc design method to λ_α concurrent actor programs. We first present the sequentialized transition relation, derived from the abstract transition relation of λ_α presented in Section 2.2 . We then present the intra-process analysis which computes, for each abstract process, the set of created processes and the set of messages sent. Finally, we present the inter-process analysis, which acts on global analysis states composed of, for each abstract actor, the intra-process analysis state, a set of reachable states, and an abstraction of the mailbox. Relying on the information inferred by intra-process analysis, the inter-process analysis reconsiders for intra-process analysis the set of newly created processes, and the processes impacted by the set of messages sent.

6.3.1. Step 1 of ModConc for λ_α: Definition of the Abstract Operational Semantics

The abstraction of the operational semantics of λ_α on which we rely for the application of ModConc to λ_α is the abstraction presented in Section 2.2. It results from a naive application of AAM and forms the starting point for the application of ModConc to λ_α.

6.3.2. Step 2 of ModConc for λ_α: Definition of the Sequentialized Transition Relation

The sequentialized transition relation has the same structure as the transition relation of the abstract operational semantics of λ_α (step 1). The three main differences are the following.

1. Rather than acting on global states, represented as process maps $\hat{\pi}$, the transition relation for ModConc acts on the local state of a single abstract actor $\hat{\varsigma}$.
2. The transition relation for ModConc is parameterized by the mailbox of the actor under analysis, so that messages can be extracted from this mailbox.
3. Instead of applying the effect of concurrent operations to the process map, the transition relation records generated communication effects. The changes are applied to the local state of the actor, but not to the state of other actors.

Transitions are therefore written $\hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p},\widehat{mb}} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'$.

We follow the same structure of presentation as in Chapter 2: we introduce the state space of the sequentialized transition relation first, and then present the transition relation rules for sequential transitions, for actor management, and for messages. Allocation of addresses, allocation of process identifiers, and atomic evaluation remains

unchanged and we therefore do not repeat their definitions, which can be found in Chapter 2 (Figures 2.12, 2.32 and 2.36).

State Space of the Sequentialized Transition Relation

Figure 6.1 depicts the state space for the sequentialized transition relation of λ_α. The state space consists of actor states containing a control component ($\hat{c}$), a behavior ($\hat{b}$), and a continuation address ($\widehat{k}$). Abstract mailboxes are represented as sets of messages. We refer to the abstract state space of Chapter 2 for the components that are not defined here, such as the stores ($\hat{\sigma}$) and the environments ($\hat{\rho}$).

λ_α

$$\hat{\varsigma} \in \widehat{\Sigma} = \widehat{Control} \times \widehat{Beh} \times \widehat{KAddr}$$
$$\hat{c} \in \widehat{Control} ::= \mathbf{ev}(e, \hat{\rho}) \mid \mathbf{val}(\hat{v}) \mid \mathbf{wait}$$
$$\hat{v} \in \widehat{Val} ::= \mathbf{clo}(lam, \hat{\rho}) \mid \mathbf{pid}(\hat{p}) \mid \mathbf{actor}(act, \hat{\rho})$$
$$\hat{b} \in \widehat{Beh} ::= \mathbf{beh}(act, \hat{\rho}) \mid \mathbf{main}$$
$$\hat{m} \in \widehat{Message} = Tag \times \widehat{Val}$$
$$\widehat{mb} \in \widehat{Mbox} = \mathcal{P}(\widehat{Message})$$
$$\hat{p} \in \widehat{PID} \text{ a finite set of process identifiers}$$

Figure 6.1.: State space of the sequentialized transition relation for λ_α.

Sequentialized Transition Relation

The sequentialized transition relation is written $\hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p}, \widehat{mb}} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'$ and acts on actor states, value stores and continuation stores, for an actor with process identifier $\hat{p}$ and mailbox $\widehat{mb}$. Transition relations may carry a communication effect. Communication effects that only affect the state of the actor transitioning or the stores are directly applied, but communication effects that affect the state of other actors are not applied, as the transition relation does not act on process maps.

Sequential transitions. Figure 6.2 depicts the transition rule for sequential transitions. If a sequential transition may be performed in a process, the process can transition (rule Seq).

λ_α

$$\frac{\hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \hookrightarrow \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'}{\hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p}, \widehat{mb}} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'} \textsc{Seq}$$

Figure 6.2.: Sequential transition rule for λ_α.

Actor management transitions. Figure 6.3 depicts the transition rules for actor management. The actor creation rule Create steps the process performing the creation into a value state with the process identifier of the created process as value. The initial state of the created actor is constructed ($\hat{\varsigma}'$), but is not reflected in the resulting state, as this is handled by the inter-process analysis. A creation effect is generated and contains the process identifier of the created actor ($\hat{p}'$), as well as its initial state ($\hat{\varsigma}'$). Note that the second argument given to the process allocation function $\widehat{palloc}$ is an empty process map ([]). This is because this sequentialized transition relation has no process map, and can only be used in an abstract setting. We therefore require a $\widehat{palloc}$ function that ignores the value of its second parameter. This is the case for the abstract process identifier allocation function defined in Figure 2.36.

The rule Become only affects the actor executing the `become` statement, and all changes are therefore reflected in the actor state. This rule generates a **b** effect, used only for informational purposes.

λ_α

$$\frac{\begin{array}{c}\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{actor}(act, \hat{\rho}') \\ \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v} \qquad \hat{\varsigma}' = \langle \mathbf{wait}, \mathbf{beh}(act, \hat{\rho}'[x \mapsto \hat{a}]), \widehat{k}_0 \rangle \\ \hat{a} = \widehat{alloc}(x, \hat{\sigma}) \qquad \hat{p}' = \widehat{palloc}(\hat{\varsigma}', []) \qquad x = \textsc{var}(act)\end{array}}{\begin{array}{c}\langle \mathbf{ev}(\texttt{(create } ae\ ae'\texttt{)}, \hat{\rho}), \hat{b}, \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{c}(\hat{p}', \hat{\varsigma}')}{\rightsquigarrow}_{\hat{p}, \widehat{mb}} \\ \langle \mathbf{val}(\mathbf{pid}(\hat{p}')), \hat{b}, \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}\end{array}} \textsc{Create}$$

$$\frac{\begin{array}{c}\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{actor}(act, \hat{\rho}') \qquad \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v} \\ \hat{b}' = \mathbf{beh}(act, \hat{\rho}'[x \mapsto \hat{a}]) \qquad x = \textsc{var}(act) \qquad \hat{a} = \widehat{alloc}(x, \hat{\sigma})\end{array}}{\langle \mathbf{ev}(\texttt{(become } ae\ ae'\texttt{)}, \hat{\rho}), \hat{b}, \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{b}(\hat{b}', \hat{v})}{\rightsquigarrow}_{\hat{p}, \widehat{mb}} \langle \mathbf{wait}, \hat{b}', \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}} \textsc{Become}$$

Figure 6.3.: Actor management transitions for λ_α.

Messages transitions. Figure 6.4 depicts the transition rules for messages in λ_α. The message sending rule Send steps the actor that sends a message into a value state and

generates a **snd** communication effect, without storing the message sent into the mailbox of the target actor. The information provided by the communication effect is used by the intra-actor analysis.

The message processing rule Process generates a **prc** effect containing the message extracted from the mailbox. The actor steps into the body of the message handler corresponding to the received message.

λ_α

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{pid}(\hat{p}') \qquad \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v}}{\langle \mathbf{ev}(\texttt{(send } ae\ t\ ae'\texttt{)}, \hat{\rho}), \hat{b}, \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{snd}(\hat{p}', t, \hat{v})}_{\hat{p}, \widehat{mb}} \langle \mathbf{val}(\hat{v}), \hat{b}, \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{Send}$$

$$\frac{(t, \hat{v}) \in \widehat{mb} \qquad (y, e) = \textsc{handler}(act, t) \qquad \hat{a} = \widehat{alloc}(y, \hat{\sigma})}{\begin{array}{c}\langle \mathbf{wait}, \mathbf{beh}(act, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{prc}(t, \hat{v})}_{\hat{p}, \widehat{mb}} \\ \langle \mathbf{ev}(e, \hat{\rho}[y \mapsto \hat{a}]), \mathbf{beh}(act, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}\end{array}} \textsc{Process}$$

Figure 6.4.: Message transitions for λ_α.

6.3.3. Step 3 of ModConc for λ_α: Definition of the Intra-Process Analysis

The intra-process analysis for λ_α analyzes the behavior of a single actor by performing a fixed-point computation of a transfer function $\mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{\hat{p}, \hat{\varsigma}_0, \hat{\sigma}_0, \widehat{\Xi}_0, \widehat{mb}}$ where $\hat{\varsigma}_0$ is the initial state of the actor under analysis, $\hat{\sigma}_0$ and $\widehat{\Xi}_0$ are the global stores before analysis, and $\widehat{mb}$ is an approximation of the mailbox of the actor under analysis.

State Space of the Intra-Process Analysis

Figure 6.5 depicts the state space of the intra-process analysis. The intra-process analysis operates on *intra-states* ($\widehat{\mathit{IntraState}}$), which consist of a set of actor states ($\mathcal{P}(\widehat{\Sigma})$), a value store ($\widehat{\mathit{Store}}$), a continuation store ($\widehat{\mathit{KStore}}$), a set of created actors ($\mathcal{P}(\widehat{\mathit{Created}})$), and a set of messages sent ($\mathcal{P}(\widehat{\mathit{Sent}})$). Created actors are represented as a pair of their process identifier and their initial actor state, i.e., the state at which they start their execution. Messages sent are represented as a tuple of the process identifier to which the message is addressed, and the message sent. A mailbox is represented as a set of messages, which themselves are pairs of tags and values.

λ_α

$$\begin{aligned}\widehat{IntraState} &= \mathcal{P}(\widehat{\Sigma}) \times \widehat{Store} \times \widehat{KStore} \\ &\quad \times \mathcal{P}(\widehat{Created}) \times \mathcal{P}(\widehat{Sent}) \\ \widehat{Created} &= \widehat{PID} \times \widehat{\Sigma} \\ \widehat{Sent} &= \widehat{PID} \times \widehat{Message}\end{aligned}$$

Figure 6.5.: State space for the intra-process analysis for λ_α.

Transfer Function of the Intra-Process Analysis

Figure 6.6 depicts the transfer function of the intra-process analysis, $Intra\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\widehat{mb}} : \widehat{IntraState} \to \widehat{IntraState}$. This transfer function performs the following operations.

1. The initial state and stores are part of the intra-process analysis state. The initial sets of created processes and sent messages are empty.
2. Effect-less transitions are taken without generating any communication effect.
3. Transitions that process a message or change the behavior of the actor are taken without adding information to the set of created actors and messages sent.
4. Transitions that create a process add an element to the set of created processes.
5. Transitions that receive a message extract the message content from the mailbox.

λ_α

$$\mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\widehat{mb}}(\langle S,\hat{\sigma},\widehat{\Xi},C,M\rangle) = \langle\{\hat{\varsigma}_0\},\hat{\sigma}_0,\widehat{\Xi}_0,\varnothing,\varnothing\rangle \tag{1}$$

$$\sqcup \bigsqcup_{\hat{\varsigma}\in S} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\varnothing\rangle \tag{2}$$
$$\hat{\varsigma},\hat{\sigma},\widehat{\Xi} \rightsquigarrow_{\hat{p},\widehat{mb}} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'$$

$$\sqcup \bigsqcup_{\hat{\varsigma}\in S} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\varnothing\rangle \tag{3}$$
$$\hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\widehat{eff}}{\rightsquigarrow}_{\hat{p},\widehat{mb}} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'$$
$$\widehat{eff}\in\left\{\mathbf{b}(\hat{b},\hat{v}),\mathbf{prc}(t,\hat{v})\right\}$$

$$\sqcup \bigsqcup_{\hat{\varsigma}\in S} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\{(\hat{p}',\hat{\varsigma}_2)\},\varnothing\rangle \tag{4}$$
$$\hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\mathbf{c}(\hat{p}',\hat{\varsigma}_2)}{\rightsquigarrow}_{\hat{p},\widehat{mb}} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'$$

$$\sqcup \bigsqcup_{\hat{\varsigma}\in S} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\{(\hat{p}',(t,\hat{v}))\}\rangle \tag{5}$$
$$\hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\mathbf{snd}(\hat{p}',t,\hat{v})}{\rightsquigarrow}_{\hat{p},\widehat{mb}} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'$$

Figure 6.6.: Transfer function for the intra-process analysis for λ_α.

6.3.4. Step 4 of MODCONC for λ_α: Definition of the Inter-Process Analysis

The inter-process analysis is defined as the fixed-point computation of a transfer function $\mathit{Inter}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_e$ where e is the program under analysis. The result of the inter-process analysis is a process map that maps each abstract process to the result of its intra-process analysis, its initial actor state, and its mailbox.

State Space of the Inter-Process Analysis

Figure 6.7 depicts the process map $\hat{\pi}$ produced by the inter-process analysis of λ_α.

λ_α

$$\hat{\pi} \in \widehat{\Pi} = \widehat{PID} \to (\widehat{IntraState} \times \widehat{\Sigma} \times \widehat{Mbox})$$

Figure 6.7.: State space for the inter-process analysis of λ_α.

Transfer Function of the Inter-Process Analysis

The transfer function relies on the following auxiliary functions, defined in Figure 6.8.

- $\mathit{explore} : \widehat{\mathit{PID}} \times \widehat{\Sigma} \times \widehat{\mathit{Mbox}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}} \to \widehat{\Pi} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}}$ performs an intra-process analysis on the actor with the process identifier $\hat{p}$, initial state $\hat{\varsigma}$, mailbox $\widehat{mb}$, using value store $\hat{\sigma}$ and continuation store Ξ. It returns a process map, value store and continuation store containing the information resulting from the intra-process analysis.
- $\mathit{created} : \widehat{\Pi} \to \mathcal{P}(\widehat{\mathit{PID}} \times \widehat{\Sigma})$ returns the set of all created actors computed by the analysis, described by their process identifier and their initial actor state.
- $\mathit{sent} : \widehat{\Pi} \times \widehat{\mathit{PID}} \to \mathcal{P}(\widehat{\mathit{Message}})$ returns the set of messages sent to actor with process identifier $\hat{p}$.

λ_α

$$\mathit{explore}(\hat{p}, \hat{\varsigma}, \widehat{mb}, \hat{\sigma}, \widehat{\Xi}) = \langle [\hat{p} \mapsto (s, \hat{\varsigma}, \widehat{mb})], \hat{\sigma}', \widehat{\Xi}' \rangle$$
$$\text{where } s = \mathrm{lfp}(\mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{\hat{p},\hat{\varsigma},\hat{\sigma},\widehat{\Xi},\widehat{mb}}) \text{ and } (\_, \hat{\sigma}', \widehat{\Xi}', \_, \_) = s$$

$$\mathit{created}(\hat{\pi}) = \bigcup_{\substack{\hat{p} \in \mathrm{dom}(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((\_,\_,\_,C,\_),\_,\_)}} C \qquad \mathit{sent}(\hat{\pi}, \hat{p}) = \bigcup_{\substack{\hat{p} \in \mathrm{dom}(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((\_,\_,\_,\_,M),\_,\_) \\ (\hat{p},\hat{m}) \in M}} \{\hat{m}\}$$

Figure 6.8.: Auxiliary functions used in the inter-process analysis for λ_α.

Figure 6.9 depicts the inter-process analysis transfer function $\mathit{Inter}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_e : \widehat{\Pi} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}} \to \widehat{\Pi} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}}$, which performs the following operations.

1. The main process has an empty mailbox and is analyzed with the intra-process analysis.
2. Each process to which a message is sent—as extracted from the results of previous intra-process analyses by the *sent* function—is re-analyzed with an updated mailbox.
3. Each process that has been created—as extracted from the result of previous intra-process analyses by the *created* function—is analyzed, starting with an empty mailbox.

λ_α

$$\mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\alpha}(\langle \hat{\pi}, \hat{\sigma}, \widehat{\Xi}\rangle) = \mathit{explore}(\hat{p}_0, \langle \mathbf{ev}(e, []), \mathbf{main}(e), \widehat{k}_0\rangle, \varnothing, \hat{\sigma}, \widehat{\Xi}) \quad (1)$$

$$\sqcup \bigsqcup_{\substack{\hat{\pi}(\hat{p})=(\_,\hat{\varsigma},\widehat{mb}) \\ \widehat{mb}'=\widehat{mb}\cup \mathit{sent}(\hat{\pi},\hat{p})}} \mathit{explore}(\hat{p}, \hat{\varsigma}, \widehat{mb}', \hat{\sigma}, \widehat{\Xi}) \quad (2)$$

$$\sqcup \bigsqcup_{(\hat{p},\hat{\varsigma})\in \mathit{created}(\hat{\pi})} \mathit{explore}(\hat{p}, \hat{\varsigma}, \varnothing, \hat{\sigma}, \widehat{\Xi}) \quad (3)$$

Figure 6.9.: Inter-process analysis transfer function for λ_α.

Complexity

The inter-process analysis may reconsider an actor for analysis for every message sent and for every actor created. As the process allocation strategy of Figure 2.26 represents the process identifier of an abstract actor by an abstract behavior, the number of abstract actor is bounded by the number of abstract behaviors. The worst-case time complexity of the analysis is therefore of $\mathcal{O}((b+m) \times |Exp|^3)$, where b is the number of abstract behaviors, m is the number of abstract messages sent, and $|Exp|^3$ comes from the worst-case time complexity of the intra-actor analysis, which has the same complexity as a sequential analysis.

6.3.5. Soundness and Termination

Theorems 16 and 17 state that the analysis described by the inter-process analysis terminates and is sound, two crucial properties for a static analysis.

Theorem 16 (Soundness)**.** *The result of* $\mathrm{lfp}(\mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\alpha})$ *is a sound over-approximation of* $\mathrm{lfp}(\mathcal{F}_e^{\lambda_\alpha})$.

Proof. The proof is detailed in Appendix B.4.1. The idea of this proof is the following. We show by a case analysis on the transition rules that the intra-process analysis is a sound over-approximation of the transfer function $\widehat{\mathcal{F}}^{\lambda_\alpha}$ restricted to states reachable with transition on the same process $\hat{p}$ from the initial state $\langle [\hat{p} \mapsto \left\{(\hat{\varsigma}_0, \widehat{mb})\right\}], \hat{\sigma}_0, \widehat{\Xi}_0\rangle$. As the inter-process analysis considers every newly created actor and every actor affected by a message send, the analysis will eventually have analyzed all running actors with over-approximation of their mailbox and stores. □

Theorem 17 (Termination)**.** *The computation of* $\mathrm{lfp}(\mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\alpha})$ *always terminates.*

Proof. The proof is detailed in Appendix B.4.1 and follows the same structure as previous termination proofs. By proving that the abstract state space is finite, and that both the transfer function of the intra-process analysis and the transfer function of the inter-process analysis are monotone, termination is ensured. □

6.4. Application of ModConc to λ_τ

We apply the ModConc design method to λ_τ programs. These programs feature shared-memory concurrency through multi-threading and support mutable references and locks. We follow the same structure of presentation as for the application of ModConc to λ_α programs: we present the sequentialized transition relation, the intra-process analysis, the inter-process analysis and conclude by proving soundness and termination.

6.4.1. Step 1 of ModConc for λ_τ: Definition of the Abstract Operational Semantics

The abstract operational semantics of λ_τ, defined by the concurrent transition relation $\rightsquigarrow_{\hat{p}}$ in Section 2.3, forms the starting point for the application of ModConc to λ_τ.

6.4.2. Step 2 of ModConc for λ_τ: Definition of the Sequentialized Transition Relation

The transition relation of λ_τ undergoes similar structural changes as the transition relation of λ_α, which are the following.

1. The transition relation acts on local thread states ($\hat{\varsigma}$) rather than on global process maps ($\hat{\pi}$).
2. The transition relation is parameterized by a *join store* ($\hat{J}$) that contains information about the return values of threads that have terminated their execution.
3. The communication effects annotate the transitions of λ_τ, and do not impact the global state of the program.

Transitions are therefore written $\hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p},\hat{J}} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'$. We introduce the state space of the sequentialized transition first, and then present the transition relation for sequential transitions, for thread management, for references and for locks. We do not repeat the definitions for the allocation function for abstract addresses and for abstract process identifiers, as we rely on the abstract definitions presented in Chapter 2, respectively Figures 2.12 and 2.58.

State Space of the Sequentialized Transition Relation

Figure 6.10 depicts the abstract state space of the sequentialized transition relation. It remains the same as the one introduced in Section 2.3: a process state is a pair of a control component ($\hat{c}$) and a continuation address (k), and process identifiers (**pid**), references (**ref**) and locks (**lock**) are first-class values.

λ_τ

$$\hat{\varsigma} \in \widehat{\Sigma} = \widehat{Control} \times \widehat{KAddr}$$
$$\hat{v} \in \widehat{Val} ::= \ldots$$
$$\mid \mathbf{pid}(\hat{p}) \mid \mathbf{ref}(\hat{a})$$
$$\mid \mathbf{lock}(\hat{a}) \mid \mathbf{locked}(\hat{p}) \mid \mathbf{unlocked}$$
$$\hat{J} \in \widehat{JoinStore} = \widehat{PID} \to \mathcal{P}(\widehat{Val})$$
$$\hat{p} \in \widehat{PID} \text{ a finite set of process identifiers}$$

Figure 6.10.: State space for the sequentialized transition relation for λ_τ.

Sequentialized Transition Relation

The sequentialized transition relation is written $\hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p}} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'$ and may carry a communication effect. Communication effects that affect the local state of the thread performing the effect or that affect the store are applied. However, communication effects that affect the state of other threads are not applied.

Sequential transitions. Figure 6.11 depicts the transition relation rule for sequential transitions. The rule Seq generates no communication effect and relies on the sequential transition relation of λ_0 to perform a sequential step in a thread.

λ_τ

$$\frac{\hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \hookrightarrow \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'}{\hat{\varsigma}, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p},\hat{J}} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'} \textsc{Seq}$$

Figure 6.11.: Sequential transitions for λ_τ.

Thread management. Figure 6.12 depicts the transition rules for thread management. The execution of the `spawn` construct (rule Spawn) generates a creation communication effect $\mathbf{c}(\hat{p}', \hat{\varsigma}')$ where $\hat{p}'$ is the abstract process identifier of the spawned thread, and $\hat{\varsigma}'$ is the initial state of the spawned thread. Similar to the process identifier allocation for λ_α, the second argument given to $\widehat{palloc}$ is an empty process map ($[]$), because the sequentialized transition relation has access to no process map. The abstract process identifier allocation function defined in Figure 2.58 does not use its second argument and abstract process identifiers are therefore not impacted by this.

The execution of the `join` construct (rule Join) generates on a join communication effect $\mathbf{j}(\hat{p}', \hat{v})$, where $\hat{p}'$ is the abstract process identifier of the thread joined, and $\hat{v}$ is the value returned by this thread, which is extracted from the join store $\hat{J}$.

λ_τ

$$\frac{\hat{\varsigma}' = \langle \mathbf{ev}(e, \hat{\rho}), \widehat{k}_0 \rangle \qquad \hat{p}' = \widehat{palloc}(\hat{\varsigma}', [])}{\langle \mathbf{ev}(\texttt{(spawn } e\texttt{)}, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{c}(\hat{p}', \hat{\varsigma}')}_{\hat{p}, \hat{f}} \langle \mathbf{val}(\mathbf{pid}(\hat{p}')), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{Spawn}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{pid}(\hat{p}') \qquad \hat{v} \in \hat{f}(\hat{p}')}{\langle \mathbf{ev}(\texttt{(join } ae\texttt{)}, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{j}(\hat{p}', \hat{v})}_{\hat{p}, \hat{f}} \langle \mathbf{val}(\hat{v}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{Join}$$

Figure 6.12.: Transition rules for thread management in λ_τ.

References. Figure 6.13 depicts the rules for references, which are adapted from the rules presented in Chapter 2. A $\mathbf{r}(\hat{a})$ communication effect is generated when reading a reference at address $\hat{a}$ is read from, and a $\mathbf{w}(\hat{a})$ communication effect is generated when a reference at address $\hat{a}$ is written to.

λ_τ

$$\frac{\hat{a} = \widehat{alloc}(ae, \hat{\sigma}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \hat{v}}{\langle \mathbf{ev}(\texttt{(ref } ae\texttt{)}, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p}, \hat{f}} \langle \mathbf{val}(\mathbf{ref}(\hat{a})), \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}} \textsc{Ref}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{ref}(\hat{a}) \qquad \hat{v} \in \hat{\sigma}(\hat{a})}{\langle \mathbf{ev}(\texttt{(deref } ae\texttt{)}, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{r}(\hat{a})}_{\hat{p}, \hat{f}} \langle \mathbf{val}(\hat{v}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi}} \textsc{Deref}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{ref}(\hat{a}) \qquad \hat{\rho}, \hat{\sigma} \vdash ae' \mathbin{\widehat{\Downarrow}} \hat{v}}{\langle \mathbf{ev}(\texttt{(ref-set! } ae\ ae'\texttt{)}, \hat{\rho}), \widehat{k} \rangle, \hat{\sigma}, \widehat{\Xi} \xrightarrow{\mathbf{w}(\hat{a})}_{\hat{p}, \hat{f}} \langle \mathbf{val}(\hat{v}), \widehat{k} \rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}], \widehat{\Xi}} \textsc{RefSet}$$

Figure 6.13.: Transition rules for references in λ_τ.

Locks. Figure 6.14 depicts the rules for locks, which are also adapted from the rules presented in Chapter 2. The acquisition of a lock generates an $\mathbf{acq}(\hat{p}, \hat{a})$ communication effect, and the release of a lock generates a $\mathbf{rel}(\hat{p}, \hat{a})$ communication effect.

λ_τ

$$\frac{\hat{a} = \widehat{alloc}(\texttt{(new-lock)}, \hat{\sigma})}{\begin{array}{c}\langle \mathbf{ev}(\texttt{(new-lock)}, \hat{\rho}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \rightsquigarrow_{\hat{p},\hat{f}} \\ \langle \mathbf{val}(\mathbf{lock}(\hat{a})), \widehat{k}\rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\mathbf{unlocked}\}], \widehat{\Xi}\end{array}} \textsc{NewLock}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{lock}(\hat{a}) \qquad \hat{\sigma}(\hat{a}) \ni \mathbf{unlocked}}{\begin{array}{c}\langle \mathbf{ev}(\texttt{(acquire } ae\texttt{)}, \hat{\rho}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{acq}(\hat{p},\hat{a})}{\rightsquigarrow}_{\hat{p},\hat{f}} \\ \langle \mathbf{val}(\mathbf{lock}(\hat{a})), \widehat{k}\rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\mathbf{locked}(\hat{p})\}], \widehat{\Xi}\end{array}} \textsc{Acquire}$$

$$\frac{\hat{\rho}, \hat{\sigma} \vdash ae \mathbin{\widehat{\Downarrow}} \mathbf{lock}(\hat{a}) \qquad \hat{\sigma}(\hat{a}) \ni \mathbf{locked}(\hat{p})}{\begin{array}{c}\langle \mathbf{ev}(\texttt{(release } ae\texttt{)}, \hat{\rho}), \widehat{k}\rangle, \hat{\sigma}, \widehat{\Xi} \overset{\mathbf{rel}(\hat{p},\hat{a})}{\rightsquigarrow}_{\hat{p},\hat{f}} \\ \langle \mathbf{val}(\mathbf{lock}(\hat{a})), \widehat{k}\rangle, \hat{\sigma} \sqcup [\hat{a} \mapsto \{\mathbf{unlocked}\}], \widehat{\Xi}\end{array}} \textsc{Release}$$

Figure 6.14.: Transition rules for locks for λ_τ.

6.4.3. Step 3 of ModConc for λ_τ: Definition of the Intra-Process Analysis

The intra-process analysis for λ_τ analyzes the behavior of a single thread by performing a fixed-point computation of an intra-process transfer function. We first present the state space of the intra-process transfer function before presenting the transfer function itself.

State Space of the Intra-Process Analysis

Figure 6.15 depicts the state space for the intra-process analysis. The domain of the intra-process transfer function consists of intra-states ($\widehat{IntraState}$), which are composed of a set of states reachable, a value store, a continuation store, a set of threads created, a set of threads joined, and a set of addresses accessed (reference addresses and lock addresses). Threads created are represented as a pair of the process identifier of the thread created and its initial thread state. Threads joined are represented by their process identifier. We introduce the notion of *join store*, which stores the return values for threads that have terminated their execution.

λ_τ

$$\begin{aligned}\widehat{IntraState} &= \mathcal{P}(\widehat{\Sigma}) \times \widehat{Store} \times \widehat{KStore} \\ &\quad \times \mathcal{P}(\widehat{Created}) \times \mathcal{P}(\widehat{PID}) \times \mathcal{P}(\widehat{Addr}) \\ \widehat{Created} &= \widehat{PID} \times \widehat{\Sigma}\end{aligned}$$

Figure 6.15.: State space of the intra-process analysis for λ_τ.

Transfer Function of the Intra-Process Analysis

Figure 6.16 depicts the intra-process transfer function $\mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\tau}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\hat{J}} : \widehat{\mathit{IntraState}} \to \widehat{\mathit{IntraState}}$, which is parameterized by the initial state of the thread under analysis ($\hat{\varsigma}$), its process identifier ($\hat{p}$), a value store ($\hat{\sigma}_0$), a continuation store ($\widehat{\Xi}_0$), and a join store ($\hat{J}$). The join store contains information about the return values of threads that have terminated, and serves to enable transitions for the `join` construct.

The transfer function operates as follows with respect to generated effects and effect sets.

1. The initial state of the process, with the corresponding stores, are visited.
2. Effect-less transitions are trivially taken and generate no communication effects.
3. Transitions that create a process add an element to the set of created processes.
4. Transitions that join on another process $\hat{p}'$ add the dependency on process identifier $\hat{p}'$ to the corresponding set.
5. Other transitions that read from a reference, write to a reference, acquire a lock, or release a lock at a specific address register the address accessed as a dependency in the corresponding set of addresses. In the case of locks, the value of $\hat{p}$ inside the communication effect must correspond to the current process identifier.

λ_τ

$$\mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\tau}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\hat{f}}(\langle S,\hat{\sigma},\widehat{\Xi},C,P,A\rangle) = \langle\{\hat{\varsigma}\},\hat{\sigma}_0,\widehat{\Xi}_0,\varnothing,\varnothing,\varnothing\rangle \tag{1}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \rightsquigarrow_{\hat{p},\hat{f}} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\varnothing,\varnothing\rangle \tag{2}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\mathbf{c}(\hat{p}',\hat{\varsigma}_2)}{\rightsquigarrow}_{\hat{p},\hat{f}} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\{(\hat{p}',\hat{\varsigma}_2)\},\varnothing,\varnothing\rangle \tag{3}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\mathbf{j}(\hat{p}',\hat{v})}{\rightsquigarrow}_{\hat{p},\hat{f}} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\{\hat{p}'\},\varnothing\rangle \tag{4}$$

$$\sqcup \bigsqcup_{\substack{\hat{\varsigma}\in S \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \overset{\widehat{\mathit{eff}}}{\rightsquigarrow}_{\hat{p},\hat{f}} \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}' \\ \widehat{\mathit{eff}}\in\{\mathbf{w}(\hat{a}),\mathbf{r}(\hat{a}),\mathbf{acq}(\hat{p},\hat{a}),\mathbf{rel}(\hat{p},\hat{a})\}}} \langle\{\hat{\varsigma}'\},\hat{\sigma}',\widehat{\Xi}',\varnothing,\varnothing,\{\hat{a}\}\rangle \tag{5}$$

Figure 6.16.: Transfer function for the intra-process analysis for λ_τ.

6.4.4. Step 4 of ModConc for λ_τ: Definition of the Inter-Process Analysis

The inter-process analysis is defined as the fixed-point computation of a transfer function $\mathit{Inter}\text{-}\widehat{\mathcal{H}}^{\lambda_\tau}_e$ where e is the program under analysis. The result of the inter-process analysis is a tuple containing in its first element a process map containing information about the reachable states of all threads, the communication effects performed by each thread, and in its second element the value store and continuation store. We first present the state space of the inter-process analysis and then define the inter-process transfer function.

State Space of the Inter-Process Analysis

Figure 6.17 depicts the state space for the inter-process analysis. Process maps of the inter-process analysis map from abstract process identifiers to pairs of intra-states of the last intra-process analysis on the corresponding thread and to the initial state of the corresponding thread.

λ_τ

$$\hat{\pi} \in \widehat{\Pi} = \widehat{\mathit{PID}} \to (\widehat{\mathit{IntraState}} \times \widehat{\Sigma})$$

Figure 6.17.: State space of the inter-process analysis for λ_τ.

Transfer Function of the Inter-Process Analysis

The inter-process transfer function relies on the following functions defined in Figure 6.18.

- $\mathit{explore} : \widehat{\Pi} \times \widehat{\mathit{PID}} \times \widehat{\Sigma} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}} \times \widehat{\mathit{JoinStore}} \to \widehat{\Pi} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}} \times \widehat{\mathit{JoinStore}}$ requires the current process map $\hat{\pi}$ and performs an intra-process analysis on the actor with the process identifier $\hat{p}$, initial state $\hat{\varsigma}$, using the value store $\hat{\sigma}$, the continuation store $\widehat{\Xi}$ and the join store $\hat{J}$. It returns a process map, a value store, a continuation store and a join store containing the information resulting from the intra-process analysis.
- $\mathit{created} : \widehat{\Pi} \to \mathcal{P}(\widehat{\mathit{PID}} \times \widehat{\Sigma})$ returns the set of all created threads inferred by the intra-process analyses, described by their process identifier and initial thread state.
- $\mathit{joins} : \widehat{\Pi} \times \widehat{\mathit{JoinStore}} \to \mathcal{P}(\widehat{\mathit{PID}} \times \widehat{\Sigma})$ returns the set of threads (described as a pair of process identifier and initial thread state) that join a thread for which the return value has been inferred and stored in the join store $\hat{J}$.
- $\mathit{conflicts} : \widehat{\Pi} \to \mathcal{P}(\widehat{\mathit{PID}} \times \widehat{\Sigma})$ returns the set of threads (described as a pair of process identifier and initial thread state) that may be in conflict with other threads because they access the same reference or lock. Through abstract counting (Might and Shivers, 2006), this function may be refined by ensuring that $p \neq p'$ when it is known that both process identifiers map to a single thread.

λ_τ

$$\mathit{explore}(\hat{\pi}, \hat{p}, \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi}, \hat{J}) = \langle [\hat{p} \mapsto (s, \hat{\varsigma})], \hat{\sigma}', \widehat{\Xi}', \hat{J}' \rangle$$
$$\text{where } s = \mathrm{lfp}(\mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\tau}_{\hat{p},\hat{\varsigma},\hat{\sigma},\widehat{\Xi},\hat{J}})$$
$$\text{and } (\_, \hat{\sigma}', \widehat{\Xi}', \_, \_, \_) = s$$
$$\text{and } \hat{J}' = \bigsqcup_{\substack{\hat{p} \in \mathrm{dom}(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((S, \_, \_, \_, \_, \_, \_), \_) \\ \langle \mathbf{val}(v), \widehat{k}_0 \rangle \in S}} [\hat{p} \mapsto \{v\}]$$
$$\mathit{conflicts}(\hat{\pi}) = \bigcup_{\substack{\hat{p}, \hat{p}' \in \mathrm{dom}(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((\_, \_, \_, \_, \_, A), \hat{\varsigma}) \\ \hat{\pi}(\hat{p}') = ((\_, \_, \_, \_, \_, A'), \hat{\varsigma}') \\ A \cap A' \neq \varnothing}} \{(\hat{p}, \hat{\varsigma}), (\hat{p}', \hat{\varsigma}')\}$$
$$\mathit{joins}(\hat{\pi}, \hat{J}) = \bigcup_{\substack{\hat{p} \in \mathrm{dom}(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((\_, \_, \_, \_, P, \_), \hat{\varsigma}) \\ \exists \hat{p}' \in P, \hat{J}(\hat{p}) \neq \varnothing}} (\hat{p}, \hat{\varsigma})$$
$$\mathit{created}(\hat{\pi}) = \bigcup_{\substack{\hat{p} \in \mathrm{dom}(\hat{\pi}) \\ \hat{\pi}(\hat{p}) = ((\_, \_, \_, C, \_, \_, \_), \_)}} C$$

Figure 6.18.: Auxiliary functions used by the inter-process transfer function for λ_τ.

Figure 6.19 depicts the inter-process transfer function $\mathit{Inter}\text{-}\widehat{\mathcal{H}}^{\lambda_\tau}_e : \widehat{\Pi} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}} \times \widehat{\mathit{JoinStore}} \to \widehat{\Pi} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}} \times \widehat{\mathit{JoinStore}}$. This transfer function performs the following operations.

1. The main thread is analyzed by the intra-process analysis.
2. Each newly created thread discovered by a previous intra-process analysis (as extracted by the *created* function) is analyzed.
3. Each thread that joins another thread for which a previous intra-process analysis has inferred a return value (as extracted by the *joins* function) is reconsidered for analysis to account for the possibly new inferred return values.
4. Each thread that accesses a memory address that is part of the set of conflicting addresses between different threads (as extracted by the *conflicts* function) is reconsidered for analysis to account for a possible change in value at the conflicting address.

The inter-process analysis of a λ_τ program e is then the computation of the fixed point of this transfer function, $\mathrm{lfp}(\mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\tau})$, and results in an over-approximation of the set of all reachable thread states contained in a process map, as well as an over-approximation of the value store and continuation store.

λ_τ

$$\mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\tau}(\langle \hat{\pi}, \hat{\sigma}, \widehat{\Xi}, \hat{J} \rangle) = \mathit{explore}([], \hat{p}_0, \langle \mathbf{ev}(e, []), \widehat{k}_0 \rangle, \hat{\sigma}, \widehat{\Xi}, \hat{J}) \tag{1}$$

$$\sqcup \bigsqcup_{(\hat{p},\hat{\varsigma}) \in \mathit{created}(\hat{\pi})} \mathit{explore}(\hat{\pi}, \hat{p}, \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi}, \hat{J}) \tag{2}$$

$$\sqcup \bigsqcup_{(\hat{p},\hat{\varsigma}) \in \mathit{joins}(\hat{\pi}, J)} \mathit{explore}(\hat{\pi}, \hat{p}, \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi}, \hat{J}) \tag{3}$$

$$\sqcup \bigsqcup_{(\hat{p},\hat{\varsigma}) \in \mathit{conflicts}(\hat{\pi})} \mathit{explore}(\hat{\pi}, \hat{p}, \hat{\varsigma}, \hat{\sigma}, \widehat{\Xi}, \hat{J}) \tag{4}$$

Figure 6.19.: Inter-process transfer function for λ_τ.

Complexity

The inter-process analysis may reconsider a thread for analysis for every thread created, for every joins performed, and for potential conflicts. Its worst-case time complexity is therefore of $\mathcal{O}((t + j + c) \times |Exp|^3)$, where t is the number of abstract threads created, j is the number of joins performed, c is the number of potential conflicts arising in the program, and $|Exp|^3$ comes from the complexity of the intra-process analysis.

6.4.5. Soundness and Termination

Theorems 18 and 19 state that the analysis described by the inter-process analysis terminates and is sound, two crucial properties for a static analysis.

Theorem 18 (Soundness). *The result of* $\mathrm{lfp}(\mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\tau})$ *is a sound over-approximation of* $\mathrm{lfp}(\mathcal{F}_e^{\lambda_\tau})$.

Proof. The proof is detailed in Appendix B.4.2. This idea of this proof is the following. We show by a case analysis on the transition rules that the intra-process analysis is a sound over-approximation of the transfer function $\widehat{\mathcal{F}}^{\lambda_\tau}$ restricted to states reachable with transition on the same process $\hat{p}$ from the initial state $\langle[\hat{p} \mapsto \{\hat{\varsigma}_0\}], \hat{\sigma}_0, \widehat{\Xi}_0\rangle \sqcup \bigsqcup_{\hat{p}\in\mathrm{dom}(\hat{\jmath}),\hat{v}\in\hat{\jmath}(\hat{p})} \langle[\hat{p} \mapsto \left\{\mathbf{val}(\hat{v}), \widehat{k}_0\right\}], [], []\rangle$. As the inter-process analysis considers every newly created thread and every thread affected by joins and conflicts, the analysis will eventually have analyzed all running threads with over-approximation of their stores.

□

Theorem 19 (Termination). *The computation of* $\mathrm{lfp}(\mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\tau})$ *always terminates.*

Proof. The proof is detailed in Appendix B.4.2 and follows the same reasoning as the proof for Theorem 17: the transfer functions are monotone and act on a finite state space, hence the analysis always terminate. □

6.5. Soundness Testing and Evaluation of Running Time, Precision, and Scalability on a Benchmark Suite

As was done for the analyses developed in Chapters 2 and 4, we have implemented the analyses resulting from the application of ModConc to λ_α and to λ_τ using our Scala-AM static analysis framework presented in Section 2.4.1. We empirically evaluate these analyses in terms of running time, precision, and scalability on the set of benchmark programs introduced in Section 2.4.2.

6.5.1. Soundness Testing

We have proven that the application of ModConc to λ_α and λ_τ results in a sound analysis (Theorems 16 and 18). We also provide empirical evidence for the soundness of our implementation through soundness testing (Andreasen *et al.*, 2017), as described in Section 2.4.3. In brief, we compare the information computed by each analysis to the information recorded during 1000 concrete runs of each benchmark, and check that all concrete information is soundly over-approximated by the analysis. No unsound results were discovered.

6.5.2. Running Time

Similar to the evaluation of the analyses developed in the previous chapters, we ran the analyses presented in this chapter on each of our benchmark programs and report on the average time required to analyze each benchmark with the same setup as used in the evaluation of Section 2.4: each program is analyzed 20 times after 10 warmup runs, with

a time budget of 30 minutes. The results are given in Table 6.1. All benchmark programs can be analyzed within the given time budget, with analysis times varying between 8 milliseconds and 42 seconds. As we study in Section 6.5.4, the factor that increases analysis time is the number of communication effects performed by the program under analysis.

Actors				Threads			
Bench.	Time	Bench.	Time	Bench.	Time	Bench.	Time
PP	377	BTX	68	ABP	34	TRAPR	41
COUNT	43	RSORT	39	COUNT	38	ATOMS	74
FJT	73	FBANK	75	DEKKER	14	STM	42052
FJC	8	SIEVE	29	FACT	337	NBODY	912
THR	43	UCT	837	MATMUL	33612	SIEVE	113
CHAM	80	OFL	6365	MCARLO	42	CRYPT	5303
BIG	76	TRAPR	69	MSORT	889	MCEVAL	12168
CDICT	98	PIPREC	42	PC	23	QSORT	227
CSLL	81	RMM	354	PHIL	19	TSP	717
PCBB	754	QSORT	1456	PHILD	24	BCHAIN	182
PHIL	54	APSP	1142	PP	14	LIFE	1742
SBAR	56	SOR	2213	RINGBUF	73	PPS	394
CIG	46	ASTAR	205	RNG	33	MINIMAX	6814
LOGM	106	NQN	498	SUDOKU	144	ACTORS	1950

Table 6.1.: Running times of the analyses presented in this chapter on our benchmark programs. The timings are expressed in milliseconds and represent the average of 20 runs after 10 warmup runs. ∞ denotes that the analysis exceeded the allocated time budget of 30 minutes.

Comparison to Naive Applications of AAM

As for analyses resulting from the application of MacroConc, we observe a substantial improvement in running time of the analyses presented here over the analyses presented in Chapter 2. Table 6.2 compares running times for benchmarks that can be analyzed within the time budget by both analyses. The application of ModConc rather than a naive application of AAM results in analyses that are up to four orders of magnitude faster, with a speedup factor ranging from 7 to 66030, which are the double of the minimal and maximal speedup factors resulting from the application of MacroConc.

Comparison to MacroConc

Whereas the analyses of Chapter 4 analyze 30 out of the 56 benchmarks within the time limit of 30 minutes, the analyses resulting from the application of ModConc successfully analyze all of them in a few seconds at most per program. Table 6.3 compares the running times for benchmarks that can be analyzed within the time budget by both analyses. Whereas the analyses of Chapter 4 have a speedup factor of up to four orders of magnitude compared to the analyses of Chapter 2, the process-modular analysis results in a speedup factor (i.e., MacroConc time over ModConc time) ranging from 1.31 and

Actors				Threads			
Bench.	Naive	Mod	Speedup	Bench.	Naive	Mod	Speedup
PP	2876	377	÷7.63	ABP	92163	34	÷2710.68
COUNT	920	43	÷21.40	DEKKER	20675	14	÷1476.79
FJT	435812	73	÷5970.03				
FJC	528242	8	÷66030.25				

Table 6.2.: Running time comparison between the analyses resulting from the naive application of AAM (column *Naive*) and the analyses resulting from the application of ModConc presented in this chapter (column *Mod*), for programs analyzed by both analyses in under 30 minutes. Columns *Naive* and *Mod* contain timings in milliseconds. Columns *Speedup* represent the speedup factor of the running time of analyses resulting from the application of AAM over the running time of analyses resulting from the application of ModConc.

27016, that is up to four more orders of magnitude. Note that this does not mean that the application of ModConc does not result in a speedup factor of eight orders of magnitude, as the benchmark profiting from the highest speedup factor in the application of MacroConc and of ModConc are different. This however demonstrates that ModConc results in analyses with a substantially better scalability than MacroConc, which itself results in analyses with a substantially better scalability than naive applications of the AAM design method. We observe a better speedup for programs that take more time for MacroConc to analyze, as the time needed to analyze such programs with ModConc is consistent across the benchmarks.

Comparison to Related Work

We reiterate the comparison to the Soter tool (D'Osualdo *et al.*, 2012). It behaves similarly to the actor analysis presented in Chapter 4 in terms of successfully analyzed benchmarks. Table 6.4 compares the analyses presented in this chapter to Soter. In terms of successfully analyzed benchmarks, while Soter analyzes 11 out of the 28 Savina benchmark programs under the 2 minutes timeout of its web interface, our process-modular actor analysis can analyze all of them in a few seconds at most. While it would not be fair to compare raw timings, as Soter is run through a web interface and we cannot remove the initialization overhead of the analysis, we can see that the timings are in some cases drastically different: our analysis takes generally under 3 seconds to analyze each benchmark, with one exception at 6 seconds, while Soter may take up to 30 seconds to analyze some programs, and fails to analyze more than half of the benchmark programs. This confirms that analyses resulting from applications of ModConc scale better than analyses relying on all-interleavings semantics such as the ones presented in Chapter 2 and used by Soter, and better than analyses relying on macro-stepping semantics such as the ones presented in Chapter 4.

Actors				Threads			
Bench.	Macro	Mod	Speedup	Bench.	Macro	Mod	Speedup
PP	494	377	÷1.31	ABP	1014	34	÷29.82
COUNT	284	43	÷6.56	COUNT	48003	38	÷1263.24
FJT	112	73	÷1.53	DEKKER	275	14	÷19.64
FJC	15	8	÷1.88	MCARLO	1134673	42	÷27016.02
THR	233744	43	÷5435.91	PC	12852	23	÷558.78
CHAM	1499941	80	÷18749.26	PHIL	463	19	÷24.37
CDICT	21246	98	÷216.80	PHILD	5168	24	÷215.33
PCBB	99630	754	÷132.14	PP	657	14	÷46.93
CIG	1337	46	÷29.97	RNG	711	33	÷21.55
BTX	2567	68	÷37.75	TRAPR	559	41	÷13.63
RSORT	12913	39	÷331.10	ATOMS	8710	74	÷117.70
FBANK	619849	75	÷8264.65	SIEVE	1026	113	÷9.08
SIEVE	79	29	÷2.72	BCHAIN	74411	182	÷408.85
TRAPR	10301	69	÷149.29	LIFE	663712	1742	÷381.01
PIPREC	387	42	÷9.21	PPS	56669	394	÷143.83

Table 6.3.: Running time comparison between the analyses resulting from the application of MacroConc (column *Macro*) and the analyses resulting from the application of ModConc presented in this chapter (column *Mod*), for programs analyzed by both analyses in under 30 minutes. Columns *Macro* and *Mod* contain timings in milliseconds. Columns *Speedup* represent the speedup factor of the running time of analyses resulting from the application of MacroConc over the running time of analyses resulting from the application of ModConc.

Bench.	Mod	Soter	Bench.	Mod	Soter	Bench.	Mod	Soter	Bench.	Mod	Soter
PP	377	150	CDICT	98	∞	BTX	68	∞	PIPREC	42	∞
COUNT	43	310	CSLL	81	∞	RSORT	39	∞	RMM	354	∞
FJT	73	470	PCBB	754	∞	FBANK	75	∞	QSORT	1456	∞
FJC	8	110	PHIL	54	29840	SIEVE	29	4240	APSP	1142	∞
THR	43	1130	SBAR	56	1710	UCT	837	∞	SOR	2213	∞
CHAM	80	1860	CIG	46	∞	OFL	6365	∞	ASTAR	205	∞
BIG	76	∞	LOGM	106	31950	TRAPR	69	∞	NQN	498	29090

Table 6.4.: Comparison between the analysis for concurrent actors presented in this chapter (column *Mod*) and the analysis for concurrent actors present in Soter (D'Osualdo *et al.*, 2012) (column *Soter*). Each benchmark program is analyzed with a time limit of 2 minutes. Running times are given in milliseconds.

6.5.3. Precision

We evaluate the precision of the analyses resulting from the application of ModConc in the same way as for the analyses developed in Chapter 4. We run each benchmark concretely and record observed concrete information about the `become` statements executed, processes created, messages received, values returned by threads, references accessed and modified, and locks acquired and released. We obtain an under-approximation of the maximally-precise analysis results by abstracting all the observed concrete values. This allows us to detect potentially spurious elements, i.e. elements that have been computed by the analysis but that may not arise in any concrete execution of the program under analysis. We report on the number of elements observed and the number of spurious elements detected in Table 6.5. We remind the reader that these numbers are mere upper bounds on the number of spurious elements, and therefore a lower bound on the precision, because concrete values that can arise under specific interleavings may not be observed during any of the 1000 concrete runs.

Actors						Threads					
Bench.	Obs.	Spu.	Bench.	Obs.	Spu.	Bench.	Obs.	Spu.	Bench.	Obs.	Spu.
PP	9	0	BTX	9	0	ABP	20	0	TRAPR	2	0
COUNT	8	0	RSORT	10	0	COUNT	6	4	ATOMS	6	0
FJT	3	0	FBANK	38	0	DEKKER	16	5	STM	11	7
FJC	2	0	SIEVE	8	0	FACT	21	8	NBODY	25	0
THR	5	2	UCT	20	8	MATMUL	40	0	SIEVE	4	2
CHAM	10	0	OFL	13	0	MCARLO	12	0	CRYPT	2	2
BIG	10	0	TRAPR	7	0	MSORT	12	0	MCEVAL	2	2
CDICT	13	0	PIPREC	8	0	PC	15	0	QSORT	18	0
CSLL	16	0	RMM	14	0	PHIL	4	0	TSP	12	8
PCBB	13	0	QSORT	6	0	PHILD	8	0	BCHAIN	7	0
PHIL	13	0	APSP	5	0	PP	6	0	LIFE	16	4
SBAR	19	0	SOR	12	1	RINGBUF	19	2	PPS	10	0
CIG	9	0	ASTAR	11	12	RNG	6	0	MINIMAX	4	0
LOGM	15	0	NQN	11	0	SUDOKU	58	0	ACTORS	18	0

Table 6.5.: Precision evaluation for ModConc analyses. Column *Obs.* lists the number of observed elements among 1000 concrete runs of each benchmark program. Column *Spu.* lists the number of potential spurious elements inferred by an analysis, i.e., abstract elements that have no corresponding element observed in any concrete run.

The analyses presented in this chapter obtain full precision on a majority of benchmarks: 42 of the 56 benchmarks are analyzed with maximal precision. A total of 67 spurious elements are detected, while the 697 other elements observed in concrete runs have been correctly inferred by the analysis. This results in a precision of 91%[1]. These numbers cannot be directly compared to the numbers resulting from our evaluation of macro-stepping analysis, as there are more benchmarks analyzed within the time limit by the analyses resulting from the application of ModConc than from the application

[1] There are 697 correctly observed elements (true positives) and 67 spurious elements (false positives), hence the precision is $\frac{697}{697+67} = 0.91$.

of MacroConc.

Comparison to MacroConc

To compare the analyses resulting from the application of MacroConc with the analyses resulting from the application of ModConc in terms of precision, we limit the comparison to benchmark programs analyzed by both approaches within the time budget. Table 6.6 reports on the number of detected spurious elements for each analysis.

Actors						Threads					
Bench.	Macro	Mod	Bench.	Macro	Mod	Bench.	Macro	Mod	Bench.	Macro	Mod
PP	0	0	CIG	0	0	ABP	0	0	RNG	0	0
COUNT	0	0	BTX	0	0	COUNT	4	4	TRAPR	0	0
FJT	0	0	RSORT	0	0	DEKKER	5	5	ATOMS	0	0
FJC	0	0	FBANK	0	0	MCARLO	0	0	SIEVE	2	2
THR	2	2	SIEVE	0	0	PC	0	0	BCHAIN	0	0
CHAM	0	0	TRAPR	0	0	PHIL	0	0	LIFE	0	4
CDICT	0	0	PIPREC	0	0	PHILD	0	0	PPS	0	0
PCBB	0	0				PP	0	0			

Table 6.6.: Comparison between the precision of analyses resulting from the application of MacroConc and from the application of ModConc. Column *Macro* and *Mod* lists the number of spurious elements for, respectively, the analyses resulting from the application of MacroConc and the analyses resulting from the application of ModConc.

These numbers are almost identical with the exception of the LIFE benchmark, where the actor analysis of this chapter infers 4 spurious elements while none are inferred by the actor analysis of Chapter 4. This is because the process-modular analysis does not contain information about the ordering between the execution of multiple processes, and re-analyzes a specific thread in the LIFE benchmark with an over-approximation of the store that contains values that are written to the store *after* the thread has finished its execution, and are therefore too over-approximative. This results in spurious writes and reads being detected. Therefore, the precision goes from 96% (Chapter 4) to 94%[2] for the process-modular analysis presented here. The cost in terms of precision is acceptable in order to obtain an analysis that can analyze all of the programs in the benchmark suite in a few seconds.

6.5.4. Scalability

We have shown that the worst-case time complexity of analyses resulting from the application of ModConc is polynomial, scaling linearly with the number of communication effects (Section 6.2.3), while it is exponential for the analyses presented in Chapters 2

[2]There are 290 correctly observed elements (true positives) and 17 spurious elements (false positives), hence the precision is $\frac{290}{290+17} = 0.94$.

and 4. We evaluate the scalability of the analyses resulting from the application of ModConc empirically using the same families of synthetic benchmark programs as the ones used in Chapter 4. In brief, these benchmarks evaluate the scalability of an analysis for λ_α with respect to the number of different static behaviors (b), and the number of different messages exchanged (m), and they evaluate the scalability of an analysis for λ_τ with respect to the number of threads created (t), the number of conflicts between a fixed number of threads (c), and the number of joins on different threads (j).

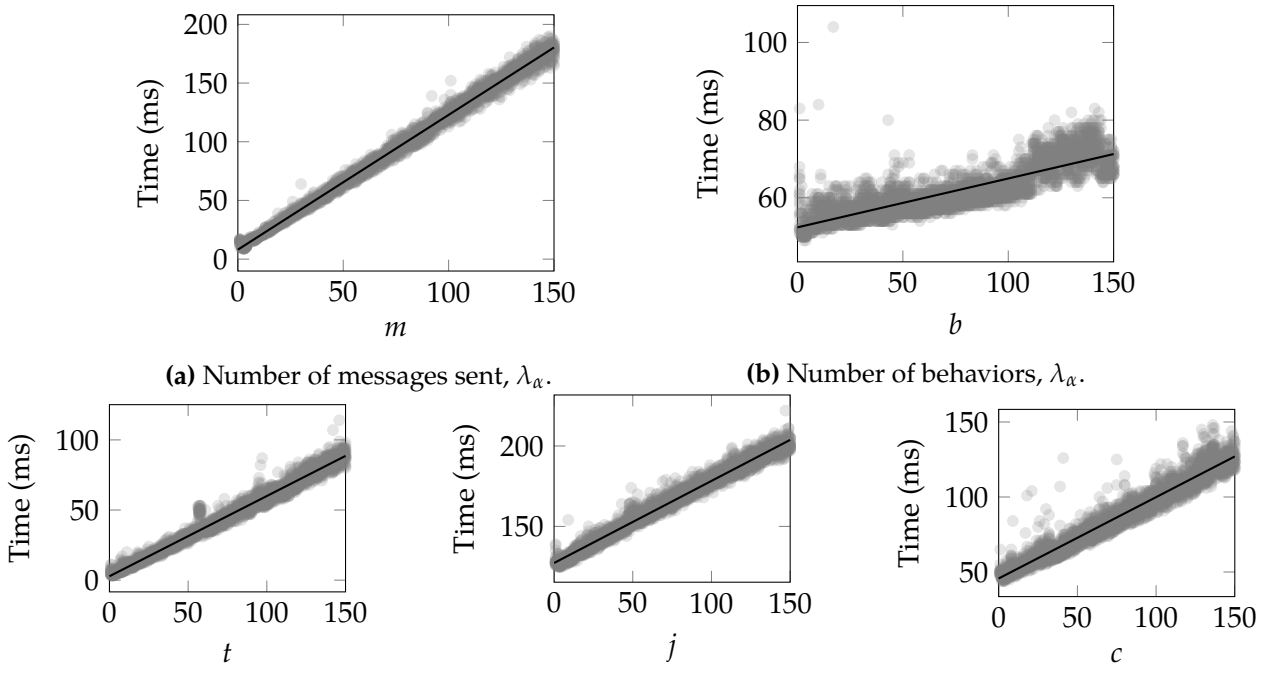
The analyses presented in this chapter scale significantly better than the analyses of Chapter 4, and we are able to run each of the benchmark program families with higher values for their parameters. Once the values for these parameters exceeded 10, the analyses from Chapter 4 started to exceed the time budget of 30 minutes. For the analyses presented in this chapter, we are able to increase the value of their parameters to 150[3]. Each of the parameters studied in this scalability evaluation corresponds to one kind of communication effect, therefore we expect to see a linear trend between analysis time and the increasing value of the parameter of each benchmark. The worst-case time complexity of an analysis resulting from the application of ModConc is $\mathcal{O}(|\widehat{Effect}| \times |Exp|^3)$, and by keeping $|Exp|$ constant, we measure the impact of an increased number of communication effects. More specifically, the complexity of the analysis for λ_α is $\mathcal{O}((m+b) \times |Exp|^3)$, where m is the number of abstract messages sent, and b is the number of abstract behaviors. The complexity of the analysis for λ_τ is $\mathcal{O}((t+j+c) \times |Exp|^3)$, where t is the number of abstract threads created, j is the number of joins performed, and c is the number of potential conflicts. We recorded the analysis time for each benchmark program for each value of the parameters on 20 runs after 10 warm-up runs. The results are reported in Figure 6.20, where each data point corresponds to the time taken to run an analysis, and where we plot the best-fitting linear regression.

On these graphs, we can see that our implementations of the ModConc analyses scale linearly with an increasing number of communication effects. This confirms the theoretical results from Section 6.2.3. Moreover, looking at the y axis further demonstrates the improved running times, in milliseconds here, compared to the analyses of Chapter 4, where the y axis goes up to 30 minutes.

6.6. Conclusion

In this chapter, we presented the ModConc analysis design method which, when applied to the operational semantics of a concurrent programming language, results in a static analysis that scales linearly with the number of communication effects performed in the program—thanks to its process-modular design. By sequentializing the operational semantics of the input language, an intra-process analysis reasons only about a single

[3]We could have increased the maximal value of the parameter, however as the maximal value of the parameter increases, the number of expressions in the benchmark program increases as well. Due to a naive abstract syntax normalization strategy in our implementation, higher values for the parameters result in stack overflows, hence we limited the maximal value to 150.



(a) Number of messages sent, λ_α.

(b) Number of behaviors, λ_α.

(c) Number of threads, λ_τ.

(d) Number of joins, λ_τ.

(e) Number of conflicts, λ_τ.

Figure 6.20.: Scalability evaluation of ModConc. Each graph corresponds to a benchmark which is parameterized by a value (m for number of messages, and b for number of actor behaviors, t for abstract threads, j for join operations, c for conflicts) that is increased from 1 to 150, and shows the running time of the analysis in function of the value of the given parameter.

process and infers the communication effects performed by this process. This intra-process analysis is driven by an inter-process analysis, which collects the communication effects inferred by the intra-process analyses in order to consider new processes and processes affected by communication effects for new runs of the intra-process analysis. We presented this analysis design method in a generic formulation and applied it to the λ_α concurrent actor language and to the λ_τ shared-memory multi-threaded language.

We have demonstrated the theoretical properties of the resulting analyses, which are sound and terminate, adding only a polynomial cost to the complexity of the underlying intra-process analysis, and therefore remain polynomial in their worst-case time complexity. Furthermore, we evaluated the analyses empirically through their implementation. The entirety of the benchmark suite described in Section 2.4.2 can be analyzed within the time limit, and the analyses take less than a minute to analyze each benchmark. The resulting analyses also perform better in terms of running time and scalability than the analyses resulting from a naive application of AAM (Chapter 2), than the analyses resulting from the application of MacroConc (Chapter 4), and than the closest related work (D'Osualdo *et al.*, 2012). This major improvement in scalability comes at a minor cost in precision, as the analyses detect 4 more spurious elements on

all of our benchmark suite. A more substantial cost in precision is that the resulting analyses are unable to reason about properties that require ordering or multiplicity information, such as bounds on mailboxes. We identify the investigation of solutions to this limitation as future work in Chapter 7. However, the precision of the analyses remains very high to support communication topology analyses (Colby, 1995; Martel and Gengler, 2000).

7

CONCLUSION AND FUTURE WORK

7.1. Summary of the Dissertation

Concurrent programs are becoming increasingly prevalent both in multi-core programming and in cloud-commodity infrastructure. However, they tend to be difficult to reason about and they may contain subtle bugs that only manifest in specific interleavings of the instructions of their processes. Tool support for concurrent programs is therefore necessary, and could be enabled by static program analyses. In this dissertation, we provide a solid foundation for such tool support by introducing two analysis design methods that are applicable to the operational semantics of concurrent programs.

We started our discourse with a naive application of the AAM analysis design method (Van Horn and Might, 2010) to concurrent programs, specifically to concurrent actor programs implemented in λ_α (Section 2.2) and to multi-threaded programs featuring shared memory implemented in λ_τ (Section 2.3). Both languages extend the base language λ_0, which features higher-order functions. The application of the AAM design method to the operational semantics of λ_α and λ_τ results in static analyses that feature many desirable properties: automation, soundness, precision and support for dynamic process creation. To evaluate the analyses developed in this dissertation, we selected 56 benchmark programs, 28 of which are concurrent actor programs adapted from the Savina benchmark suite (Imam and Sarkar, 2014), and 28 of which are shared-memory multi-threaded programs of our design (Section 2.4.2). These benchmark programs contain challenging features for static analyses to support such as dynamic process creation and termination, and the use of higher-order functions. As we have demonstrated in our evaluation (Section 2.4), the analyses resulting from a naive application of AAM to concurrent programs do not scale well, as only 6 of the 56 benchmark programs can be analyzed within a time limit of 30 minutes.

We then reviewed existing analyses for concurrent programs, and observed that no existing analysis exhibits all of the desirable properties that are automation, soundness, scalability, precision and support for dynamic process creation (Section 3.1). As the analyses introduced in Chapter 2 fail to scale, we identified from the related work two

analysis designs that improve on existing analyses in terms of scalability: state space reduction and process-modular analysis (Section 3.2). We incorporated these insights in the AAM design method, giving rise to the MacroConc and the ModConc analysis design methods.

Our first analysis design method, MacroConc (Chapter 4), relies on macro-stepping semantics as a way to mitigate the state explosion problem and to improve the scalability of the resulting analyses. Refining the concrete all-interleavings semantics of a concurrent programming language with macro-stepping semantics enables reducing the non-determinism inherent to its semantics. This is performed by advancing the execution of a process in *macro steps*, which accumulate successive steps of the transition relation on the same process until a possible conflict arises and macro steps on other processes need to be considered. We presented MacroConc in a generic manner (Section 4.1) and described its theoretical properties (Section 4.2): the design method gives rise to sound analyses that are as precise as the analyses resulting from a naive application of AAM presented in Chapter 2, yet might explore fewer process interleavings.

We presented analyses resulting from MacroConc, for λ_α programs (Section 4.3) and for λ_τ programs (Section 4.4), and evaluated these analyses in terms of running time, precision and scalability (Section 4.5). We observed that analyses resulting MacroConc feature reduction in running time of up to four orders of magnitude compared to the analyses resulting from a naive application of AAM. The precision of the analyses resulting from the application of MacroConc and from the naive application of AAM is the same, and is high: we demonstrated a 96% precision on our benchmark suite for a communication topology analysis. However, we established formally that the worst-case time complexity of the analyses resulting from the application of MacroConc is of $\mathcal{O}(2^{|Exp|})$, i.e., exponential in the size of the program (Section 4.2), and we observe empirically (Section 4.5.4) that their scalability remains limited, as they exhibit exponential behavior in the worst-case.

As part of our presentation of analysis design methods for concurrent programs, we studied different mailbox abstractions that can be used in a static analysis of actor programs in order to improve the precision of the analysis (Chapter 5). We identified two important properties of mailbox abstractions for the precision of static analyses: whether an abstraction preserves ordering and whether an abstraction preserves multiplicity (Section 5.1). We presented and categorized five mailbox abstractions that vary according to these properties (Section 5.2): a set abstraction used in Chapters 2, 4 and 6, a multiset abstraction, a finite multiset abstraction, a finite list abstraction, and a new graph abstraction. We proved the soundness of each of these abstractions, and evaluated their impact on the precision and the scalability of an actor analysis resulting from the application of our MacroConc design method (Section 5.3). The abstractions discussed are significantly more precise than the coarse set abstraction used in the other chapters of this dissertation, yet with almost no cost in terms of performance.

We introduced ModConc (Chapter 6) next, our second analysis design method that results in analyses that feature a process-modular design. We presented ModConc in a generic manner (Section 6.1) and discussed the theoretical properties of analyses

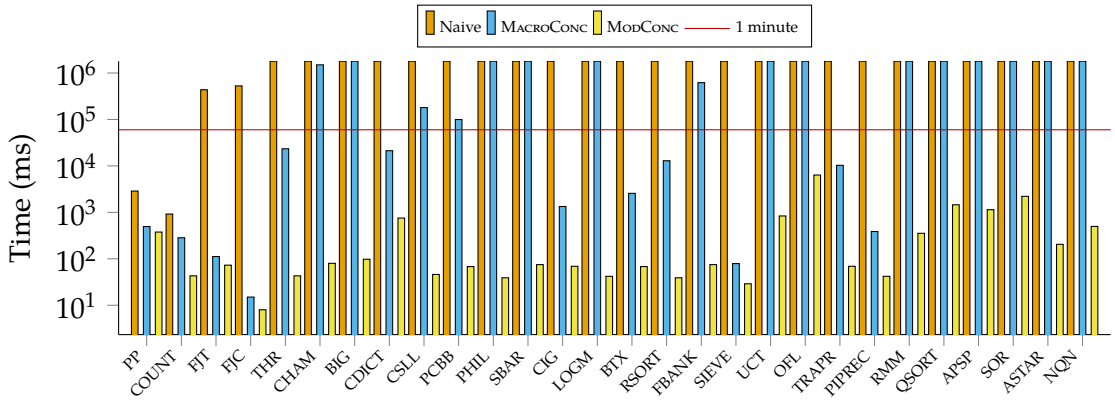
resulting from the application of ModConc (Section 6.2). Such analyses are sound, terminate, and feature a complexity that is only impacted by a linear factor of the number of communication effects present in the analyzed programs. Formally, their worst-case time complexity is of $\mathcal{O}(|\widehat{Effect}| \times |Exp|^3)$ where $\widehat{Effect}$ is the abstraction of the set of all effects that can arise in the execution of the program under analysis. Their precision is however lower, as analyses resulting from the application of ModConc are not able to reason about the ordering nor about the multiplicity of communication effects.

We applied ModConc to the operational semantics of the λ_α concurrent actor language (Section 6.3) and to the operational semantics of the λ_τ shared-memory multi-threaded programming language (Section 6.4). We evaluated the resulting analyses in terms of running time, precision and scalability (Section 6.5). The running times of the resulting analyses on our benchmark suite improve by up to four orders of magnitude compared to the running times of analyses resulting from the application of MacroConc, as a result of the process-modular design of the analyses. The cost in terms of precision is minimal, as only one program of our benchmark suite is impacted, resulting in four detected false positives. The precision of the analyses remains high: 91% on the full benchmark suite, and 95% on the programs that can also be analyzed by analyses resulting from the application of MacroConc. Whether or not this loss can be regained by combining process-modular analyses with more precise domains such as our mailbox abstractions remains an open question. Finally, we demonstrate the improved scalability through a set of synthetic benchmark programs, where the analysis time increases linearly with the number of communication effects in the program.

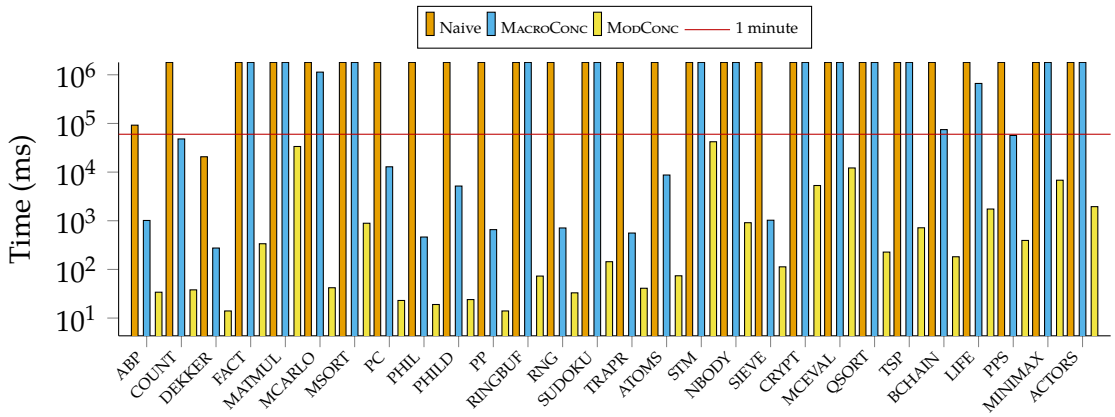
Figure 7.1 summarizes the improvements in terms of running time of our two design methods. Figure 7.1a shows the improvement for the analysis of λ_α programs, and Figure 7.1b shows the improvement for the analysis of λ_τ programs. Note the logarithmic scale of these graphs, and note how MacroConc consistently results in improved running times on programs that can be analyzed within the time limit, and how ModConc consistently results in improved running times compared to MacroConc.

7.2. Contributions

Our first contribution is MacroConc (Chapter 4), an analysis design method that improves the efficiency of AAM-style analyses for concurrent programs without compromising their precision. Incorporating macro-stepping (Agha *et al.*, 1997) into the operational semantics of a concurrent programming language before applying the AAM design method (Van Horn and Might, 2010) results in static program analyses that explore fewer process interleavings. We call this design method MacroConc and used it to design an analysis of concurrent actor programs and an analysis of shared-memory multi-threaded programs. The resulting analyses are formally proven to be sound, to terminate and to preserve their precision with respect to an all-interleavings analysis. We also demonstrate empirically that such analyses can achieve high precision and can yield running times up to four orders of magnitude better than a naive applications of



(a) λ_α programs.



(b) λ_τ programs.

Figure 7.1.: Evolution of running times for analyses resulting from the application of MacroConc and of ModConc. The times are capped at 30 minutes, the time limit used in our evaluation. The y axis is in milliseconds and is logarithmic.

AAM.

Our second contribution is a categorization of the mailbox abstractions that can be used by static analyses for concurrent actor programs, which includes a new graph-based abstraction for mailboxes (Chapter 5). We described five mailbox abstractions, which we categorized according to whether they preserve ordering of messages and according to whether they preserve multiplicity of messages. We proved the soundness of each mailbox abstraction formally, and evaluated their impact empirically on an actor analysis resulting from the application of MacroConc. We demonstrated that a finite list abstraction and a graph abstraction yield the highest precision in analyses for local-process properties, through an analysis that verifies the absence of errors and one that infers bounds for mailboxes.

Our third contribution is ModConc (Chapter 6), an analysis design method that improves the efficiency of AAM-style analyses for concurrent programs, by eliminating the state explosion problem at the cost of precision. The resulting analyses feature a process-modular design and are scalable in nature. Each process of a concurrent program is analyzed in isolation to infer potential interferences with other and with newly created processes, which will have to be reconsidered for analysis until a fixed point is reached. We applied this design method to concurrent actor programs and shared-memory multi-threaded programs, proved the soundness and termination of the resulting analyses, and evaluated their running time, precision and scalability empirically. We observed that the running times on programs from our benchmark suite are improved by up to four more orders of magnitude compared to analyses resulting from the application of MacroConc. The precision is reduced but remains very high as we demonstrate. Finally, the analyses resulting from the application of ModConc are shown to scale linearly with the number of communication effects performed in the program under analysis.

Our fourth and final contribution is Scala-AM, a static analysis framework with which we implemented all the analyses described in this dissertation. This modular and extensible framework is used throughout the dissertation to provide empirical evidence of the soundness, scalability and precision of the analyses developed in this dissertation. This contribution enables future experiments to be developed on the analyses presented in this chapter with minimal implementation effort. This is demonstrated in Chapter 5, where we plug new mailbox abstractions into the analysis developed for actors in Chapter 4 to evaluate their impact on static analysis for actors. The flexibility of this framework is further demonstrated by its use outside of this dissertation in the domain of static analysis by abstract interpretation (Vandercammen *et al.*, 2015; Vandercammen and De Roover, 2016; De Bleser *et al.*, 2017; Vandercammen and De Roover, 2017; Van Es *et al.*, 2017b).

7.3. Limitations and Future Work

We state the limitations of our approach, discuss how these limitations can be addressed in future work, and identify possible avenues of future research on our design methods.

7.3.1. Applicability to Real-World Concurrent Programs

We have shown that analyses resulting from the application of MacroConc yield an improvement in running time of up to four orders of magnitude compared to naive applications of AAM. The analyses resulting from the application of ModConc also yield an improvement of up to four orders of magnitude compared to naive applications of AAM, and also yield an improvement of up to four orders of magnitude compared to analyses resulting from the application of MacroConc, although not on the same programs. This has been demonstrated on our benchmark suite counting 56 concurrent programs, ranging from 17 to 293 lines of code. However, it remains to be investigated how such analyses fare in practice on larger programs. Existing model checking analyses featuring state space reduction have been applied to programs of thousands of lines of code (Yang *et al.*, 2008; Abdulla *et al.*, 2014), and existing static analyses featuring a process-modular design have been applied to programs of millions of lines of code (Miné and Delmas, 2015). These existing analyses may take days to verify large programs, and it remains to be investigated how analyses resulting from the application of MacroConc and ModConc will behave on such large programs.

To investigate this, it is necessary to apply MacroConc and ModConc to real-world languages rather than to λ_α and λ_τ. Our implementation is a first step in that direction, as it supports a large subset of Scheme as the base sequential language, rather than the minimal λ_0 of our formalization. However, a number of extensions to the concurrency models are necessary to reflect real-world uses of concurrency. For example, while we included locks in λ_τ, multi-threaded programs may use other means of synchronization such as semaphores, monitors, or `synchronized` blocks. Moreover, it has been shown that developers tend to combine multiple concurrency models (Godefroid and Nagappan, 2008; Tasharofi *et al.*, 2013). Applying MacroConc and ModConc to languages that feature such combinations would prove useful to support real-world concurrent programs.

Finally, we applied MacroConc and ModConc to the design of a communication topology analysis, which infers the communication effects that are generated by each process. Although Chapter 5 presented analyses that infer mailbox bounds and that verify the absence of errors, the extent to which analyses resulting from the application of our design methods support other clients should be investigated. We foresee future applications in program comprehension and automated theorem proving.

7.3.2. Definition of the Restriction Function for Macro-Stepping Semantics

In the application of MacroConc to λ_α and λ_τ, we use simple yet efficient definitions for the restriction function of the macro-stepping semantics (Sections 4.3.3 and 4.4.2). However, other definitions of these restriction functions should be investigated. For example, the restriction function for λ_τ breaks a macro step before a second communication effect is generated, even if performing this effect would not conflict with other processes. Possible refinements could allow for more than one communication effect to be performed within the same macro-step, as for the restriction function for λ_α. This could result in

additional improvements in running time.

Such restriction functions have to be designed carefully, as an incorrect definition may lead to an unsound analysis. For example, we have shown that the original macro-stepping strategy of Agha *et al.* (1997) is not sound for ordered mailbox models (see Section 4.3.1). As MacroConc is applicable to many models of concurrent programs, guidelines on how to devise an efficient and sound restriction function should be formulated.

7.3.3. Applicability of Mailbox Abstractions to Large Programs

We studied several mailbox abstractions in Chapter 5, and evaluated their impact on the precision and running time of analyses resulting from MacroConc. First, additional mailbox abstractions are worth studying. For example, while our finite list abstraction approximates the mailbox to a set once a certain bound is reached, it could use a different abstract representation (e.g., a multiset) for increased precision.

Second, a combination of multiple mailbox abstractions in the analysis of the same program should be investigated. As we have shown, some benchmark programs benefit from an improved precision and running time when specific mailbox abstractions are used, while other programs do not see this improvement. Devising a strategy to abstract the mailbox of multiple actors in the same program differently, according to a heuristic or to user annotations, could further improve the precision and running time on other benchmark programs.

As we have shown, changing the mailbox abstraction results in an improvement in precision with almost no cost in performance for the MacroConc actor analysis. It remains to be seen whether the abstractions we presented are applicable to large programs. However, the possible improvements described here could enable analyses that use more precise abstractions or feature such strategies to support larger programs.

7.3.4. Ordering and Multiplicity Information in Process-Modular Analysis

While analyses resulting from the application of MacroConc exhibit the same precision as analyses resulting from a naive application of AAM for concurrent programs, ModConc sacrifices precision. For a communication topology analysis, we have shown that this loss of precision is minimal in Chapter 6, as on 294 values inferred by the analysis, only four of them are spurious. However, the main cost is the loss of information about ordering and multiplicity in the analyses. As demonstrated through the study of mailbox abstractions in Chapter 5, such information support important applications such as inference of mailbox bounds for concurrent actor programs.

The analyses resulting from the application of ModConc store the communication effects discovered in sets, which lose all ordering and multiplicity information. Our study of mailbox bounds has shown different ways of preserving this information more precisely in the case of an actor analysis. Investigating the use of similar abstractions for the set of communication effects is an interesting avenue for future work.

We also identify two other possible solutions to this limitation. First, Midtgaard *et al.* (2016a) describe an analysis with a process-modular design, targeted at processes that communicate synchronously. Although this process-modular analysis is limited to programs consisting of two processes, ordering and multiplicity information is preserved in its representation of communication between the processes. Extending this work to support more than two processes and to support dynamic process creation could enable preserving ordering and multiplicity information in the results of the intra-process analyses. Second, the Soter tool (D'Osualdo *et al.*, 2012)—against which we extensively compare our work throughout this dissertation—performs an analysis on actor programs that abstracts to sets. However, as demonstrated in Chapter 5, Soter is able to reason about properties that require ordering and multiplicity information. This is because the analysis of Soter constructs a coarse model of the program on which a model checker can be run to verify properties expressed through user-provided code annotations. This second model checking part can restore the ordering and multiplicity information that is lost during the model construction. Using this as inspiration to restore ordering and multiplicity information in analyses resulting from the application of ModConc would enable these analyses to verify properties that require more precision.

7.3.5. Process Sensitivities for Increased Precision

The allocation strategy for processes used in this dissertation (introduced in Chapter 2) allocates processes at addresses that corresponds to their initial code (initial behavior for λ_α, and expression to evaluate for λ_τ). Similarly, the allocation strategy for value addresses and for continuation addresses is a simple 0-CFA strategy where a variable or continuation is allocated at an address that corresponds to a single location in the code. In the literature, a number of other allocation strategies have been presented, from the k-CFA of Shivers (1991) to the unified methodology for polyvariant CFA of Gilray *et al.* (2016a) which consists of 11 different allocation strategies for AAM-based analyses. Such allocation strategies have an influence on the precision and running time an analysis without impacting soundness (Might and Manolios, 2009). No such study of allocation strategies for processes has been performed, and we foresee the following allocation strategies for processes that could improve the precision of the analyses resulting from MacroConc and ModConc.

By instrumenting the operational semantics with a *timestamp* that captures the history of the program's execution, as in the original formulation of AAM (Van Horn and Might, 2010), one could express process allocation strategies that are sensitive to the history of the concurrent operations performed in the program. For example, similarly to the k-CFA allocation strategy allocates a variable at an address that contains information about the last k call sites at the allocation point (Shivers, 1991), a process allocation strategy could include the last k processes created, or the last k communication effects performed, into the process identifiers. As a result, analyses could distinguish processes that are now mapped to the same abstract process. As is the case for k-CFA, this comes at a cost in running time which would have to be further studied to assess the feasibility of such process allocation strategies.

7.3.6. Combining MacroConc and ModConc

While we describe MacroConc and ModConc as two different analysis design methods, they share a common starting point in the transition relations described in Chapter 2. We foresee possible combinations of the two design methods, where part of the program is analyzed by an analysis resulting from the application of MacroConc for high precision but lower efficiency, while other parts of the program are analyzed by an analysis resulting from the application of ModConc for improved running time at the cost of a lower precision. Such combinations need to be investigated and could result in analyses that are at the same time precise at the place where they need to be, and scalable.

7.4. Concluding Remarks

We started this dissertation by motivating the need for static analysis tools that support modern concurrent programs. We presented multiple desirable features of such tools. *Automation* is important to enable developers to use static analysis tools without requiring expertise in static analysis. *Soundness* is crucial to enable verifying properties in such a way that the output of the tool can be trusted. *Scalability* is needed for concurrent programs as such programs exhibit a high degree of non-determinism in their process interleavings, exacerbating the complexity of their behavior. *Precision* is necessary to provide accurate results to the users of a static analysis instead of having them ignore results that are too imprecise. *Support for dynamic process creation* enables analyzing modern concurrent programs that feature dynamic topologies.

In this dissertation, we started from a naive application of the AAM design method to concurrent programs, demonstrating that it achieves most of these desirable properties, except scalability. We identified from the existing static analysis literature on concurrent programs two ways to address this scalability issue, and presented two AAM-based design methods for analyses of concurrent programs, MacroConc and ModConc, each inspired by one of these solutions. MacroConc results in analyses that feature state space reduction through a macro-stepping formulation of the concrete semantics of the analyzed language. This improves the scalability of the resulting analyses without compromising precision, although their worst-case time complexity remains exponential. ModConc results in analyses that feature a process-modular design in which each process of a concurrent program is analyzed in isolation, improving scalability to a polynomial worst-case time complexity, for a small cost in precision.

We applied each of these design methods to a concurrent actor language, λ_α, and to a shared-memory multi-threaded language, λ_τ. We proved the soundness and termination of the resulting analyses formally, and evaluated their running time, precision and scalability empirically. Analyses resulting from the application of MacroConc improve the running time of the analyses by up to four orders of magnitude and achieve a high precision, yet remain subject to scalability issues due to their worst-case exponential time complexity. Analyses resulting from the application of ModConc further improve the running time of the analyses by up to four more orders of magnitude compared to

MacroConc, for a minimal cost in terms of precision. These analyses scale linearly with the number of communication effects performed in the analyzed programs.

Our analysis design methods comprise an important improvement over the state of the art in static analysis support for modern concurrent programs. Several aspects of MacroConc and ModConc can be improved further. These include improving the state space reduction of MacroConc through a more refined effect restriction function, reestablishing ordering information into ModConc analyses for improved precision, investigating the impact of more precise process sensitivities than the one used in this dissertation, and combining the two design methods in the same analysis. Nonetheless, these analysis design methods have been shown to support complex concurrent programs.

In conclusion, MacroConc and ModConc provide a solid foundation for the design of static analyses targeted at modern concurrent programs. Analyses featuring such designs serve the needs of developers of concurrent programs in terms of tool support.

A

NOTATIONS

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems.

— McConnell (2004)

We list here the notations used throughout this dissertation.

A.1. Domains

Domain names

Domain names are either denoted with Latin letters starting with a capital (e.g., *Store*), or with a single Greek letter (e.g., Σ).

Tuples

Multiple domains are paired together using the multiplication operator. If A and B are domains, $A \times B$ is the domain composed of pairs of elements of A and B. Elements of a tuple domain are denoted with angle brackets or parentheses. For example, if $a \in A$ and $b \in B$, we write $\langle a, b \rangle \in A \times B$ or $(a, b) \in A \times B$.

Sums

Multiple domains can be combined with the sum operator to provide choice. If A and B are domains, $A + B$ is the domain containing either elements of A or elements of B. Elements of a sum domain have no special notation. If $a \in A$, then we write $a \in A + B$. Similarly, if $b \in B$, we write $b \in A + B$.

Sequences

A domain consisting of possibly infinite sequences of elements of domain A is denoted A^*. Elements of this domain are concatenated with the : operator. The empty sequence is denoted ϵ. For example, if $a_i \in A$, then $a_1 : a_2 : \epsilon \in A^*$. The size of a sequence $a_1 : \ldots : a_n$ is denoted $|a_1 : \ldots : a_n|$.

Finite sequences

The notation A_n^* denotes a subset of the domain A^* where all sequences have at most n elements, i.e., $a \in A_n^* \iff a \in A^* \wedge |a| \leq n$.

Powerset

The powerset of a domain is the domain composed of sets of element of the domain. For example, $\mathcal{P}(A)$ is the powerset of domain A, and contains elements such as $\{a_1\}$, $\{a_1, a_2\}$, where $a_i \in A$.

Backus-Naur form

We sometimes define domains using the Backus-Naur form. For example, the domain of booleans can be defined as $\mathbb{B} ::= \texttt{\#t} \mid \texttt{\#f}$.

A.2. Functions

Total functions

A total function f is denoted as $f : A \to B$, where A is the *domain* of the function, i.e. $\text{dom}(f) = A$, and the *range* of the function is B, i.e. $\text{range}(f) = B$. A total function maps every element of its domain to an element of its range.

Partial functions

A partial function f is denoted as $f : A \rightharpoonup B$ and behaves similarly as a total function, except that it may not be defined for all elements of its domain.

Extension

Functions are extended using the following notation: $f[a \mapsto b]$ is the function that behaves as f on every element, except on a where it returns b. That is, $f[a \mapsto b](x) = f(x)$ if $x \neq a$, and $f[a \mapsto b](a) = b$.

Membership

We also use the notation $f[a \mapsto b]$ to match over a function f which associates a to b. That is, a transition rule that has $f[a \mapsto b]$ as a premise requires that $f(a) = b$.

Empty function

The empty function is denoted by square brackets: []. In case of partial functions, this function is defined for no elements of its domain. In case of total functions whose range is a lattice, the empty function maps any element from its domain to the bottom element of the lattice, i.e. it is the function $\lambda x.\bot$.

Function joining

When the range of a function is a lattice, joining is extended to the function domains in the following way: $f \sqcup g = \lambda x.f(x) \sqcup g(x)$.

A.3. Sets

Set construction

A set is defined between curly braces: the set $\{1, 2\}$ contains the elements 1 and 2.

Set union

Two sets can be combined through the union operator ($\cup$): $\{1, 2\} \cup \{2, 3\} = \{1, 2, 3\}$.

Set difference

The set difference operator ($\setminus$) applied to sets removes element from the second set that are present in the first set: $\{1, 2\} \setminus \{2, 3\} = \{1\}$.

Set comprehension

Set comprehension is used to construct new sets from iterations over existing sets. For example, $A - B$ can be described by the set comprehension $\{a \mid a \in A \wedge a \notin B\}$.

Big operators

We use big operators to denote an operator applied to a number of elements. For example, the set of all square numbers $\{0^2\} \cup \{1^2\} \cup \{2^2\} \cup \ldots$ can be denoted as follows.

$$\bigcup_{i \in \mathbb{N}} i^2$$

B

PROOFS

> *Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.*
>
> *A: The definitions are almost always wrong.*
>
> — Pierce (2002)

In this appendix, we provide detailed proofs for the theorems presented in the dissertation. Some proofs have been mechanized, and are annotated with a check mark (✓). Note that we overload α to denote multiple abstraction functions, when it is not used in an ambiguous way.

B.1. Proofs for Naive Application of AAM to Concurrent Programs (Chapter 2)

B.1.1. Proofs for Abstract Interpretation of λ_0

We prove soundness of each component of the abstract interpretation of λ_0 separately, and combine the results to prove the soundness of the abstract interpretation of λ_0 at the end of this section.

Addresses

Value addresses and continuation addresses are a parameter of the analysis, of which we give a concrete instantiation in Figure 2.5 (p. 17) and an abstract instantiation in Figure 2.12 (p. 22). We assume that address allocation is sound, which is a sound assumption as Might and Manolios (2009) have proven that any address abstract allocation strategy leads to a sound analysis. We therefore rely on an abstraction function for addresses, $\alpha : Addr \to \widehat{Addr}$.

Assumption 1 (Address allocation is sound)**.**

$$\alpha(\sigma) \sqsubseteq \hat{\sigma} \implies \alpha(alloc(ae, \sigma)) \sqsubseteq \widehat{alloc}(ae, \hat{\sigma})$$
$$\alpha(\rho) \sqsubseteq \hat{\rho} \wedge \alpha(\sigma) \sqsubseteq \hat{\sigma} \wedge \alpha(\Xi) \sqsubseteq \widehat{\Xi} \implies \alpha(kalloc(e, \rho, \sigma, \Xi)) \sqsubseteq \widehat{kalloc}(e, \hat{\rho}, \hat{\sigma}, \widehat{\Xi})$$

Environments

Concrete environments are defined in Figure 2.2 (p. 15), and abstract environments are defined in Figure 2.10 (p. 21).

Definition 1 (Abstraction of environments)**.** *Environments are abstracted using the following abstraction function* $\alpha : Env \to \widehat{Env}$*.*

$$\alpha(\rho) = \lambda x.\alpha(\rho(x))$$

The partial-order relation on environments ($\sqsubseteq$*) is the point-wise ordering relation on functions, i.e.,* $\hat{\rho} \sqsubseteq \hat{\rho}' \iff \forall x, \hat{\rho}(x) \sqsubseteq \hat{\rho}'(x)$

Lemma 1 (Environment lookup is sound)**.**

$$\alpha(\rho) \sqsubseteq \hat{\rho} \implies \forall x, \alpha(\rho(x)) \sqsubseteq \hat{\rho}(x)$$

Proof. This directly follows from the definition of the partial-order relation on environments. □

Lemma 2 (Environment extension is sound)**.**

$$\alpha(a) \sqsubseteq \hat{a} \wedge \alpha(\rho) \sqsubseteq \hat{\rho} \implies \alpha(\rho[x \mapsto a]) \sqsubseteq \hat{\rho}[x \mapsto \hat{a}]$$

Proof. We have the following.

$$\begin{aligned}
\alpha(\rho[x \mapsto a]) &= \lambda y. \begin{cases} \alpha(a) & \text{if } x = y \\ \alpha(\rho(x)) & \text{otherwise} \end{cases} \\
&\sqsubseteq \lambda y. \begin{cases} \hat{a} & \text{if } x = y \\ \hat{\rho}(y) & \text{otherwise} \end{cases} \\
&= \hat{\rho}[x \mapsto \hat{a}]
\end{aligned}$$

Therefore, we have $\alpha(\rho[x \mapsto a]) \sqsubseteq \hat{\rho}[x \mapsto \hat{a}]$. □

Value Stores

Concrete value stores are defined in Figure 2.2 (p. 15), and abstract value stores are defined in Figure 2.10 (p. 21).

Definition 2 (Abstraction of value stores)**.** *Value stores are abstracted using the following abstraction function* $\alpha : \mathit{Store} \to \widehat{\mathit{Store}}$.

$$\alpha(\sigma) = \lambda\hat{a}. \bigcup_{\alpha(a) \sqsubseteq \hat{a}} \alpha(\sigma(a))$$

As for environments, the partial-order relation on value stores is the point-wise partial-ordering on functions, i.e., $\hat{\sigma} \sqsubseteq \hat{\sigma}' \iff \forall \hat{a}, \hat{\sigma}(\hat{a}) \sqsubseteq \hat{\sigma}'(\hat{a})$.

Lemma 3 (Value store lookup is sound)**.**

$$\alpha(a) \sqsubseteq \hat{a} \wedge \alpha(\sigma) \sqsubseteq \hat{\sigma} \implies \alpha(\sigma(a)) \sqsubseteq \hat{\sigma}(\hat{a})$$

Proof. From the premise on the stores, we have $\forall \hat{a}, \alpha(\sigma)(\hat{a}) \sqsubseteq \hat{\sigma}(\hat{a})$. Applying the definition of α, we get:

$$\bigcup_{\alpha(a') \sqsubseteq \hat{a}} \alpha(\sigma(a')) \sqsubseteq \hat{\sigma}(\hat{a})$$

Because of the assumption on addresses ($\alpha(a) \sqsubseteq \hat{a}$), we know that $\alpha(\sigma(a))$ is a member of the left-hand side of the relation, which concludes the proof because $\alpha(\sigma(a)) \sqsubseteq \bigcup_{\alpha(a') \sqsubseteq \hat{a}} \alpha(\sigma(a')) \sqsubseteq \hat{\sigma}(\hat{a})$. □

Lemma 4 (Value store extension is sound)**.**

$$\alpha(a) \sqsubseteq \hat{a} \wedge \alpha(\sigma) \sqsubseteq \hat{\sigma} \wedge \alpha(v) \sqsubseteq \hat{v} \implies \alpha(\sigma[a \mapsto v]) \sqsubseteq \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}]$$

Proof.

$$\begin{aligned}
\alpha(\sigma[a \mapsto v]) &= \lambda\hat{a}. \bigcup_{\alpha(a') \sqsubseteq \hat{a}} \alpha(\sigma[a \mapsto v](a')) \\
&= \lambda\hat{a}. \left(\bigcup_{\alpha(a') \sqsubseteq \hat{a}} \alpha(\sigma(a')) \cup \begin{cases} \{\alpha(v)\} & \text{if } \alpha(a) \sqsubseteq a \\ \varnothing & \text{otherwise} \end{cases} \right) \\
&= \left(\lambda\hat{a}. \bigcup_{\alpha(a') \sqsubseteq \hat{a}} \alpha(\sigma(a')) \right) \sqcup [\hat{a} \mapsto \{\alpha(v)\}] \\
&= \alpha(\sigma) \sqcup [\hat{a} \mapsto \{\alpha(v)\}] \\
&\sqsubseteq \hat{\sigma} \sqcup [\hat{a} \mapsto \{\hat{v}\}]
\end{aligned}$$

□

Continuation Stores

Concrete continuation stores are defined in Figure 2.2 (p. 15), and abstract continuation stores are defined in Figure 2.10 (p. 21).

Definition 3 (Abstraction of continuation stores)**.** *Continuation stores are abstracted using the following abstraction function* $\alpha : KStore \to \widehat{KStore}$*.*

$$\alpha(\Xi) = \lambda\widehat{k}. \bigcup_{\alpha(k) \sqsubseteq \widehat{k}} \alpha(\Xi(k))$$

As for environments ad value stores, the partial-order relation on continuation stores is the point-wise partial-ordering on functions, i.e., $\widehat{\Xi} \sqsubseteq \widehat{\Xi}' \iff \forall \widehat{k}, \widehat{\Xi}(\widehat{k}) \sqsubseteq \widehat{\Xi}'(\widehat{k})$*.*

Lemma 5 (Continuation store lookup is sound)**.**

$$\alpha(k) \sqsubseteq \widehat{k} \wedge \alpha(\Xi) \sqsubseteq \widehat{\Xi} \implies \alpha(\Xi(k)) \sqsubseteq \widehat{\Xi}(\widehat{k})$$

Proof. This proof is identical to the proof of Lemma 3 □

Lemma 6 (Continuation store extension is sound)**.**

$$\alpha(k) \sqsubseteq \widehat{k} \wedge \alpha(\Xi) \sqsubseteq \widehat{\Xi} \wedge \alpha(\kappa) \sqsubseteq \hat{\kappa} \implies \alpha(\Xi[k \mapsto \kappa]) \sqsubseteq \widehat{\Xi} \sqcup [\widehat{k} \mapsto \{\hat{\kappa}\}]$$

Proof. This proof is identical to the proof of Lemma 6 □

Atomic Evaluation

Concrete atomic evaluation is defined in Figure 2.3 (p. 16), and abstract atomic evaluation is defined in Figure 2.11 (p. 22).

Lemma 7 (Atomic evaluation is sound)**.** *If* $\rho, \sigma \vdash ae \Downarrow v$*, then* $\alpha(\rho), \alpha(\sigma) \vdash ae \,\widehat{\Downarrow}\, \hat{v}$ *where* $\alpha(v) \sqsubseteq \hat{v}$*.*

Proof. We have to show that $\alpha(v) \sqsubseteq \hat{v}$. This is done by case analysis on *ae*.

- If $ae = lam$, then $v = \mathbf{clo}(lam, \rho)$, $\hat{v} = \mathbf{clo}(lam, \alpha(\rho))$, and we have $\alpha(v) \sqsubseteq \hat{v}$.
- If $ae = x$, then $v = \sigma(\rho(x))$, $\hat{v} = \alpha(\sigma)(\alpha(\rho)(x))$. By Lemma 3, we have $\alpha(\sigma(\rho(x))) \sqsubseteq \alpha(\sigma)(\alpha(\rho(x)))$.

□

Transition Relation

The concrete transition relation is defined in Figure 2.4 (p. 17), and its abstract version in Figure 2.13 (p. 23).

Lemma 8 (Soundness of the transition relation)**.**

$$\varsigma, \sigma, \Xi \hookrightarrow \varsigma', \sigma', \Xi' \implies \alpha(\varsigma), \alpha(\sigma), \alpha(\Xi) \mathrel{\widehat{\hookrightarrow}} \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}' \\ \wedge\ \alpha(\varsigma') \sqsubseteq \hat{\varsigma}' \wedge \alpha(\sigma) \sqsubseteq \hat{\sigma}' \wedge \alpha(\Xi) \sqsubseteq \widehat{\Xi}'$$

Proof. This is proven by a case analysis on the transition rule. We detail the first two cases, and the remaining cases follow the same reasoning.

- If rule Atomic applies, we have $\varsigma = \langle \mathbf{ev}(ae, \rho), k \rangle$ and $\varsigma' = \langle \mathbf{val}(v), k \rangle$, and the stores remain the same. We also have that $\alpha(\varsigma), \alpha(\sigma), \alpha(\Xi) \mathrel{\widehat{\hookrightarrow}} \langle \mathbf{val}(\hat{v}), \alpha(k) \rangle, \alpha(\sigma), \alpha(\Xi)$. Therefore, we have to show the following:
  - $\alpha(v) \sqsubseteq \hat{v}$, which holds because atomic evaluation is sound (Lemma 7).
- If rule App applies, we have $\varsigma = \langle \mathbf{ev}(\texttt{(}f\ ae\texttt{)}, \rho), k \rangle$, $\varsigma' = \langle \mathbf{ev}(e, \rho'[x \mapsto a]), k \rangle$. By Lemma 7, we know that if $\rho, \sigma \vdash f \Downarrow \mathbf{clo}(\texttt{(}\lambda\texttt{(}x\texttt{)}\ \ e\texttt{)}, \rho')$, then $\alpha(\rho), \alpha(\sigma) \vdash f \mathrel{\widehat{\Downarrow}} \mathbf{clo}(\texttt{(}\lambda\texttt{(}x\texttt{)}\ \ e\texttt{)}, \hat{\rho}')$ where $\alpha(\rho') \sqsubseteq \hat{\rho}'$, and through the same lemma, we know that $\alpha(\rho), \alpha(\sigma) \vdash ae \mathrel{\widehat{\Downarrow}} \hat{v}$ where $\alpha(v) \sqsubseteq \hat{v}$, hence $\hat{\varsigma}' = \langle \mathbf{ev}(e, \hat{\rho}'[x \mapsto \hat{a}]), \alpha(k) \rangle$, $\sigma' = \sigma[a \mapsto v]$, and $\hat{\sigma}' = \alpha(\sigma) \sqcup [\hat{a} \mapsto \{\hat{v}\}]$, where a and $\hat{a}$ result from address allocation. Moreover, we know from Assumption 1 that $\alpha(a) \sqsubseteq \hat{a}$. It remains to show the following:
  - $\alpha(\sigma[a \mapsto v]) \sqsubseteq \alpha(\sigma) \sqcup [\hat{a} \mapsto \{\hat{v}\}]$, which holds by Lemma 4.
  - $\alpha(\rho[x \mapsto a]) \sqsubseteq \alpha(\rho)[x \mapsto \hat{a}]$, which holds by Lemma 2.
- If rule Letrec1 applies, the relation holds: the assumption of soundness on allocation (Assumption 1) shows that the allocated addresses are soundly over-approximated, Lemma 2 shows that the environment is extended in a sound manner, and Lemma 6 shows that the continuation store is extended in a sound manner.
- If rule Letrec2 applies, the relation holds: Lemma 4 shows that the value store is extended in a sound manner, and Lemma 5 shows that the lookup in the continuation store is soundly over-approximated.
- If rule Error applies, the relation holds by applying the definition of α (with $\varsigma = \langle \mathbf{ev}(\texttt{(error)}, \rho), k \rangle$, $\varsigma' = \langle \mathbf{err}, k \rangle$, $\hat{\varsigma}' = \langle \mathbf{err}, \alpha(k) \rangle, \alpha(\sigma), \alpha(k))$

□

Transfer Function

The concrete transfer function is defined in Figure 2.8 (p. 18), and its abstract version in Figure 2.15 (p. 24).

Lemma 9 (Soundness of the transfer function)**.**

$$\forall S, \alpha(\mathcal{F}_e^{\lambda_0}(S)) \sqsubseteq \widehat{\mathcal{F}}_e^{\lambda_0}(\alpha(S))$$

Proof. Unfolding the definitions, we get:

$$\alpha(\{(\mathcal{I}(e), [], [k_0 \mapsto \epsilon])\} \cup \bigcup_{\substack{(\varsigma,\sigma,\Xi)\in S \\ \varsigma,\sigma,\Xi\hookrightarrow\varsigma',\sigma',\Xi'}} (\varsigma', \sigma', \Xi')) \sqsubseteq \left\{(\widehat{\mathcal{I}}(e), [], [\widehat{k_0} \mapsto \epsilon])\right\} \cup \bigcup_{\substack{(\hat{\varsigma},\hat{\sigma},\widehat{\Xi})\in\alpha(S) \\ \hat{\varsigma},\hat{\sigma},\widehat{\Xi} \hookrightarrow \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'}} (\hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}')$$

α is distributive over $\cup$ and this can be reduced to show the following cases.

- $\alpha(\mathcal{I}(e), [], [k_0 \mapsto \epsilon]) \sqsubseteq (\widehat{\mathcal{I}}(e), [], [\widehat{k_0} \mapsto \epsilon])$, which trivially holds by Assumption 1.
- $\alpha(\bigcup_{(\varsigma,\sigma,\Xi)\in S\varsigma,\sigma,\Xi\hookrightarrow\varsigma',\sigma',\Xi'}(\varsigma', \sigma', \Xi')) \sqsubseteq \bigcup_{(\hat{\varsigma},\hat{\sigma},\widehat{\Xi})\in\alpha(S)\hat{\varsigma},\hat{\sigma},\widehat{\Xi} \hookrightarrow \hat{\varsigma}',\hat{\sigma}',\widehat{\Xi}'}(\hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}')$, which holds by Lemma 8.

□

Soundness.

The following fixed-point transfer theorem established by Cousot (2002) is useful to simplify soundness proofs.

Lemma 10 (Soundness of least-fixed points)**.** *For two transfer functions $\mathcal{F}$ and $\widehat{\mathcal{F}}$, an abstraction function α, and a partial-order relation $\sqsubseteq$, we have:*

$$(\forall S, \alpha(\mathcal{F}(S)) \sqsubseteq \widehat{\mathcal{F}}(\alpha(S))) \implies \alpha(\mathrm{lfp}(\mathcal{F})) \sqsubseteq \mathrm{lfp}(\widehat{\mathcal{F}})$$

Theorem 1 (Soundness of $\widehat{\mathcal{F}}_e^{\lambda_0}$)**.** $\alpha(\mathrm{lfp}(\mathcal{F}_e^{\lambda_0})) \sqsubseteq \mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_0})$.

Proof. This is proven applying Lemma 10 followed by Lemma 9. □

Monotonicity

Lemma 11 ($\widehat{\mathcal{F}}_e^{\lambda_0}$ is monotone)**.** $\forall S, S', S \sqsubseteq S' \implies \widehat{\mathcal{F}}_e^{\lambda_0}(S) \sqsubseteq \widehat{\mathcal{F}}_e^{\lambda_0}(S')$

Proof. If $S \sqsubseteq S'$, then $\exists S'', S' = S \cup S''$. Also, from the definition of $\widehat{\mathcal{F}}_e^{\lambda_0}$, it is clear that it is distributive ($\forall A, B, \widehat{\mathcal{F}}_e^{\lambda_0}(A \cup B) = \widehat{\mathcal{F}}_e^{\lambda_0}(A) \cup \widehat{\mathcal{F}}_e^{\lambda_0}(B)$). Therefore, we have to show that:

$$\widehat{\mathcal{F}}_e^{\lambda_0}(S) \sqsubseteq \widehat{\mathcal{F}}_e^{\lambda_0}(S) \cup \widehat{\mathcal{F}}_e^{\lambda_0}(S'')$$

This holds for any S and S''. □

Finiteness

Lemma 12. $\langle \mathcal{P}(\widehat{\Sigma} \times \widehat{Store} \times \widehat{KStore}), \sqsubseteq \rangle$ *is a finite lattice.*

Proof. All components defined in Figure 2.10 (p. 21) are either enumerations based on other components ($\widehat{Control}$, $\widehat{Kont}$, $\widehat{Frame}$, $\widehat{Val}$), tuples of other components ($\widehat{\Sigma}$), functions acting on other components ($\widehat{Env}$, $\widehat{Store}$, $\widehat{KStore}$), or are explicitly finite sets ($\widehat{Addr}$, $\widehat{KAddr}$, *Var*, *Exp*). Hence, the domain $\mathcal{P}(\widehat{\Sigma} \times \widehat{Store} \times \widehat{KStore})$ has finitely many elements. □

Termination

Theorem 2 (Termination)**.** *The computation of* $\mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_0})$ *always terminates.*

Proof. Tarski's fixed-point theorem (Tarski, 1955) states that the least-fixed point of monotone function ($\widehat{\mathcal{F}}_e^{\lambda_0}$, Lemma 11) over a finite lattice ($\langle \mathcal{P}(\Sigma \times Store \times KStore), \subseteq \rangle$, Lemma 12) is reached in a finite number of steps, i.e., always terminates. □

B.1.2. Proofs for Abstract Interpretation of λ_α

Process Identifiers

Process identifiers are a parameter of the analysis, of which we give a concrete instantiation in Figure 2.26 (p. 33) and an abstract instantiation in Figure 2.36 (p. 39). We assume that process identifier allocation is sound, which is a sound assumption as Might and Manolios (2009) have proven that any allocation strategy leads to a sound analysis. We therefore rely on an abstraction function for process identifiers, $\alpha : PID \to \widehat{PID}$.

Assumption 2 (Process identifier allocation is sound)**.**

$$\alpha(\varsigma) \sqsubseteq \hat{\varsigma} \wedge \alpha(\pi) \sqsubseteq \hat{\pi} \implies \alpha(palloc(\varsigma, \pi)) \sqsubseteq \widehat{palloc}(\hat{\varsigma}, \hat{\pi})$$

Process Map

Concrete process maps for λ_α are defined in Figure 2.26 (p. 33), and abstract process maps are defined in Figure 2.36 (p. 39).

Definition 4 (Abstraction of process maps)**.** *Process maps are abstracted using the following abstraction function* $\alpha : \Pi \to \widehat{\Pi}$.

$$\alpha(\pi) = \lambda \hat{p}. \bigcup_{\alpha(p) \sqsubseteq \hat{p}} \alpha(\pi(p))$$

As for environments and value stores, the partial-order relation on continuation stores is the point-wise partial-ordering on functions, i.e., $\hat{\pi} \sqsubseteq \hat{\pi}' \iff \forall \hat{p}, \hat{\pi}(\hat{p}) \sqsubseteq \hat{\pi}'(\hat{p})$.

Lemma 13 (Process map lookup is sound)**.**

$$\alpha(p) \sqsubseteq \hat{p} \wedge \alpha(\pi) \sqsubseteq \hat{\pi} \implies \alpha(\pi(p)) \sqsubseteq \hat{\pi}(\hat{p})$$

Proof. This proof is identical to the proof of Lemma 3 □

Lemma 14 (Process map extension is sound)**.**

$$\alpha(p) \sqsubseteq \hat{p} \wedge \alpha(\pi) \sqsubseteq \hat{\pi} \wedge \alpha(\varsigma) \sqsubseteq \hat{\varsigma} \implies \alpha(\pi[p \mapsto \varsigma]) \sqsubseteq \hat{\pi} \sqcup [\hat{p} \mapsto \{\hat{\varsigma}\}]$$

Proof. This proof is identical to the proof of Lemma 6 □

Mailboxes

Concrete mailboxes are defined in Figure 2.21 (p. 31), and abstract mailboxes are the focus of Chapter 5. The soundness of the different mailbox abstractions is proven in Appendix B.3.

Atomic Evaluation

Atomic evaluation for λ_α is extended from the atomic evaluation of λ_0, in Figure 2.22 (p. 31), and its abstraction is defined in Figure 2.32 (p. 36).

Lemma 15 (Atomic evaluation is sound for λ_τ)**.** *If $\rho, \sigma \vdash ae \Downarrow v$, then $\alpha(\rho), \alpha(\sigma) \vdash ae \,\widehat{\Downarrow}\, \hat{v}$ where $\alpha(v) \sqsubseteq \hat{v}$.*

Proof. Rule Actor is the only rule added, and it mimics rule Closure. We therefore extend the case analysis of the proof of Lemma 7 with one more case, $ae = act$, and we have $v = \mathbf{actor}(act, \rho)$, $\hat{v} = \mathbf{actor}(act, \alpha(\rho))$, and therefore $\alpha(v) \sqsubseteq \hat{v}$. □

Transition Relation

The concrete transition relation for λ_α is defined in Figure 2.23 (p. 32) for the sequential rule (Seq), in Figure 2.24 (p. 32) for the actor management rules (Create and Become), and in Figure 2.25 (p. 33) for messages rules (Send and Process). Their abstract version is given in respectively Figure 2.33 (p. 37), Figure 2.34 (p. 38), and Figure 2.35 (p. 38).

Lemma 16 (Soundness of the transition relation)**.**

$$\begin{aligned}
\pi, \sigma, \Xi \rightsquigarrow_p \pi', \sigma', \Xi' \implies & \alpha(\pi), \alpha(\sigma), \alpha(\Xi) \rightsquigarrow_{\alpha(p)} \hat{\pi}', \hat{\sigma}', \widehat{\Xi}' \\
& \wedge \alpha(\pi') \sqsubseteq \hat{\pi}' \wedge \alpha(\sigma') \sqsubseteq \hat{\sigma}' \wedge \alpha(\Xi') \sqsubseteq \widehat{\Xi}' \\
\pi, \sigma, \Xi \overset{eff}{\rightsquigarrow}_p \pi', \sigma', \Xi' \implies & \alpha(\pi), \alpha(\sigma), \alpha(\Xi) \overset{\widehat{eff}}{\rightsquigarrow}_{\alpha(p)} \hat{\pi}', \hat{\sigma}', \widehat{\Xi}' \wedge \alpha(\pi') \sqsubseteq \hat{\pi}' \\
& \wedge \alpha(\sigma') \sqsubseteq \hat{\sigma}' \wedge \alpha(\Xi') \sqsubseteq \widehat{\Xi}' \wedge \alpha(eff) \sqsubseteq \widehat{eff}
\end{aligned}$$

Proof. We perform a case analysis on the transition rule that applies

- Rule Seq applies. We have $\pi, \sigma, \Xi \rightsquigarrow_p \pi[p \mapsto \varsigma'], \sigma', \Xi'$ and $\alpha(\pi), \alpha(\sigma), \alpha(\Xi) \rightsquigarrow_{\alpha(p)} \alpha(\pi) \sqcup [\alpha(p) \mapsto \hat{\varsigma}'], \hat{\sigma}', \widehat{\Xi}'$. From Lemma 13, we know that $\alpha(\varsigma, mb) \sqsubseteq \alpha(\pi)(\alpha(p))$, and from Lemma 8, we know that $\alpha(\varsigma), \alpha(\sigma), \alpha(\Xi) \,\widehat{\rightsquigarrow}\, \hat{\varsigma}', \hat{\sigma}', \widehat{\Xi}'$ with $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$,

$\alpha(\sigma') \sqsubseteq \hat{\sigma}'$, $\alpha(\Xi') \sqsubseteq \widehat{\Xi}'$. Joining the resulting state in the process map is sound (Lemma 14).

- Rule CREATE applies. Lookup in the process map is sound (Lemma 13), extension of the process map is sound (Lemma 14), allocation of addresses and process identifiers is sound (Assumptions 1 and 2), and atomic evaluation is sound (Lemma 15).
- Rule BECOME applies. Lookup in the process map is sound (Lemma 13), extension of the process map is sound (Lemma 14), atomic evaluation is sound (Lemma 15), and the new abstract state of the process executing the `become` is a sound over-approximation of its concrete counter-part.
- Rule SEND applies. Lookup in the process map is sound (Lemma 13), extension of the process map is sound (Lemma 14), atomic evaluation is sound (Lemma 15), and enqueuing in a mailbox is sound, as proven for each mailbox abstractions in Appendix B.3.
- Rule PROCESS applies. Lookup in the process map is sound (Lemma 13), extension of the process map is sound (Lemma 14) and dequeuing messages from the mailbox is sound (see Appendix B.3).

□

Transfer Function

The transfer function for λ_α is defined in Figure 2.28 (p. 34), and its abstract version is defined in Figure 2.39 (p. 40).

Lemma 17 (Soundness of the transfer function)**.**

$$\forall S, \alpha(\mathcal{F}_e^{\lambda_\alpha}(S)) \sqsubseteq \widehat{\mathcal{F}}_e^{\lambda_\alpha}(\alpha(S))$$

Proof. The proof is identical to the proof of Lemma 9. □

Soundness

Theorem 3 (Soundness)**.** $\alpha(\mathrm{lfp}(\mathcal{F}_e^{\lambda_\alpha})) \sqsubseteq \mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_\alpha})$.

Proof. This is proven by applying Lemma 10 followed by Lemma 17. □

Monotonicity

Lemma 18 ($\widehat{\mathcal{F}}_e^{\lambda_\alpha}$ is monotone)**.** $\forall S, S', S \subseteq S' \implies \widehat{\mathcal{F}}_e^{\lambda_\alpha}(S) \subseteq \widehat{\mathcal{F}}_e^{\lambda_\alpha}(S')$

Proof. This proof follows exactly the same reasoning as Lemma 11: $\widehat{\mathcal{F}}_e^{\lambda_\alpha}$ is distributive for the join operation, hence it is monotone. □

Finiteness

Lemma 19. *$\langle \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}), \subseteq \rangle$ is a finite lattice.*

Proof. With the exception of the $\widehat{Mbox}$ component which is a parameter, all components defined in Figure 2.29 (p. 35) are either enumerations based on other components ($\widehat{Control}$, $\widehat{Val}$, $\widehat{Beh}$), tuples of other components ($\widehat{\Sigma}$, $\widehat{Message}$), functions acting on other components ($\widehat{\Pi}$), or are explicitly finite sets ($\widehat{PID}$). The set mailbox abstraction defined in Figure 2.30 (p. 36), used in the actor analysis of Chapter 2, is finite as well. Hence, the domain $\mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore})$ has finitely many elements. □

Termination

Theorem 4 (Termination)**.** *The computation of* $\mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_\alpha})$ *always terminates.*

Proof. This is ensured by Tarski's fixed-point theorem (Tarski, 1955), and Lemmas 18 and 19. □

B.1.3. Proofs for Abstract Interpretation of λ_τ

Process Identifiers

Similarly as for λ_α, process identifiers are a parameter of the analysis, defined in Figure 2.26 (p. 33) and an abstract instantiation in Figure 2.36 (p. 39). We rely on an abstraction function for process identifiers, $\alpha : PID \to \widehat{PID}$.

Assumption 3 (Process identifier allocation is sound)**.**

$$\alpha(\varsigma) \sqsubseteq \hat{\varsigma} \wedge \alpha(\pi) \sqsubseteq \hat{\pi} \implies \alpha(palloc(\varsigma, \pi)) \sqsubseteq \widehat{palloc}(\hat{\varsigma}, \hat{\pi})$$

Process Map

Process maps for λ_τ follow the same definition as for λ_α, and process maps are defined concretely in Figure 2.43 (p. 45) and in their abstract version in Figure 2.52 (p. 50).

Definition 5 (Abstraction of process maps)**.** *Process maps are abstracted using the following abstraction function* $\alpha : \Pi \to \widehat{\Pi}$.

$$\alpha(\pi) = \lambda \hat{p}. \bigcup_{\alpha(p) \sqsubseteq \hat{p}} \alpha(\pi(p))$$

As for environments and value stores, the partial-order relation on continuation stores is the point-wise partial-ordering on functions, i.e., $\hat{\pi} \sqsubseteq \hat{\pi}' \iff \forall \hat{p}, \hat{\pi}(\hat{p}) \sqsubseteq \hat{\pi}'(\hat{p})$.

Lemma 20 (Process map lookup is sound)**.**

$$\alpha(p) \sqsubseteq \hat{p} \wedge \alpha(\pi) \sqsubseteq \hat{\pi} \implies \alpha(\pi(p)) \sqsubseteq \hat{\pi}(\hat{p})$$

Proof. This proof is identical to the proof of Lemma 3 □

Lemma 21 (Process map extension is sound)**.**

$$\alpha(p) \sqsubseteq \hat{p} \wedge \alpha(\pi) \sqsubseteq \hat{\pi} \wedge \alpha(\varsigma) \sqsubseteq \hat{\varsigma} \implies \alpha(\pi[p \mapsto \varsigma]) \sqsubseteq \hat{\pi} \sqcup [\hat{p} \mapsto \{\hat{\varsigma}\}]$$

Proof. This proof is identical to the proof of Lemma 6 □

Atomic Evaluation

Atomic evaluation is the same as for λ_0.

Lemma 22 (Atomic evaluation is sound for λ_τ)**.** *If* $\rho, \sigma \vdash ae \Downarrow v$*, then* $\alpha(\rho), \alpha(\sigma) \vdash ae \,\widehat{\Downarrow}\, \hat{v}$ *where* $\alpha(v) \sqsubseteq \hat{v}$*.*

Proof. Atomic evaluation remains unchanged for λ_τ with respect to λ_0, hence the proof of Lemma 7 applies. □

Transition Relation

The concrete transition relation rules for λ_τ are given in Figure 2.45 (p. 46) for the sequential rule (Seq), in Figure 2.46 (p. 47) for thread management rules (Spawn and Join), Figure 2.47 (p. 47) for references rules (Ref, Deref, and RefSet), and Figure 2.48 (p. 48) for rules on locks (NewLock, Acquire and Release). Their abstract versions are defined in Figure 2.33 (p. 37), Figure 2.34 (p. 38), and Figure 2.35 (p. 38) respectively.

Lemma 23 (Soundness of the transition relation)**.**

$$\begin{aligned}
\pi, \sigma, \Xi \rightsquigarrow_p \pi', \sigma', \Xi' \implies & \alpha(\pi), \alpha(\sigma), \alpha(\Xi) \rightsquigarrow_{\alpha(p)} \hat{\pi}', \hat{\sigma}', \widehat{\Xi}' \\
& \wedge\ \alpha(\pi') \sqsubseteq \hat{\pi}' \wedge \alpha(\sigma') \sqsubseteq \hat{\sigma}' \wedge \alpha(\Xi') \sqsubseteq \widehat{\Xi}' \\
\pi, \sigma, \Xi \overset{eff}{\rightsquigarrow}_p \pi', \sigma', \Xi' \implies & \alpha(\pi), \alpha(\sigma), \alpha(\Xi) \overset{\widehat{eff}}{\rightsquigarrow}_{\alpha(p)} \hat{\pi}', \hat{\sigma}', \widehat{\Xi}' \wedge \alpha(\pi') \sqsubseteq \hat{\pi}' \\
& \wedge\ \alpha(\sigma') \sqsubseteq \hat{\sigma}' \wedge \alpha(\Xi') \sqsubseteq \widehat{\Xi}' \wedge \alpha(eff) \sqsubseteq \widehat{eff}
\end{aligned}$$

Proof. We perform a case analysis on the transition rule that applies, and the reasoning is similar to the reasoning in the proof of Lemma 16.

- Rule Seq applies. Lookup in the process map is sound (Lemma 20), extension of the process map is sound (Lemma 21, and the sequential transition relation is soundly over-approximated (Lemma 8).
- Rule Spawn applies. Lookup in the process map is sound from Lemma 20, and extension of the process map is sound from Lemma 21.
- Rule Join applies. Apart from lookup in the process and extension of the process map, which are sound (Lemmas 20 and 21), this case relies on the fact that atomic evaluation is sound (Lemma 22).

- Rule Ref applies. This case relies on the soundness of lookup from and extensions to the process map (Lemmas 20 and 21).
- Rule Deref applies. This case relies on the soundness of lookup from and extensions to the process map (Lemmas 20 and 21), and also on soundness of lookup in the store (Lemma 3).
- Rule RefSet applies. This case relies on the soundness of lookup from and extensions to the process map (Lemmas 20 and 21).
- Rule NewLock applies. This case relies on the soundness of lookup from and extensions to the process map (Lemmas 20 and 21).
- Rule Acquire applies. This case relies on the soundness of lookup from and extensions to the process map (Lemmas 20 and 21), and also on soundness of lookup in the store (Lemma 3).
- Rule Release applies. This case relies on the soundness of lookup from and extensions to the process map (Lemmas 20 and 21), and also on soundness of lookup in the store (Lemma 3).

□

Transfer Function

The concrete transfer function for λ_τ is defined in Figure 2.51 (p. 49), and its abstract version is defined in Figure 2.61 (p. 54).

Lemma 24 (Soundness of the transfer function)**.**

$$\forall S, \alpha(\mathcal{F}_e^{\lambda_\tau}(S)) \sqsubseteq \widehat{\mathcal{F}}_e^{\lambda_\tau}(\alpha(S))$$

Proof. The proof is identical to the proof of Lemma 9. □

Soundness

Theorem 5 (Soundness)**.** $\alpha(\mathrm{lfp}(\mathcal{F}_e^{\lambda_\tau})) \sqsubseteq \mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_\tau})$.

Proof. This is proven by applying Lemma 10 followed by Lemma 24. □

Monotonicity

Lemma 25 ($\widehat{\mathcal{F}}_e^{\lambda_\tau}$ is monotone)**.** $\forall S, S', S \subseteq S' \implies \widehat{\mathcal{F}}_e^{\lambda_\tau}(S) \subseteq \widehat{\mathcal{F}}_e^{\lambda_\tau}(S')$

Proof. This proof follows exactly the same reasoning as Lemma 11: $\widehat{\mathcal{F}}_e^{\lambda_\tau}$ is distributive for the join operation, hence it is monotone. □

Finiteness

Lemma 26. *$\langle \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}), \sqsubseteq \rangle$ is a finite lattice.*

Proof. All components defined in Figure 2.52 (p. 50) are either enumerations based on other components ($\widehat{Val}$), tuples of other components function acting on other components ($\widehat{\Pi}$), or are explicitly finite sets ($\widehat{PID}$). Hence, the domain $\mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore})$ has finitely many elements. □

Termination

Theorem 6 (Termination)**.** *The computation of* $\mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_\tau})$ *always terminates.*

Proof. This is ensured by Tarski's fixed-point theorem (Tarski, 1955), and Lemmas 25 and 26. □

B.2. Proofs for Application of MacroConc to Concurrent Programs (Chapter 4)

B.2.1. Proofs for the Application of MacroConc to λ_α

Macro-Stepping

The macro-stepping transfer function for λ_α is defined in Figure 4.4 (p. 86).

Lemma 27 (Soundness of macro-stepping transfer function)**.** *If the effect restriction function f^{λ_α} allows for at most one interfering transition in a macro step, and* $\mathrm{lfp}(\mathit{Macro}\text{-}\mathcal{G}^{\lambda_\alpha}_{p,\pi_0,\sigma_0,\Xi_0}) = \langle S, F \rangle$*, then for every state $\langle \pi, \sigma, \Xi \rangle$ reachable by $\mathcal{F}_e^{\lambda_\alpha}$ on process p starting at state π_0, σ_0, Ξ_0, then either:*

- *there exists a state $\langle \pi', \sigma, \Xi \rangle$ in S such that $\pi'(p) = \pi(p)$, or*
- *there exists a state $\langle \pi', \sigma, \Xi \rangle$ such that $\pi'(p) = \pi(p)$ reachable by $\mathcal{F}_e^{\lambda_\alpha}$ from a state in F.*

Proof. First note that both $\mathcal{G}^{\lambda_\alpha}_{p,\pi_0,\sigma_0,\Xi_0}$ rely on the same transition relation $\rightsquigarrow_p$. From the definition of $\mathcal{G}^{\lambda_\alpha}_{p,\pi_0,\sigma_0,\Xi_0}$, we have:

- States that are added to the set S are states that are reachable from states in S with either no effect performed, or with an effect that does not violate the strategy encoded in the restriction function f^{λ_α}.
- States that are added to the set F are states on which no progress can be made, because either process p has finished its execution, or because any progress would violate the strategy of the restriction function.

This means that the set $\langle S, F \rangle$ consists of (1) states in S that are reachable by only performing transitions on process p that do not violate the macro-stepping strategy, and hence with at most one interfering transition; and (2) of states in F at which the macro-step has been stopped. Because f^{λ_α} allows for at most one interfering transition in a macro step, there is no state $\langle \pi, \sigma, \Xi \rangle$ reachable by $\mathcal{F}_e^{\lambda_\alpha}$ starting at π_0, σ_0, Ξ_0 for which no corresponding state (i.e., for which $\pi(p) = \pi'(p)$) is reachable from $\mathcal{F}_e^{\lambda_\alpha}$ from F. □

Restriction Function

Lemma 28 (Soundness of the restriction function)**.** *The ordered macro-stepping restriction function $f^{\lambda_\alpha}(\mathit{eff})$ does not allow more than one transition acting on the same component of the global state space.*

Proof. All possibly interfering transition rules of the semantics of λ_α are annotated with a communication effect: **c** for actor creation, **b** for behavior change (become), **snd** for message sends, and **prc** for message processing. The ordered macro-stepping restriction function disallows more than one process communication effect per macro step, and also disallows more than one send communication effect per macro step. Moreover, the macro step will always stop after a `become` statement, as the actor transitions to a **wait** state according to rule Become, and only rule Process can be applied to such a state. Altogether, this means that a macro step can only have one of the following sequence of effects, where parentheses denote optionality, and a star superscript (*) denotes any number of effects: (**prc**), (**c**)*, **snd**, (**c**)*, (**b**), or (**prc**), (**c**)*, (**b**). This ensures that at most one interfering transition is contained in a macro step: **c** and **b** effects are not interfering as they cannot be observed by other actors, **prc** are interfering because they dequeue a message from the mailbox of the actor performing the transition, **snd** are interfering because they enqueue a message on the target actor, and **prc** and **snd** are not interfering with each other because they do not act on the mailbox at the same place (**prc** dequeues a message from the front, **snd** enqueues a message to the back). Hence, interleavings of the other transitions performed in the macro step cannot be observed at the level of a single actor, and it is safe to only start further macro steps from states in the set F returned by the macro-stepping transfer function $\mathit{Macro}\text{-}\mathcal{G}^{\lambda_\alpha}_{p,\pi_0,\sigma_0,\Xi_0}$. □

Concrete Macro-Stepping

The global transfer function for the macro-stepping semantics is defined in Figure 4.5 (p. 87).

We first define what local process properties mean: a local process property is a property that only looks at process states in isolation.

Definition 6 (Local process properties)**.** *Local process properties are properties that can be*

expressed on the result of the local function defined as follows.

$$local : \mathcal{P}(\Pi \times Store \times KStore) \to (PID \to \mathcal{P}((\Sigma \times Mbox) \times Store \times KStore))$$
$$local(S) = \lambda p. \bigsqcup_{\substack{(\pi,\sigma,\Xi)\in S \\ p\in \mathrm{dom}(\pi)}} \langle \pi(p), \sigma, \Xi \rangle$$

Lemma 29 (Equivalence of concrete semantics)**.**

$$\mathrm{lfp}(Global\text{-}\mathcal{G}_e^{\lambda_\alpha}) = \langle S, \_ \rangle, \mathrm{lfp}(\mathcal{F}_e^{\lambda_\alpha}) = S' \implies local(S) = local(S')$$

Proof. The initial state is explored by both all-interleavings semantics and macro-stepping semantics. This follows directly from the definition of $\mathcal{F}_e^{\lambda_\alpha}$ and $Global\text{-}\mathcal{G}_e^{\lambda_\alpha}$. We know from Lemma 27 (and because the restriction function is sound, Lemma 28 that $\mathrm{lfp}(Macro\text{-}\mathcal{G}^{\lambda_\alpha}_{p,\pi_0,\sigma_0,\Xi_0})$ is a sound over-approximation of the fixed-point of $\mathcal{F}_e^{\lambda_\alpha}$ restricted to the states reachable from initial state $\langle \pi_0, \sigma_0, \Xi_0 \rangle$ on process p. This means that performing a macro step on a process will not disable possible future transitions due to interleavings of interferences being ignored. As for every iteration of $Global\text{-}\mathcal{G}_e^{\lambda_\alpha}$, a macro step is performed on all running processes, all local process states are accounted for. □

Soundness

Theorem 7 (Soundness)**.** $\mathrm{lfp}(Global\text{-}\widehat{\mathcal{G}}_e^{\lambda_\alpha}) = \langle S, F \rangle, \mathrm{lfp}(\mathcal{F}_e^{\lambda_\alpha}) = S' \implies \alpha(local)(S) = \alpha(local)(S')$

Proof. Because the concrete all-interleavings semantics and the concrete macro-stepping semantics are equivalent (Lemma 29), performing a further sound abstraction as in Chapter 2 is sound. □

Monotonicity

Lemma 30 ($Macro\text{-}\widehat{\mathcal{G}}^{\lambda_\alpha}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}$ and $Global\text{-}\widehat{\mathcal{G}}_e^{\lambda_\alpha}$ are monotone)**.**

$$\forall S, S' \quad S \sqsubseteq S' \implies Macro\text{-}\widehat{\mathcal{G}}^{\lambda_\alpha}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}(S) \subseteq Macro\text{-}\widehat{\mathcal{G}}^{\lambda_\alpha}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}(S')$$
$$\forall S, S' \quad S \sqsubseteq S' \implies Global\text{-}\widehat{\mathcal{G}}_e^{\lambda_\alpha}(S) \subseteq Global\text{-}\widehat{\mathcal{G}}_e^{\lambda_\alpha}(S')$$

Proof. This proof follows exactly the same reasoning as Lemma 11: $Macro\text{-}\widehat{\mathcal{G}}^{\lambda_\alpha}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}$ and $Global\text{-}\widehat{\mathcal{G}}_e^{\lambda_\alpha}$ are distributive for the join operation, hence they are monotone. □

Finiteness

Lemma 31 (*Macro* - $\widehat{\mathcal{G}}^{\lambda_\alpha}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}$ and *Global* - $\widehat{\mathcal{G}}^{\lambda_\alpha}_{e}$ act on finite lattices)**.** *The following lattices are finite.*

$$\langle \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}) \times \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}) \times \mathcal{P}(\widehat{Effect}), \sqsubseteq \rangle$$
$$\langle \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}) \times \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}), \sqsubseteq \rangle$$

Proof. The domain $\mathcal{P}(\Pi \times Store \times KStore)$ is proven finite in Lemma 19. Moreover, *Effect* is a finite domain. Therefore, both the domain of *Macro* - $\widehat{\mathcal{G}}^{\lambda_\alpha}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}$ and the domain of *Global* - $\widehat{\mathcal{G}}^{\lambda_\alpha}_{e}$ are finite. □

Termination

Theorem 8 (Termination)**.** *The computation of* $\mathrm{lfp}(\textit{Global} \text{ - } \widehat{\mathcal{G}}^{\lambda_\alpha}_{e})$ *always terminates.*

Proof. By Tarski's fixed-point theorem (Tarski, 1955), and Lemmas 30 and 31, we know that the computation of $\mathrm{lfp}(\textit{Macro} \text{ - } \widehat{\mathcal{G}}^{\lambda_\alpha}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0})$ always terminates, and therefore that the computation of $\mathrm{lfp}(\textit{Macro} \text{ - } \widehat{\mathcal{G}}^{\lambda_\alpha}_{e})$ always terminates. □

B.2.2. Proofs for the Application of MacroConc to λ_τ

Macro-Stepping

The macro-stepping transfer function for λ_τ is defined in Figure 4.6 (p. 89).

Lemma 32 (Soundness of macro-stepping transfer function)**.** *If the effect restriction function* f^{λ_τ} *allows for at most one interfering transition in a macro step, and* $\mathrm{lfp}(\textit{Macro} \text{ - } \mathcal{G}^{\lambda_\tau}_{p,\pi_0,\sigma_0,\Xi_0}) = \langle S, F \rangle$*, then for every state* $\langle \pi, \sigma, \Xi \rangle$ *reachable by* $\mathcal{F}^{\lambda_\tau}_{e}$ *on process* p *starting at state* π_0, σ_0, Ξ_0*, then either:*

- *there exists a state* $\langle \pi', \sigma, \Xi \rangle$ *in* S *such that* $\pi'(p) = \pi(p)$*, or*
- *there exists a state* $\langle \pi', \sigma, \Xi \rangle$ *such that* $\pi'(p) = \pi(p)$ *reachable by* $\mathcal{F}^{\lambda_\alpha}_{e}$ *from a state in* F*.*

Proof. This is identical to the proof of Lemma 27. □

Restriction Function

Lemma 33 (Soundness of the restriction function)**.** *The ordered macro-stepping restriction function* $f^{\lambda_\alpha}(\mathit{eff})$ *does not allow more than one transition acting on the same component of the global state space.*

Proof. All possibly interfering transition rules of the semantics of λ_τ are annotated with a communication effect. The only rules that are not annotated with effects are the following.

- Rule Seq. It cannot perform nor observe any interference, as it can only evaluate a pure subset of λ_τ.
- Rule Ref. It cannot perform nor observe any interference, as it merely creates a new reference which, at creation time, is not accessible by any other thread.
- Rule NewLock. For the same reason as rule Ref, it cannot perform nor observe any interference.

The definition of f^{λ_τ} implies that a macro step can perform *at most one* communication effect. As only rules that perform communication effect may interfere on the global state, only one transition acting on the global state space is allowed within a macro step. □

Concrete Macro-Stepping

The global transfer function for the macro-stepping semantics is defined in Figure 4.7 (p. 90).

Definition 7 (Local process properties)**.** *Local process properties are properties that can be expressed on the result of the local function defined as follows.*

$$\begin{aligned} &\mathit{local} : \mathcal{P}(\Pi \times \mathit{Store} \times \mathit{KStore}) \to (\mathit{PID} \to \mathcal{P}(\Sigma \times \mathit{Store} \times \mathit{KStore})) \\ &\mathit{local}(S) = \lambda p. \bigsqcup_{\substack{(\pi,\sigma,\Xi)\in S \\ p\in \mathrm{dom}(\pi)}} \langle \pi(p), \sigma, \Xi \rangle \end{aligned}$$

Lemma 34 (Equivalence of concrete semantics)**.**

$$\mathrm{lfp}(\mathit{Global}\text{-}\widehat{\mathcal{G}}_e^{\lambda_\tau}) = \langle S, F \rangle, \mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_\tau}) = S' \implies \mathit{local}(S) = \mathit{local}(S')$$

Proof. The proof follows the same structure as the proof of Lemma 29, and is based on Lemmas 32 and 33. □

Soundness

Theorem 9 (Soundness)**.** $\mathrm{lfp}(\mathit{Global}\text{-}\widehat{\mathcal{G}}_e^{\lambda_\tau}) = \langle S, \_ \rangle, \mathrm{lfp}(\widehat{\mathcal{F}}_e^{\lambda_\tau}) = S' \implies \alpha(\mathit{local})(S) = \alpha(\mathit{local})(S')$

Proof. Because the concrete all-interleavings semantics and the concrete macro-stepping semantics are equivalent (Lemma 34), performing a further sound abstraction as in Chapter 2 is sound. □

Monotonicity

Lemma 35 ($\mathit{Macro}\text{-}\widehat{\mathcal{G}}^{\lambda_\tau}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}$ and $\mathit{Global}\text{-}\widehat{\mathcal{G}}_e^{\lambda_\tau}$ are monotone)**.**

$$\begin{aligned} &\forall S, S', S \sqsubseteq S' \implies \mathit{Macro}\text{-}\widehat{\mathcal{G}}^{\lambda_\tau}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}(S) \subseteq \mathit{Macro}\text{-}\widehat{\mathcal{G}}^{\lambda_\tau}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}(S') \\ &\forall S, S', S \sqsubseteq S' \implies \mathit{Global}\text{-}\widehat{\mathcal{G}}_e^{\lambda_\tau}(S) \subseteq \mathit{Global}\text{-}\widehat{\mathcal{G}}_e^{\lambda_\tau}(S') \end{aligned}$$

Proof. This proof follows exactly the same reasoning as Lemma 11: *Macro* - $\widehat{\mathcal{G}}^{\lambda_\tau}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}$ and *Global* - $\widehat{\mathcal{G}}^{\lambda_\tau}_e$ are distributive for the join operation, hence they are monotone. □

Finiteness

Lemma 36 (*Macro* - $\widehat{\mathcal{G}}^{\lambda_\tau}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}$ and *Global* - $\widehat{\mathcal{G}}^{\lambda_\tau}_e$ act on finite lattices)**.** *The following lattices are finite.*

$$\langle \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}) \times \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}) \times \mathcal{P}(\widehat{Effect}), \sqsubseteq \rangle$$
$$\langle \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}) \times \mathcal{P}(\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}), \sqsubseteq \rangle$$

Proof. The domain $\mathcal{P}(\Pi \times Store \times KStore)$ is proven finite in Lemma 26. Moreover, *Effect* is a finite domain. Therefore, both the domain of *Macro* - $\widehat{\mathcal{G}}^{\lambda_\tau}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0}$ and the domain of *Global* - $\widehat{\mathcal{G}}^{\lambda_\tau}_e$ are finite. □

Termination

Theorem 10 (Termination)**.** *The computation of* $\mathrm{lfp}(\textit{Global} \text{-} \widehat{\mathcal{G}}^{\lambda_\tau}_e)$ *always terminates.*

Proof. By Tarski's fixed-point theorem (Tarski, 1955), and Lemmas 35 and 36, we know that the computation of $\mathrm{lfp}(\textit{Macro} \text{-} \widehat{\mathcal{G}}^{\lambda_\tau}_{\hat{p},\hat{\pi}_0,\hat{\sigma}_0,\widehat{\Xi}_0})$ always terminates, and therefore that the computation of $\mathrm{lfp}(\textit{Global} \text{-} \widehat{\mathcal{G}}^{\lambda_\tau}_e)$ always terminates. □

B.3. Proofs for Mailbox Abstractions (Chapter 5)

We first recapitulate what a sound mailbox abstraction is, and then prove the different mailbox abstractions presented in Chapter 5 sound after repeating the definition of each mailbox abstraction. A mailbox abstraction is sound if the abstraction soundly over-approximates the concrete mailbox. This means that the following equations should hold.

- $\forall mb, size(mb) \leq \widehat{size}(\alpha(mb))$, i.e., the size is soundly over-approximated.
- $\alpha(empty) \sqsubseteq \widehat{empty}$, i.e., the empty abstract mailbox is a sound over-approximation of the empty concrete mailbox.
- $\forall mb, m, \alpha(enq(m, mb)) \sqsubseteq \widehat{enq}(\alpha(mb), \alpha(m)))$, i.e., message enqueuing is sound.
- $\forall m, mb, mb', (m, mb') \in deq(mb) \implies \exists \widehat{mb}', (m, \widehat{mb}') \in \widehat{deq}(\alpha(mb)) \wedge \alpha(mb') \sqsubseteq \widehat{mb}'$, i.e., dequeueing a message from a mailbox is soundly over-approximated.

Note that the soundness of a mailbox abstraction is not influenced by the domain of messages used. This is because a mailbox is merely a container of messages, and a

mailbox therefore does not act upon the values of messages. We therefore prove the soundness of these abstractions in the context of concrete messages. Applying sound abstractions on messages preserve this soundness.

B.3.1. Soundness of the Set Abstraction

The set abstraction is defined in Figure 5.2 (p. 107).

The subsumption relation for the set abstraction is plain equality.

$$\widehat{mb} \sqsubseteq \widehat{mb}' \iff \widehat{mb} = \widehat{mb}'$$

The abstraction function $\alpha_{Set} : Mbox \to Set$ transforms a sequence of messages into a set of messages.

$$\alpha_{Set}(\epsilon) = \varnothing \qquad \alpha_{Set}(m : mb) = \{m\} \cup \alpha_{Set}(mb)$$

We will use the following lemma to simplify the following proofs.

Lemma 37 (α_{Set} can abstract in the reverse order)**.** $\alpha_{Set}(mb : m) = \alpha_{Set}(m : mb)$

Proof. This follows from the fact that sets have no order and is proven by induction on mb. □

Lemma 38 ($\widehat{size}_{Set}$ is sound ✓)**.** $\forall mb, size(mb) \leq \widehat{size}_{Set}(\alpha_{Set}(mb))$

Proof. The proof is by case analysis on mb. Either $mb = \epsilon$, and we have $size(\epsilon) = \widehat{size}_{Set}(\varnothing) = 0$. Or, $mb \neq \epsilon$ and we have $\widehat{size}_{Set}(\alpha_{Set}(m : mb')) = \infty$, and the lemma follows because $\forall n, n \leq \infty$. □

Lemma 39 ($\widehat{empty}_{Set}$ is sound ✓)**.** $\alpha_{Set}(empty) \sqsubseteq \widehat{empty}_{Set}$

Proof. This directly follows from the definition of α_{Set}, because $\alpha_{Set}(empty) = \varnothing$. □

Lemma 40 ($\widehat{enq}_{Set}$ is sound ✓)**.** $\forall m, mb : \alpha_{Set}(enq(m, mb)) \sqsubseteq \widehat{enq}_{Set}(m, \alpha_{Set}(mb))$

Proof. The proof is by induction on mb. Either mb is empty and this holds directly. Otherwise, $mb = m' : mb'$ and we have to show the following inequality (after some unfolding of definitions and use of Lemma 37):

$$\{m'\} \cup \alpha_{Set}(enq(m, mb')) \sqsubseteq \{m, m'\} \cup \alpha_{Set}(mb')$$

Which follows from the induction hypothesis: $\alpha_{Set}(enq(m, mb')) \sqsubseteq \widehat{enq}_{Set}(m, \alpha_{Set}(mb'))$. □

Lemma 41 ($\widehat{deq}_{Set}$ is sound ✓)**.** $\forall m, mb, mb' : (m, mb') \in deq(mb) \implies (m, \alpha(mb')) \in \widehat{deq}_{Set}(\alpha_{Set}(mb))$

Proof. We know from the premise that $(m, mb') \in deq(mb)$, therefore $mb = mb' : m$, and because $\alpha_{Set}(mb' : m) = \{m\} \cup \alpha_{Set}(mb')$, we have to show:

$$(m, \alpha_{Set}(mb')) \in \widehat{deq}_{Set}(\{m\} \cup \alpha_{Set}(mb'))$$

We have two possibilities.

- Either $m \in \alpha_{Set}(mb')$, and we have $\{m\} \cup \alpha_{Set}(mb') = \alpha_{Set}(mb')$. From this and the definition of $\widehat{deq}_{Set}$, we have $(m, \alpha_{Set}(mb')) \in \widehat{deq}_{Set}(\alpha_{Set}(mb))$, which is what we had to prove.
- Or $m \notin \alpha_{Set}(mb')$, and we have $(\{m\} \cup \alpha_{Set}(mb')) \setminus \{m\} = \alpha_{Set}(mb')$. From this and the definition of $\widehat{deq}_{Set}$, we have $(m, \alpha_{Set}(mb')) \in \widehat{deq}_{Set}(\alpha_{Set}(mb))$, which is again what we had to prove.

□

Theorem 11 (*Set* sound ✓)**.** *The set abstraction is a sound mailbox abstraction.*

Proof. This follows from Lemmas 38 to 41. □

B.3.2. Soundness of the Multiset Abstraction

The multiset mailbox abstraction is defined in Figure 5.3 (p. 108)

The abstraction function $\alpha_{MS} : Mbox \to MS$ converts a concrete mailbox represented by list of messages to a multiset, where multisets are represented by functions from messages to naturals.

$$\alpha_{MS}(\epsilon) = \lambda m.0$$
$$\alpha_{MS}(m : mb) = \alpha_{MS}(mb)[m \mapsto \alpha_{MS}(mb)(m) + 1]$$

The partial order relation is multiset equality.

$$\widehat{mb}_1 \sqsubseteq \widehat{mb}_2 \iff \forall m, \widehat{mb}_1(m) = \widehat{mb}_2(m)$$

Lemma 42 (α_{MS} can abstract in the reverse order)**.** $\alpha_{MS}(mb : m) = \alpha_{MS}(m : mb)$

Proof. This follows from the fact that multisets have no order and is proven by induction on mb. □

Lemma 43 ($size_{MS}$ is sound)**.** $\forall mb, size(mb) \leq size_{MS}(\alpha_{MS}(mb))$

Proof. The proof is by induction on mb. With $mb = \epsilon$, we have $0 = 0$. With $mb = m : mb'$, we get:

$$1 + size(mb') = size_{MS}(\alpha_{MS}(m : mb'))$$

We can unfold the abstraction function once, and perform the following, where we let $\widehat{mb}' = \alpha_{MS}(mb')$.

$$\begin{aligned} size_{MS}(\widehat{mb}'[m \mapsto \widehat{mb}'(m)+1]) &= \sum_{m' \in \{m\} \cup \mathrm{dom}(\widehat{mb}')} \widehat{mb}'[m \mapsto \widehat{mb}'(m)+1](m') \\ &= 1 + \widehat{mb}'(m) + \sum_{m' \in \mathrm{dom}(\widehat{mb}') - m} \widehat{mb}'(m') \\ &= 1 + \sum_{m' \in \mathrm{dom}(\widehat{mb}')} \widehat{mb}'(m') \\ &= 1 + size_{MS}(\widehat{mb}') \end{aligned}$$

We therefore have $1 + size(mb') = 1 + size_{MS}(\alpha_{MS}(mb'))$, which holds from the induction hypothesis ($size(mb') = size_{MS}(\alpha_{MS}(mb'))$). □

Lemma 43 was not fully mechanically verified with Coq. It relies on the following assumption which holds but couldn't be mechanically verified:

$$\forall mb, m : size_{MS}(\widehat{enq}_{MS}(m, \alpha_{MS}(mb))) = 1 + size_{MS}(\alpha_{MS}(mb))$$

Lemma 44 ($\widehat{empty}_{MS}$ is sound ✓)**.** $\alpha_{MS}(empty) \sqsubseteq \widehat{empty}_{MS}$

Proof. This follows from the definition of α_{MS}. □

Lemma 45 ($\widehat{enq}_{MS}$ is sound ✓)**.** $\forall m, mb : \alpha_{MS}(enq(m, mb)) \sqsubseteq \widehat{enq}_{MS}(m, \alpha_{MS}(mb))$

Proof. This is proven by induction on mb. If $mb = \epsilon$, we have $[m \mapsto 1] \sqsubseteq [m \mapsto 1]$ which holds by reflexivity. If $mb = mb' : m'$, our induction hypothesis is $\alpha_{MS}(enq(m, mb')) = \widehat{enq}_{MS}(m, \alpha_{MS}(mb'))$, and we have to show that $\alpha_{MS}(enq(m', mb') : m) \sqsubseteq \widehat{enq}_{MS}(m, \alpha_{MS}(m' : mb'))$. The rest of the proof follows by unfolding α_{MS} once on each side, using the induction hypothesis, and showing equality of both sides (which requires performing case analysis on whether $m = m'$). □

Lemma 46 ($\widehat{deq}_{MS}$ is sound ✓)**.** $\forall m, mb, mb' : (m, mb') \in deq(mb) \implies (m, \alpha_{MS}(mb')) \in \widehat{deq}_{MS}(\alpha_{MS}(mb))$

Proof. Because $(m, mb') \in deq(mb)$, we know that $mb = m : mb'$. Let $\widehat{mb} = \alpha_{MS}(mb')[m \mapsto \alpha_{MS}(mb') + 1]$. It is easy to show that both $\alpha_{MS}(mb') = \widehat{mb}[m \mapsto \widehat{mb} - 1]$ and $\widehat{mb} = \alpha_{MS}(mb)$. From the definition of $\widehat{deq}_{MS}$, and because mb contains m (therefore, $\alpha_{MS}(mb)(m) \geq 1$), we know that:

$$(m, \widehat{mb}[m \mapsto \widehat{mb} - 1]) \in \widehat{deq}_{MS}(\widehat{mb})$$

Expanding the definition of $\widehat{mb}$, and with the fact that $\forall x, x + 1 - 1 = x$, we get:

$$(m, \alpha_{MS}(mb')) \in \widehat{deq}_{MS}(\alpha_{MS}(mb))$$

Which is exactly what we had to proof. □

Theorem 12 (*MS* sound)**.** *The multiset abstraction is a sound mailbox abstraction.*

Proof. This follows from Lemmas 43 to 46. □

Because this proof depends on Lemma 43, it is not mechanically verified. Should Lemma 43 be mechanically verified, Theorem 12 would follow.

B.3.3. Soundness of the Finite Multiset Abstraction

The finite multiset abstraction is defined in Figure 5.4 (p. 109).

The abstraction function $\alpha_{MS_n} : Mbox \to MS_n$ converts a concrete mailbox represented by list of messages to a finite multiset, where multisets are represented by functions from messages to naturals.

$$\begin{aligned} \alpha_{MS_n}(\epsilon) &= \lambda x.0 \\ \alpha(m : mb) &= \alpha_{MS_n}(mb)[m \mapsto \alpha_{MS_n}(mb)(m) + 1] && \text{if } \alpha_{MS_n}(mb)(m) < n \\ &= \alpha_{MS_n}(mb)[m \mapsto \infty] && \text{otherwise} \end{aligned}$$

The partial order relation is the following.

$$\widehat{mb}_1 \sqsubseteq_{MS_n} \widehat{mb}_2 \iff \forall m, \widehat{mb}_1(m) = \widehat{mb}_2(m) \vee \widehat{mb}_2(m) = \infty$$

Lemma 47 (α_{MS_n} can abstract in the reverse order)**.** $\alpha_{MS_n}(mb : m) = \alpha_{MS_n}(m : mb)$

Proof. This follows from the fact that multisets have no order and is proven by induction on mb. □

Lemma 48 ($size_{MS_n}$ is sound)**.** $\forall mb, size(mb) \leq size_{MS_n}(\alpha_{MS_n}(mb))$

Proof. Either $\alpha_{MS_n}(mb)$ contains an element m associated to ∞, and the property trivially holds, as the right-hand-side is ∞. If $\alpha_{MS_n}(mb)$ contains no such element, the property proof reduces to Lemma 43. □

Similarly to Lemma 43, we couldn't mechanically verify that proof. However, should Lemma 43 be mechanically verified, Lemma 48 would follow.

Lemma 49 ($\widehat{empty}_{MS_n}$ is sound ✓)**.** $\alpha_{MS_n}(empty) \sqsubseteq_{MS_n} \widehat{empty}_{MS_n}$

Proof. This follows from the definition of α_{MS_n}. □

Lemma 50 ($\widehat{enq}_{MS_n}$ is sound ✓)**.** $\forall m, mb : \alpha_{MS_n}(enq(m, mb)) \sqsubseteq_{MS_n} \widehat{enq}_{MS_n}(m, \alpha_{MS_n}(mb))$

Proof. This proof is by induction on mb and is identical to the proof of Lemma 45. □

Lemma 51 ($\widehat{deq}_{MS_n}$ is sound ✓)**.** $\forall m, mb, mb' : (m, mb') \in deq(mb) \implies \exists \widehat{mb}', (m, \widehat{mb}') \in \widehat{deq}_{MS_n}(\alpha_{MS_n}(mb)) \wedge \alpha(mb') \sqsubseteq_{MS_n} \widehat{mb}'$

Proof. We know from the premise that $mb = mb' : m$. This proof is by case analysis on $\alpha(mb')(m)$

- If $\alpha(mb')(m) = \infty$, we also have $\alpha_{MS_n}(mb)(m) = \infty$ and $\alpha_{MS_n}(mb') = \alpha_{MS_n}(mb)$ and from the definition of $\widehat{deq}_{MS_n}$, we have $(m, \alpha_{MS_n}(mb')) \in \widehat{deq}_{MS_n}(\alpha_{MS_n}(mb))$. Therefore, by taking $\widehat{mb}' = \alpha_{MS_n}(mb')$, the lemma holds by reflexivity of $\sqsubseteq_{MS_n}$.
- If $\alpha_{MS_n}(mb')(m) \in \mathbb{N}$, we either have:
  - $\alpha_{MS_n}(mb)(m) = \infty$, in which case $\alpha(mb')(m) = n$ where n is the bound. We fix $\widehat{mb}' = \alpha_{MS_n}(mb')[m \mapsto \infty]$. It follows from the definition of $\sqsubseteq_{MS_n}$ that $\alpha_{MS_n}(mb') \sqsubseteq_{MS_n} \widehat{mb}'$. Note that $\alpha_{MS_n}(mb) = \alpha_{MS_n}(mb')[m \mapsto \infty]$. From this and the definition of $\widehat{deq}_{MS_n}$, we therefore have $(m, \widehat{mb}') \in \widehat{deq}_{MS_n}(m, \alpha_{MS_n}(mb))$, which is what remained to prove.
  - $\alpha_{MS_n}(mb)(m) \in \mathbb{N}$. The finite multiset $\alpha_{MS_n}(mb)$ therefore has the same behavior as the multiset $\alpha_{MS}(mb)$ for message m, and the lemma follows from Lemma 46.

□

Theorem 13 (MS_n sound)**.** *The finite multiset abstraction is a sound mailbox abstraction.*

Proof. This follows from Lemmas 48 to 51. □

Again, because this proof depends on Lemma 48, which itself depends on Lemma 43, this could be mechanically verified. Should Lemma 43 be mechanically verified, Theorem 13 would follow.

B.3.4. Soundness of the Finite List Abstraction

The finite list mailbox abstraction is defined in Figure 5.5 (p. 110).

The abstraction function $\alpha_{L_n} : Mbox \to L_n$ represents a mailbox by a list only if its length does not exceed the bound, otherwise it represents it as a set.

$$\begin{aligned} \alpha_{L_n}(mb) &= mb && \text{if } |mb| \leq n \\ &= \alpha_{Set}(mb) && \text{otherwise} \end{aligned}$$

The partial order relation is plain equality if both mailboxes are from the same domain. Otherwise, it is set equality after converting the mailbox represented by a list into a set.

$$\begin{aligned}
\widehat{mb}_1, \widehat{mb}_2 \in \mathit{Mbox} \wedge \widehat{mb}_1 = \widehat{mb}_2 &\implies \widehat{mb}_1 \sqsubseteq_{L_n} \widehat{mb}_2 \\
\widehat{mb}_1 \in \mathit{Mbox} \wedge \widehat{mb}_2 \in \mathit{Set} \wedge \alpha_{\mathit{Set}}(\widehat{mb}_1) = \widehat{mb}_2 &\implies \widehat{mb}_1 \sqsubseteq_{L_n} \widehat{mb}_2 \\
\widehat{mb}_1 \in \mathit{Set} \wedge \widehat{mb}_2 \in \mathit{Mbox} \wedge \widehat{mb}_1 = \alpha_{\mathit{Set}}(\widehat{mb}_2) &\implies \widehat{mb}_1 \sqsubseteq_{L_n} \widehat{mb}_2 \\
\widehat{mb}_1, \widehat{mb}_2 \in \mathit{Set} \wedge \widehat{mb}_1 = \widehat{mb}_2 &\implies \widehat{mb}_1 \sqsubseteq_{L_n} \widehat{mb}_2
\end{aligned}$$

Lemma 52 (size_{L_n} is sound ✓)**.** $\forall mb, \mathit{size}(mb) \leq \mathit{size}_{L_n}(\alpha_{L_n}(mb))$

Proof. This is by case analysis on $\alpha_{L_n}(mb)$. Either this results in a set, and the inequality holds due to Lemma 38. Or, this results in a list, and we have to show that $\mathit{size}(mb) \leq \mathit{size}(mb)$, which holds by reflexivity of $\leq$. □

Lemma 53 ($\widehat{\mathit{empty}}_{L_n}$ is sound ✓)**.** $\alpha_{L_n}(\mathit{empty}) = \widehat{\mathit{empty}}_{L_n}$

Proof. This follows from the definition of α_{L_n}. □

Lemma 54 ($\widehat{enq}_{L_n}$ is sound ✓)**.** $\forall m, mb : \alpha_{L_n}(\mathit{enq}(m, mb)) = \widehat{enq}_{L_n}(m, \alpha_{L_n}(mb))$

Proof. This is with a simple case analysis on the size of the mailbox.

- Either $|mb| \leq n$, and $\alpha_{L_n}(mb) = mb$. By unfolding the definition of $\widehat{enq}_{L_n}$, we arrive at $\alpha_{L_n}(\mathit{enq}(m, mb)) = \alpha_{L_n}(\mathit{enq}(m, mb))$, which holds by reflexivity.
- Or, $|mb| > n$, and the mailbox is abstracted by a set: $\alpha_{L_n}(mb) = \alpha_{\mathit{Set}}(mb)$. Since enqueuing can only increase the size of the mailbox, $\mathit{enq}(m, mb)$ will also be abstracted by a set. We therefore have to show that $\alpha_{\mathit{Set}}(\mathit{enq}(m, mb)) = \widehat{enq}_{\mathit{Set}}(m, \alpha_{\mathit{Set}}(mb))$, which directly follows from Lemma 40.

□

Lemma 55 ($\widehat{deq}_{L_n}$ is sound ✓)**.** $\forall m, mb, mb' : (m, mb') \in \mathit{deq}(mb) \implies \exists \widehat{mb}', (m, \widehat{mb}') \in \widehat{deq}_{L_n}(\alpha_{L_n}(mb)) \wedge \alpha_{L_n}(mb') \sqsubseteq_{L_n} \widehat{mb}'$

Proof. Because $(m, mb') \in \mathit{deq}(mb)$, we know that $mb = mb' : m$. The proof follows by case analysis on the size of the mailbox.

- Either $|mb| \leq n$, and therefore $\alpha_{L_n}(mb) = mb$, which implies that $\widehat{deq}_{L_n}(\alpha_{L_n}(mb)) = \mathit{deq}(mb)$. By fixing $\widehat{mb}' = mb'$, the lemma holds for this case: $(m, mb') \in \mathit{deq}(mb)$ by the premise, and $mb' \sqsubseteq_{L_n} mb'$ by reflexivity of $\sqsubseteq_{L_n}$.
- Or $|mb| > n$, and therefore $\alpha_{L_n}(mb) = \alpha_{\mathit{Set}}(mb)$.

- If $|mb'| = n$, we fix $\widehat{mb}' = \alpha_{Set}(mb')$. We have $(m, \alpha_{Set}(mb')) \in \widehat{deq}_{Set}(\alpha_{Set}(mb))$ by Lemma 41, therefore $(m, \alpha_{Set}(mb')) \in \widehat{deq}_{L_n}(\alpha_{Set}(mb))$. Moreover, $\alpha_{L_n}(mb') \sqsubseteq_{L_n} \alpha_{Set}(mb')$, since the left-hand-side is a list, and the right-hand-side is the set abstraction of that same list.
- If $|mb'| > n$, we fix $\widehat{mb}' = \alpha_{L_n}(mb')$, and the rest of the proof follows from Lemma 41 and reflexivity of $\sqsubseteq_{L_n}$, since both $\alpha_{L_n}(mb')$ and $\alpha_{L_n}(mb)$ are elements of *Set*.
- We cannot have $|mb'| < n$, because we have $|mb| > n$ and $mb = m : mb'$, therefore $|mb| = |mb'| + 1$.

□

Theorem 14 (L_n sound ✓)**.** *The finite list abstraction with bound $n > 0$ is a sound mailbox abstraction.*

Proof. This follows from Lemmas 52 to 55. □

B.3.5. Soundness of the Graph Abstraction

The graph mailbox abstraction is defined in Figure 5.7 (p. 113).

PathLength returns the length of the single path between *from* and *to*, if it exists, and is unique. An over-approximative (but suitable) definition is the following.

$$\begin{aligned} \mathit{PathLength}(\mathit{from}, \mathit{to}, \langle V, E\rangle) &= \infty \text{ if } \exists m, |\{m' \mid (m, m') \in E\}| > 1 \vee \mathit{cycle}(\langle V, E\rangle) \\ &= 0 \text{ if } \mathit{from} = \mathit{to} \wedge \mathit{to} \notin E(\mathit{from}) \\ &= 1 + \mathit{PathLength}(\mathit{next}, \mathit{to}, \langle V, E\rangle) \text{ if } \{\mathit{next}\} \in E(\mathit{from}) \end{aligned}$$

It is over-approximative because there might be a unique path between *from* and *to*, but graph $\langle V, E\rangle$ might for example contain a loop that is not connected to this path, and *PathLength* would over-approximate with ∞. The abstraction function consists of enqueuing elements of the list of messages on the empty graph in reverse order, e.g., $\alpha(m_1 : m_2 : m_3) = \widehat{enq}_G(m_3, \widehat{enq}_G(m_2, \widehat{enq}_G(m_1, \bot)))$. This is done through an auxiliary abstraction function $\alpha_G^{\widehat{mb}}$: abstracting a sequence of message is performed by enqueuing the first message in $\widehat{mb}$, followed by enqueuing the remainder of messages with $\alpha_G^{\widehat{mb}'}$ where $\widehat{mb}' = \widehat{enq}_G(m, \widehat{mb})$.

$$\begin{aligned} \alpha_G(mb) &= \alpha_G^{\bot}(mb) \\ \alpha_G^{\widehat{mb}}(\epsilon) &= \widehat{mb} \\ \alpha_G^{\widehat{mb}}(m : mb) &= \alpha_G^{\widehat{enq}_G(m, \widehat{mb})}(mb) \end{aligned}$$

The partial order relation in this case is the following.

$$\frac{}{\bot \sqsubseteq_G \bot} \qquad \frac{V \subseteq V' \wedge E \subseteq E'}{\langle V, E, f, l\rangle \sqsubseteq_G \langle V', E', f, l\rangle}$$

Lemma 56 ($size_G$ is sound)**.** $\forall mb, size(mb) \leq size_G(\alpha_{L_n}(mb))$

Proof. This proof requires to show that when the graph is in fact a single trace of length n from f to l, it represents the corresponding concrete mailbox of length n with the same elements, with full precision, and that *PathLength* computes this length n, or returns ∞.

- When ∞ is returned, the property holds because $\forall n, n \leq \infty$.
- When *PathLength* returns n, we know that each node has a single successor and there are no cycle in the graph. Therefore, there exists a single path between *from* and *to* which contains distinct nodes.

Therefore, *PathLength* computes the length of the single path from f to l if it exists and is unique. This is the case only if the mailbox has been created by successfully enqueuing $n+1$ different messages, and therefore has length $n+1$, which is what $size_G$ returns. □

Because this proof is based on non-simple graph operations, it is rather difficult to mechanically verify with Coq, hence it was not mechanically verified.

Lemma 57 ($\widehat{empty}_G$ is sound ✓)**.** $\alpha_G(empty) = \widehat{empty}_G$

Proof. This follows from the definition of α_G. □

Lemma 58 ($\widehat{enq}_G$ is sound ✓)**.** $\forall m, mb : \alpha_G(enq(m, mb)) = \widehat{enq}_G(m, \alpha_G(mb))$

Proof. We first prove that $\forall m, mb, \widehat{mb} : \alpha_G^{\widehat{mb}}(enq(m, mb)) = \widehat{enq}_G(m, \alpha_G^{\widehat{mb}}(mb))$. This is by induction on mb. Either $mb = \epsilon$ and we have:

$$\alpha_G^{\widehat{mb}}(enq(m, \epsilon)) = \widehat{enq}_G(m, \alpha_G^{\widehat{mb}}(\epsilon))$$
$$\alpha_G^{\widehat{mb}}(m : \epsilon) = \widehat{enq}_G(m, \alpha_G^{\widehat{mb}}(\epsilon))$$

From the definition of $\alpha_G^{\widehat{mb}}$, we get:

$$\widehat{enq}_G(m, \widehat{mb}) = \widehat{enq}_G(m, \widehat{mb})$$

Or, $mb = m' : mb'$ and we have to show the following:

$$\alpha_G^{\widehat{mb}}(enq(m, m' : mb')) = \widehat{enq}_G(m, \alpha_G^{\widehat{mb}}(m' : mb'))$$

We can rewrite the left hand side as follows.

$$\alpha_G^{\widehat{mb}}(enq(m, m' : mb')) = \alpha_G^{\widehat{mb}}(m' : enq(m, mb'))$$
$$= \alpha_G^{\widehat{enq}_G(m', \widehat{mb})}(enq(m, mb'))$$

From the induction hypothesis, $\forall \widehat{mb}, \alpha_G^{\widehat{mb}}(enq(m, mb')) = \widehat{enq}_G(m, \alpha_G^{\widehat{mb}}(mb'))$, we get $\widehat{enq}_G(m, \alpha_G^{\widehat{enq}_G(m', \widehat{mb})}(mb'))$, which is equal to the right-hand side after unfolding the definition of $\alpha_G^{\widehat{mb}}$ once. To prove soundness of $\widehat{enq}_G$, it suffices to notice that the theorem we just proved is a more general case. By fixing $\widehat{mb} = \bot$, we finish our proof. □

Lemma 59 ($\widehat{deq}_G$ is sound)**.** $\forall m, mb, mb' : (m, mb') \in deq(mb) \implies \exists \widehat{mb}', (m, \widehat{mb}') \in \widehat{deq}_G(\alpha_G(mb)) \wedge \alpha_G(mb') \sqsubseteq_G \widehat{mb}'$

Proof. We know from the premise that $mb = mb' : m$. We perform a case analysis on mb'.

- Either $mb' = \epsilon$. Therefore, $\alpha_G(mb) = \langle \{m\}, \varnothing, m, m \rangle$, and from the definition of $\widehat{deq}_G$, we have: $(m, \bot) \in \widehat{deq}_G(\alpha_G(mb))$. This is what we had to prove, because $\alpha(mb') = \bot$.
- Or, $mb' = mb'' : m'$. What we have to show then is the following.

  $$\exists \widehat{mb}', (m, \widehat{mb}') \in \widehat{deq}_G(\alpha_G^{\langle \{m,m'\}, \{\langle m,m' \rangle\}, m', m \rangle}(mb'')) \wedge \alpha_G^{\langle \{m'\}, \varnothing, m', m' \rangle}(mb'') \sqsubseteq_G \widehat{mb}'$$

  We observe that $\widehat{enq}_G$ is monotone on the level of the graph: it only adds nodes and edges. It also keeps the last element constant. Therefore, we know that $\alpha_G^{\langle \{m'\}, \varnothing, m', m' \rangle}(mb'')$ will have the form $\langle V, E, m', l \rangle$. We also know that $\alpha_G^{\langle \{m,m'\}, \{\langle m,m' \rangle\}, m, m' \rangle}(mb'')$ will have the form $\langle V', E', m, l' \rangle$ where $V \subseteq V'$ and $E \subseteq E'$. Furthermore, we have $(m, m') \in E'$. Because of that and by the definition of $\widehat{deq}_G$, we have $(m, \langle V', E', f', m' \rangle) \in \widehat{deq}_G(\alpha_G^{\langle \{m,m'\}, \{\langle m,m' \rangle\}, m, m' \rangle}(mb''))$. We finally observe that $\forall x$, the l element of $\alpha_G^x(mb)$ is independent of x. Therefore, we have $l = l'$. We conclude the proof by fixing $\widehat{mb}' = \langle V', E', m', l \rangle$, because $\langle V, E, m', l \rangle \sqsubseteq_G \langle V', E', m', l \rangle$.

  □

The foundations of this proof have been mechanically verified, but not the proof itself.

Theorem 15 (G sound)**.** *The graph abstraction is a sound mailbox abstraction.*

Proof. This follows from Lemmas 56 to 59. □

Because this proof depends on Lemmas 56 and 59, it couldn't fully be verified.

B.4. Proofs for Application of ModConc to Concurrent Programs (Chapter 6)

B.4.1. Proofs for the Application of ModConc to λ_α

Soundness of Intra-Process Analysis

The intra-process analysis for λ_α is defined in Figure 6.6 (p. 135), and its state space is defined in Figure 6.5 (p. 134).

Lemma 60 (The intra-process analysis is sound)**.** $\mathrm{lfp}(\mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\widehat{mb}})$ *is a sound over-approximation of the fixed point of* $\widehat{\mathcal{F}}^{\lambda_\alpha}$ *restricted to states reachable with transitions on process* $\hat{p}$ *from the initial state* $\langle[\hat{p} \mapsto \left\{(\hat{\varsigma}_0, \widehat{mb})\right\}], \hat{\sigma}_0, \widehat{\Xi}_0\rangle$.

Proof. This is proven by a case analysis on the transition rules that can apply.

- When rule Seq applies, the sequentialized transition relation rule is identical to the transition rule used by $\widehat{\mathcal{F}}^{\lambda_\alpha}_e$.
- When rule Create applies, the resulting state and stores are soundly over-approximated, while a soundly over-approximating actor state is added to the set of created actors.
- When rule Become applies, the sequentialized transition rule is identical to the transition rule used by $\widehat{\mathcal{F}}^{\lambda_\alpha}_e$, as no interferences arise.
- When rule Send applies, the resulting state and stores are soundly over-approximated, while a soundly over-approximating message is stored in the set of sent messages.
- When rule Process applies, the resulting state and stores are soundly over-approximated. Also, the message processed is extracted from the abstract mailbox which is represented by a set. The non-determinism resulting from the use of a set ensures that all possible message orderings and multiplicities are accounted for.

Hence, the behavior of the actor with process identifier $\hat{p}$ is soundly over-approximated for a specific mailbox $\widehat{mb}$ and specific stores $\hat{\sigma}_0$ and $\widehat{\Xi}_0$. □

Soundness of Inter-Process Analysis

The inter-process analysis for λ_α is defined in Figure 6.9 (p. 137), and its state space is defined in Figure 6.7 (p. 135).

Theorem 16 (The inter-process analysis is sound)**.** $\mathrm{lfp}(\mathit{Inter}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_e)$ *is a sound over-approximation of* $\mathrm{lfp}(\mathcal{F}^{\lambda_\alpha}_e)$.

Proof. First, the initial state is soundly over-approximated. Second, every sent message is added to the mailbox of the target actor, and the target actor is re-explored by the intra-process analysis, which is itself sound, hence every sent message is accounted for in a sound manner. Finally, every newly created process is analyzed by the intra-process analysis, which is sound. Each run of the intra-process analysis may discover newly created processes and new messages sent, and the inter-process analysis will therefore reconsider the affected actors for analysis. The components of intra-analysis states ($\widehat{\mathit{IntraState}}$) only increase in size, as no rule removes elements from any component. When the fixed point is reached, every created actor has been analyzed with a mailbox that accounts for all the messages sent to this actor, hence the result of the inter-process analysis is sound. □

Monotonicity

Lemma 61 (*Intra* - $\widehat{\mathcal{H}}^{\lambda_\alpha}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\widehat{mb}}$ and *Inter* - $\widehat{\mathcal{H}}^{\lambda_\alpha}_{e}$ are monotone)**.**

$$\forall S, S' \quad S \subseteq S' \implies \mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\widehat{mb}}(S) \subseteq \mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\widehat{mb}}(S')$$
$$\forall S, S' \quad S \subseteq S' \implies \mathit{Inter}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{e}(S) \subseteq \mathit{Inter}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{e}(S')$$

Proof. This proof follows the same reasoning as Lemma 11: both transfer functions are distributive for the join operation, hence they are monotone. □

Finiteness

Lemma 62. $\langle \widehat{\mathit{IntraState}}, \sqsubseteq \rangle$ *and* $\langle \widehat{\Pi} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}}, \sqsubseteq \rangle$ *are finite lattices*

Proof. Elements of $\widehat{\mathit{IntraState}}$ are tuples of components that are finite (see Lemma 19), and hence $\widehat{\mathit{IntraState}}$ is finite itself. Elements of $\widehat{\Pi}$ are functions of which the domain ($\widehat{\mathit{PID}}$) and the range ($\widehat{\mathit{IntraState}} \times \widehat{\Sigma}$) are finite, and hence $\widehat{\Pi} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KStore}}$ is also finite. □

Termination

Theorem 17 (Termination)**.** *The computation of* $\mathrm{lfp}(\mathit{Inter}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{e})$ *always terminates.*

Proof. By Tarski's fixed-point theorem (Tarski, 1955) Lemmas 61 and 62., we know that the computation of $\mathrm{lfp}(\mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\widehat{mb}})$ terminates, and that the computation of $\mathrm{lfp}(\mathit{Inter}\text{-}\widehat{\mathcal{H}}^{\lambda_\alpha}_{e})$ also terminates. □

B.4.2. Proofs for the Application of ModConc to λ_τ

Soundness of Intra-Process Analysis

The intra-process analysis for λ_τ is defined in Figure 6.16 (p. 143), and its state space is defined in Figure 6.15 (p. 141)..

Lemma 63 (The intra-process analysis is sound)**.** $\mathrm{lfp}(\mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\tau}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\hat{J}})$ *is a sound over-approximation of the fixed point of* $\widehat{\mathcal{F}}^{\lambda_\tau}$ *restricted to states reachable with transitions on process* $\hat{p}$ *from the initial state* $\langle [\hat{p} \mapsto \{\hat{\varsigma}_0\}], \hat{\sigma}_0, \widehat{\Xi}_0 \rangle \bigsqcup_{\substack{\hat{p} \in \mathrm{dom}(\hat{J}) \\ \hat{v} \in \hat{J}(\hat{p})}} \langle [\hat{p} \mapsto \left\{ \mathbf{val}(\hat{v}), \widehat{k}_0 \right\}], [], [] \rangle$.

Proof. This is proven by a case analysis on the transition rules that can apply.

- When one of the rule Seq, Ref, and NewLock applies, the sequentialized transition relation rule is identical to the transition rule used by $\widehat{\mathcal{F}}^{\lambda_\tau}_{e}$, as there are no possible interferences with other processes.

- When rule CREATE applies, the resulting state and stores are soundly over-approximated by the sequentialized transition relation, while the state of the created thread is also soundly over-approximated and is added to the set of created threads.
- When rule JOIN applies, the resulting state and stores are soundly over-approximated by the sequentialized transition relation. The value returned by the call to `join` is a sound over-approximation assuming that the join store parameter $\hat{J}$ is sound.
- When one of the rules DEREF, REFSET, ACQUIRE, and RELEASE applies, the resulting state and stores are soundly over-approximated, assuming that the stores given as parameters (σ_0 and Ξ_0) are sound. The accessed or modified address is added to the set of addresses of the intra-process analysis state.

Hence, the behavior of the thread with process identifier $\hat{p}$ is soundly over-approximated for specific stores $\hat{\sigma}_0$ and $\widehat{\Xi}_0$, and join store $\hat{J}$. □

Soundness of Inter-Process Analysis

The inter-process analysis for λ_τ is defined in Figure 6.19 (p. 145), and its state space is defined in Figure 6.17 (p. 143).

Theorem 18 (The inter-process analysis is sound)**.** $\mathrm{lfp}(\mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\tau})$ *is a sound over-approximation of* $\mathrm{lfp}(\mathcal{F}_e^{\lambda_\tau})$.

Proof. The initial state is soundly over-approximated. Every created thread is analyzed by the intra-process analysis, which is sound. Every thread of which the execution has reached its end state is added in the join store, which is passed when considering new threads for intra-process analysis. Moreover, every thread which depends on a thread for which a new return value has been inferred is reanalyzed to account for this information. Finally, every pair of threads that may conflicts are re-analyzed. Through the fixed-point formulation, this means that eventually, all created threads are analyzed by the intra-process analysis with a sound over-approximation of the value store, the continuation store and the join store. The components of intra-analysis states ($\widehat{\mathit{IntraState}}$) only increase in size, as no rule removes elements from any component. Hence, the result of the inter-process analysis of a λ_τ program e is a sound over-approximation of $\mathcal{F}_e^{\lambda_\tau}$. □

Monotonicity

Lemma 64 ($\mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\tau}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\hat{J}}$ and $\mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\tau}$ are monotone)**.**

$$\forall S, S', S \subseteq S' \implies \mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\tau}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\hat{J}}(S) \subseteq \mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\tau}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\hat{J}}(S')$$
$$\forall S, S', S \subseteq S' \implies \mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\tau}(S) \subseteq \mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\tau}(S')$$

Proof. This proof follows the same reasoning as Lemma 11: both transfer functions are distributive for the join operation, hence they are monotone. □

Finiteness

Lemma 65 (*$\langle \widehat{IntraState}, \sqsubseteq \rangle$ and $\langle \widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}, \sqsubseteq \rangle$ are finite lattices*)**.** *$\langle \widehat{IntraState}, \sqsubseteq \rangle$ and $\langle \widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}, \sqsubseteq \rangle$ are finite lattices*

Proof. Elements of $\widehat{IntraState}$ are tuples of components that are finite (see Lemma 26), and hence $\widehat{IntraState}$ is finite itself. Elements of $\widehat{\Pi}$ are functions of which the domain ($\widehat{PID}$) and the range ($\widehat{IntraState} \times \widehat{\Sigma}$) are finite, and hence $\widehat{\Pi} \times \widehat{Store} \times \widehat{KStore}$ is also finite. □

Termination

Theorem 19 (Termination)**.** *The computation of* $\mathrm{lfp}(\mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\tau})$ *always terminates.*

Proof. By Tarski's fixed-point theorem (Tarski, 1955) Lemmas 64 and 65., we know that the computation of $\mathrm{lfp}(\mathit{Intra}\text{-}\widehat{\mathcal{H}}^{\lambda_\tau}_{\hat{p},\hat{\varsigma}_0,\hat{\sigma}_0,\widehat{\Xi}_0,\hat{f}})$ terminates, and that the computation of $\mathrm{lfp}(\mathit{Inter}\text{-}\widehat{\mathcal{H}}_e^{\lambda_\tau})$ also terminates. □

BIBLIOGRAPHY

[1] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for java", *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 2, pp. 207–255, 2006. DOI: 10.1145/1119479.1119480.

[2] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction", in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds., ACM, 2014, pp. 373–384, ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535845.

[3] Y. Afek, G. Korland, and A. Zilberstein, "Lowering STM overhead with static analysis", in *Languages and Compilers for Parallel Computing - 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers*, K. D. Cooper, J. M. Mellor-Crummey, and V. Sarkar, Eds., ser. Lecture Notes in Computer Science, vol. 6548, Springer, 2010, pp. 31–45, ISBN: 978-3-642-19594-5. DOI: 10.1007/978-3-642-19595-2_3.

[4] G. Agha, *ACTORS - a model of concurrent computation in distributed systems*, ser. MIT Press series in artificial intelligence. MIT Press, 1986, ISBN: 978-0-262-01092-4.

[5] ——, "Concurrent object-oriented programming", *Commun. ACM*, vol. 33, no. 9, pp. 125–141, 1990. DOI: 10.1145/83880.84528.

[6] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation", *J. Funct. Program.*, vol. 7, no. 1, pp. 1–72, 1997.

[7] E. Albert, M. Gómez-Zamalloa, and M. Isabel, "Combining static analysis and testing for deadlock detection", in *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, E. Ábrahám and M. Huisman, Eds., ser. Lecture Notes in Computer Science, vol. 9681, Springer, 2016, pp. 409–424, ISBN: 978-3-319-33692-3. DOI: 10.1007/978-3-319-33693-0_26.

[8] E. S. Andreasen, A. Møller, and B. B. Nielsen, "Systematic approaches for increasing soundness and precision of static analyzers", in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ACM, 2017, pp. 31–36.

[9] J. Armstrong, R. Virding, and M. Williams, *Concurrent programming in ERLANG*. Prentice Hall, 1993, ISBN: 978-0-13-285792-5.

[10] C. Artho and A. Biere, "Applying static analysis to large-scale, multi-threaded java programs", in *13th Australian Software Engineering Conference (ASWEC 2001), 26-28 August 2001, Canberra, Australia*, IEEE Computer Society, 2001, pp. 68–75, ISBN: 0-7695-1254-2. DOI: 10.1109/ASWEC.2001.948499.

[11] T. Arts, M. Dam, L. Fredlund, and D. Gurov, "System description: Verification of distributed erlang programs", in *Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*, C. Kirchner and H. Kirchner, Eds., ser. Lecture Notes in Computer Science, vol. 1421, Springer, 1998, pp. 38–41, ISBN: 3-540-64675-2. DOI: 10.1007/BFb0054244.

[12] T. Arts and T. Noll, "Verifying generic erlang client-server implementations", in *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers*, M. Mohnen and P. W. M. Koopman, Eds., ser. Lecture Notes in Computer Science, vol. 2011, Springer, 2000, pp. 37–52, ISBN: 3-540-41919-5. DOI: 10.1007/3-540-45361-X_3.

[13] H. G. Baker and C. Hewitt, "The incremental garbage collection of processes", *SIGART Newsletter*, vol. 64, pp. 55–59, 1977. DOI: 10.1145/872736.806932.

[14] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Gros, A. Kamsky, S. McPeak, and D. R. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world", *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010. DOI: 10.1145/1646353.1646374.

[15] R. L. Bocchino Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A type and effect system for deterministic parallel java", in *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, S. Arora and G. T. Leavens, Eds., ACM, 2009, pp. 97–116, ISBN: 978-1-60558-766-0. DOI: 10.1145/1640089.1640097.

[16] C. Boyapati, R. Lee, and M. C. Rinard, "Ownership types for safe programming: Preventing data races and deadlocks", in *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002.*, M. Ibrahim and S. Matsuoka, Eds., ACM, 2002, pp. 211–230, ISBN: 1-58113-471-1. DOI: 10.1145/582419.582440.

[17] C. Boyapati and M. C. Rinard, "A parameterized type system for race-free java programs", in *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001.*, L. M. Northrop and J. M. Vlissides, Eds., ACM, 2001, pp. 56–69, ISBN: 1-58113-335-9. DOI: 10.1145/504282.504287.

[18] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction", in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Z. Shao and B. C. Pierce, Eds., ACM, 2009, pp. 289–300, ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480917.

[19] R. Carlsson, K. Sagonas, and J. Wilhelmsson, "Message analysis for concurrent languages", in *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, R. Cousot, Ed., ser. Lecture Notes in Computer Science, vol. 2694, Springer, 2003, pp. 73–90, ISBN: 3-540-40325-6. DOI: 10.1007/3-540-44898-5_5.

[20] D. Caromel, L. Henrio, and B. P. Serpette, "Asynchronous sequential processes", *Inf. Comput.*, vol. 207, no. 4, pp. 459–495, 2009. DOI: 10.1016/j.ic.2008.12.004.

[21] R. H. Carver and Y. Lei, "A general model for reachability testing of concurrent programs", in *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings*, J. Davies, W. Schulte, and M. Barnett, Eds., ser. Lecture Notes in Computer Science, vol. 3308, Springer, 2004, pp. 76–98, ISBN: 3-540-23841-7. DOI: 10.1007/978-3-540-30482-1_14.

[22] M. Christakis, A. Gotovos, and K. Sagonas, "Systematic testing for detecting concurrency errors in erlang programs", in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, IEEE Computer Society, 2013, pp. 154–163, ISBN: 978-1-4673-5961-0. DOI: 10.1109/ICST.2013.50.

[23] M. Christakis and K. Sagonas, "Static detection of deadlocks in erlang", Technical report, June, Tech. Rep., 2011.

[24] ——, "Static detection of race conditions in erlang", in *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, M. Carro and R. Peña, Eds., ser. Lecture Notes in Computer Science, vol. 5937, Springer, 2010, pp. 119–133, ISBN: 978-3-642-11502-8. DOI: 10.1007/978-3-642-11503-5_11.

[25] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil, "Deny capabilities for safe, fast actors", in *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, E. G. Boix, P. Haller, A. Ricci, and C. Varela, Eds., ACM, 2015, pp. 1–12, ISBN: 978-1-4503-3901-8. DOI: 10.1145/2824815.2824816.

[26] C. Colby, "Analyzing the communication topology of concurrent programs", in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, USA, June 21-23, 1995*, N. D. Jones, Ed., ACM Press, 1995, pp. 202–213, ISBN: 0-89791-720-0. DOI: 10.1145/215465.215592.

[27] E. C. Cooper and J. G. Morrisett, "Adding threads to standard ml", 1990.

[28] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng, "Bandera: Extracting finite-state models from java source code", in *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, C. Ghezzi, M. Jazayeri, and A. L. Wolf, Eds., ACM, 2000, pp. 439–448, ISBN: 1-58113-206-9. DOI: 10.1145/337180.337234.

[29] P. Cousot and R. Cousot, "Invariance proof methods and analysis techniques for parallel programs", in *Automatic Program Construction Techniques*, A. Biermann, G. Guiho, and Y. Kodratoff, Eds., Macmillan, New York, New York, United States, 1984, ch. 12, pp. 243–271.

[30] P. Cousot, "Constructive design of a hierarchy of semantics of a transition system by abstract interpretation", *Theor. Comput. Sci.*, vol. 277, no. 1-2, pp. 47–103, 2002. DOI: 10.1016/S0304-3975(00)00313-3.

[31] ——, "The verification grand challenge and abstract interpretation", in *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, B. Meyer and J. Woodcock, Eds., ser. Lecture Notes in Computer Science, vol. 4171, Springer, 2005, pp. 189–201, ISBN: 978-3-540-69147-1. DOI: 10.1007/978-3-540-69149-5_21.

[32] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints", in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds., ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[33] ——, "Modular static program analysis", in *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, R. N. Horspool, Ed., ser. Lecture Notes in Computer Science, vol. 2304, Springer, 2002, pp. 159–178, ISBN: 3-540-43369-4. DOI: 10.1007/3-540-45937-5_13.

[34] ——, "Semantic analysis of communicating sequential processes (shortened version)", in *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, J. W. de Bakker and J. van Leeuwen, Eds., ser. Lecture Notes in Computer Science, vol. 85, Springer, 1980, pp. 119–133, ISBN: 3-540-10003-2. DOI: 10.1007/3-540-10003-2_65.

[35] F. Dagnat and M. Pantel, "Static analysis of communications for erlang", in *Proceedings of 8th International Erlang/OTP User Conference*, 2002.

[36] M. Dam and L. Fredlund, "On the verification of open distributed systems", in *Proceedings of the 1998 ACM symposium on Applied Computing, SAC'98, Atlanta, GA, USA, February 27 - March 1, 1998*, K. M. George and G. B. Lamont, Eds., ACM, 1998, pp. 532–540, ISBN: 0-89791-969-6. DOI: 10.1145/330560.330917.

[37] J. De Bleser, Q. Stiévenart, J. Nicolay, and C. De Roover, "Static taint analysis of event-driven scheme programs", in *10th European Lisp Symposium, ELS 2017, April 3-4, 2017, Brussels, Belgium*, 2017, pp. 80–87.

[38] J. De Koster, T. Van Cutsem, and W. De Meuter, "43 years of actors: A taxonomy of actor models and their key properties", in *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016*, S. Clebsch, T. Desell, P. Haller, and A. Ricci, Eds., ACM, 2016, pp. 31–40, ISBN: 978-1-4503-4639-9. DOI: 10.1145/3001886.3001890.

[39] S. Demeyer, A. Parsai, G. Laghari, and B. van Bladel, Eds., *Proceedings of the 16th edition of the BElgian-NEtherlands software eVOLution symposium, Antwerp, Belgium, December 4-5, 2017*, vol. 2047, CEUR Workshop Proceedings, CEUR-WS.org, 2018.

[40] E. D'Osualdo, J. Kochems, and C. L. Ong, "Automatic verification of erlang-style concurrency", in *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, F. Logozzo and M. Fähndrich, Eds., ser. Lecture Notes in Computer Science, vol. 7935, Springer, 2013, pp. 454–476, ISBN: 978-3-642-38855-2. DOI: 10.1007/978-3-642-38856-9_24.

[41] E. D'Osualdo, J. Kochems, and L. Ong, "Soter: An automatic safety verifier for erlang", in *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA*, G. A. Agha, R. H. Bordini, A. Marron, and A. Ricci, Eds., ACM, 2012, pp. 137–140, ISBN: 978-1-4503-1630-9. DOI: 10.1145/2414639.2414658.

[42] E. Duesterwald and M. L. Soffa, "Concurrency analysis in the presence of procedures using a data-flow framework", in *Proceedings of the Symposium on Testing, Analysis, and Verification, TAV 1991, Victoria, British Columbia, Canada, October 8-10, 1991*, W. E. Howden, Ed., ACM, 1991, pp. 36–48, ISBN: 0-89791-449-X. DOI: 10.1145/120807.120811.

[43] D. R. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks", in *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, M. L. Scott and L. L. Peterson, Eds., ACM, 2003, pp. 237–252, ISBN: 1-58113-757-5. DOI: 10.1145/945445.945468.

[44] C. Flanagan and M. Abadi, "Types for safe locking", in *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, S. D. Swierstra, Ed., ser. Lecture Notes in Computer Science, vol. 1576, Springer, 1999, pp. 91–108, ISBN: 3-540-65699-5. DOI: 10.1007/3-540-49099-X_7.

[45] C. Flanagan and S. N. Freund, "Type-based race detection for java", in *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, M. S. Lam, Ed., ACM, 2000, pp. 219–232, ISBN: 1-58113-199-2. DOI: 10.1145/349299.349328.

[46] C. Flanagan, S. N. Freund, and S. Qadeer, "Thread-modular verification for shared-memory programs", in *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, D. L. Métayer, Ed., ser. Lecture Notes in Computer Science, vol. 2305, Springer, 2002, pp. 262–277, ISBN: 3-540-43363-5. DOI: 10.1007/3-540-45927-8_19.

[47] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia, "Modular verification of multithreaded programs", *Theor. Comput. Sci.*, vol. 338, no. 1-3, pp. 153–183, 2005. DOI: 10.1016/j.tcs.2004.12.006.

[48] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software", in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, J. Palsberg and M. Abadi, Eds., ACM, 2005, pp. 110–121, ISBN: 1-58113-830-X. DOI: 10.1145/1040305.1040315.

[49] C. Flanagan and S. Qadeer, "A type and effect system for atomicity", in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, R. Cytron and R. Gupta, Eds., ACM, 2003, pp. 338–349, ISBN: 1-58113-662-5. DOI: 10.1145/781131.781169.

[50] ——, "Predicate abstraction for software verification", in *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, J. Launchbury and J. C. Mitchell, Eds., ACM, 2002, pp. 191–202, ISBN: 1-58113-450-9. DOI: 10.1145/503272.503291.

[51] ——, "Thread-modular model checking", in *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, T. Ball and S. K. Rajamani, Eds., ser. Lecture Notes in Computer Science, vol. 2648, Springer, 2003, pp. 213–224, ISBN: 3-540-40117-2. DOI: 10.1007/3-540-44829-2_14.

[52] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations", in *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, R. Cartwright, Ed., ACM, 1993, pp. 237–247, ISBN: 0-89791-598-4. DOI: 10.1145/155090.155113.

[53] A. Flores-Montoya, E. Albert, and S. Genaim, "May-happen-in-parallel based deadlock analysis for concurrent objects", in *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques,*

DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings, D. Beyer and M. Boreale, Eds., ser. Lecture Notes in Computer Science, vol. 7892, Springer, 2013, pp. 273–288, ISBN: 978-3-642-38591-9. DOI: 10.1007/978-3-642-38592-6_19.

[54] P. Fonseca, C. Li, and R. Rodrigues, "Finding complex concurrency bugs in large multi-threaded applications", in *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*, C. M. Kirsch and G. Heiser, Eds., ACM, 2011, pp. 215–228, ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966465.

[55] L. Fredlund and H. Svensson, "Mcerlang: A model checker for a distributed functional programming language", in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, R. Hinze and N. Ramsey, Eds., ACM, 2007, pp. 125–136, ISBN: 978-1-59593-815-2. DOI: 10.1145/1291151.1291171.

[56] P. Garoche, "Static analysis of an actor-based process calculus by abstract interpretation. (analyse statique d'un calcul d'acteurs par interprétation abstraite)", PhD thesis, National Polytechnic Institute of Toulouse, France, 2008.

[57] P. Garoche, M. Pantel, and X. Thirioux, "Static safety for an actor dedicated process calculus by abstract interpretation", in *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, R. Gorrieri and H. Wehrheim, Eds., ser. Lecture Notes in Computer Science, vol. 4037, Springer, 2006, pp. 78–92, ISBN: 3-540-34893-X. DOI: 10.1007/11768869_8.

[58] T. Gilray, M. D. Adams, and M. Might, "Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis", in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, J. Garrigue, G. Keller, and E. Sumii, Eds., ACM, 2016, pp. 407–420, ISBN: 978-1-4503-4219-3. DOI: 10.1145/2951913.2951936.

[59] T. Gilray, S. Lyde, M. D. Adams, M. Might, and D. Van Horn, "Pushdown control-flow analysis for free", in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodk and R. Majumdar, Eds., ACM, 2016, pp. 691–704, ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837631.

[60] P. Godefroid, "Model checking for programming languages using verisoft", in *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, P. Lee, F. Henglein, and N. D. Jones, Eds., ACM Press, 1997, pp. 174–186, ISBN: 0-89791-853-3. DOI: 10.1145/263699.263717.

[61] ——, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. Lecture Notes in Computer Science. Springer, 1996, vol. 1032, ISBN: 3-540-60761-7. DOI: 10.1007/3-540-60761-7.

[62] P. Godefroid and N. Nagappan, "Concurrency at microsoft: An exploratory survey", in *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[63] P. Godefroid and P. Wolper, "A partial approach to model checking", in *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, IEEE Computer Society, 1991, pp. 406–415, ISBN: 0-8186-2230-X. DOI: 10.1109/LICS.1991.151664.

[64] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv, "Thread-modular shape analysis", in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, J. Ferrante and K. S. McKinley, Eds., ACM, 2007, pp. 266–277, ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250765.

[65] A. Gupta, C. Popeea, and A. Rybalchenko, "Predicate abstraction and refinement for verifying multi-threaded programs", in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, T. Ball and M. Sagiv, Eds., ACM, 2011, pp. 331–344, ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926424.

[66] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming", *Theor. Comput. Sci.*, vol. 410, no. 2-3, pp. 202–220, 2009. DOI: 10.1016/j.tcs.2008.09.019.

[67] P. Haller and F. Sommers, *Actors in Scala*. Artima Incorporation, 2012.

[68] K. Havelund, "Java pathfinder, A translator from java to promela", in *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings*, D. Dams, R. Gerth, S. Leue, and M. Massink, Eds., ser. Lecture Notes in Computer Science, vol. 1680, Springer, 1999, p. 152, ISBN: 3-540-66499-8. DOI: 10.1007/3-540-48234-2_11.

[69] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA pathfinder", *STTT*, vol. 2, no. 4, pp. 366–381, 2000. DOI: 10.1007/s100090050043.

[70] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer, "Thread-modular abstraction refinement", in *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, W. A. H. Jr. and F. Somenzi, Eds., ser. Lecture Notes in Computer Science, vol. 2725, Springer, 2003, pp. 262–274, ISBN: 3-540-40524-0. DOI: 10.1007/978-3-540-45069-6_27.

[71] C. Hewitt, P. B. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence", in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, N. J. Nilsson, Ed., William Kaufmann, 1973, pp. 235–245.

[72] C. Hewitt and B. Smith, "A plasma primer", *Draft. Cambridge, Massachusetts: MIT Artificial Intelligence Laboratory*, 1975.

[73] R. Hickey, "The clojure programming language", in *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus*, J. Brichau, Ed., ACM, 2008, p. 1, ISBN: 978-1-60558-270-2. DOI: 10.1145/1408681.1408682.

[74] C. A. R. Hoare, "Communicating sequential processes", *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978. DOI: 10.1145/359576.359585.

[75] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004, ISBN: 978-0-321-22862-8.

[76] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types", *J. ACM*, vol. 63, no. 1, 9:1–9:67, 2016. DOI: 10.1145/2827695.

[77] ——, "Multiparty asynchronous session types", in *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, G. C. Necula and P. Wadler, Eds., ACM, 2008, pp. 273–284, ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328472.

[78] S. Hong and M. Kim, "A survey of race bug detection techniques for multithreaded programmes", *Softw. Test., Verif. Reliab.*, vol. 25, no. 3, pp. 191–217, 2015. DOI: 10.1002/stvr.1564.

[79] D. Hovemeyer and W. Pugh, "Finding bugs is easy", *SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004. DOI: 10.1145/1052883.1052895.

[80] F. Huch, "Verification of erlang programs using abstract interpretation and model mhecking", in *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.*, D. Rémi and P. Lee, Eds., ACM, 1999, pp. 261–272, ISBN: 1-58113-111-9. DOI: 10.1145/317636.317908.

[81] S. M. Imam and V. Sarkar, "Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries", in *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*, E. G. Boix, P. Haller, A. Ricci, and C. Varela, Eds., ACM, 2014, pp. 67–80, ISBN: 978-1-4503-2189-1. DOI: 10.1145/2687357.2687368.

[82] S. Jagannathan, "Locality abstractions for parallel and distributed computing", in *Theory and Practice of Parallel Programming, International Workshop TPPP'94, Sendai, Japan, November 7-9, 1994, Proceedings*, T. Ito and A. Yonezawa, Eds., ser. Lecture Notes in Computer Science, vol. 907, Springer, 1994, pp. 320–345, ISBN: 3-540-59172-9. DOI: 10.1007/BFb0026577.

[83] S. Jagannathan and S. Weeks, "Analyzing stores and references in a parallel symbolic language", in *LISP and Functional Programming*, 1994, pp. 294–305. DOI: 10.1145/182409.182493.

[84] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. T. Vechev, "Stateless model checking of event-driven applications", in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, J. Aldrich and P. Eugster, Eds., ACM, 2015, pp. 57–73, ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814282.

[85] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?", in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds., IEEE Computer Society, 2013, pp. 672–681, ISBN: 978-1-4673-3076-3. DOI: 10.1109/ICSE.2013.6606613.

[86] C. B. Jones, "Specification and design of (parallel) programs", in *IFIP Congress*, 1983, pp. 321–332.

[87] S. L. P. Jones, A. D. Gordon, and S. Finne, "Concurrent haskell", in *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, H. Boehm and G. L. S. Jr., Eds., ACM Press, 1996, pp. 295–308, ISBN: 0-89791-769-3. DOI: 10.1145/237721.237794.

[88] V. Kahlon, F. Ivancic, and A. Gupta, "Reasoning about threads communicating via locks", in *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, K. Etessami and S. K. Rajamani, Eds., ser. Lecture Notes in Computer Science, vol. 3576, Springer, 2005, pp. 505–518, ISBN: 3-540-27231-3. DOI: 10.1007/11513988_49.

[89] H. Kastenberg and A. Rensink, "Dynamic partial order reduction using probe sets", in *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, F. van Breugel and M. Chechik, Eds., ser. Lecture Notes in Computer Science, vol. 5201, Springer, 2008, pp. 233–247, ISBN: 978-3-540-85360-2. DOI: 10.1007/978-3-540-85361-9_21.

[90] S. Khurshid, C. S. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing", in *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, H. Garavel and J. Hatcliff, Eds., ser. Lecture Notes in Computer Science, vol. 2619, Springer, 2003, pp. 553–568, ISBN: 3-540-00898-5. DOI: 10.1007/3-540-36577-X_40.

[91] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, "Effective stateless model checking for C/C++ concurrency", *PACMPL*, vol. 2, no. POPL, 17:1–17:32, 2018. DOI: 10.1145/3158105.

[92] R. Kuhn, B. Hanafee, and J. Allen, *Reactive design patterns*. Manning Publications Company, 2017.

[93] P. B. Ladkin and B. Simons, "Compile-time analysis of communicating processes", in *Proceedings of the 6th international conference on Supercomputing, ICS 1992, Washington, DC, USA, July 19-24, 1992*, K. Kennedy and C. D. Polychronopoulos, Eds., ACM, 1992, pp. 248–259, ISBN: 0-89791-485-6. DOI: 10.1145/143369.143417.

[94] S. Lauterburg, M. Dotta, D. Marinov, and G. A. Agha, "A framework for state-space exploration of java-based actor programs", in *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, IEEE Computer Society, 2009, pp. 468–479, ISBN: 978-0-7695-3891-4. DOI: 10.1109/ASE.2009.88.

[95] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha, "Evaluating ordering heuristics for dynamic partial-order reduction techniques", in *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, D. S. Rosenblum and G. Taentzer, Eds., ser. Lecture Notes in Computer Science, vol. 6013, Springer, 2010, pp. 308–322, ISBN: 978-3-642-12028-2. DOI: 10.1007/978-3-642-12029-9_22.

[96] D. Lea, *Concurrent programming in Java - design principles and patterns*, ser. Java series. Addison-Wesley-Longman, 1997, ISBN: 978-0-201-69581-6.

[97] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon, "The objective caml system release 3.11", *Documentation and users manual. INRIA*, 2008.

[98] A. Lindgren, "A prototype of a soft type system for erlang", Master's thesis, Uppsala University, 1996.

[99] S. Marlow and P. Wadler, "A practical subtyping system for erlang", in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997.*, S. L. P. Jones, M. Tofte, and A. M. Berman, Eds., ACM, 1997, pp. 136–149, ISBN: 0-89791-918-1. DOI: 10.1145/258948.258962.

[100] M. Martel and M. Gengler, "Communication topology analysis for concurrent programs", in *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, K. Havelund, J. Penix, and W. Visser, Eds., ser. Lecture Notes in Computer Science, vol. 1885, Springer, 2000, pp. 265–286, ISBN: 3-540-41030-9. DOI: 10.1007/10722468_16.

[101] S. McConnell, *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004, ISBN: 9780735619678.

[102] N. Mercouroff, "An algorithm for analyzing communicating processes", in *Mathematical Foundations of Programming Semantics, 7th International Conference, Pittsburgh, PA, USA, March 25-28, 1991, Proceedings*, S. D. Brookes, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, Eds., ser. Lecture Notes in Computer

Science, vol. 598, Springer, 1991, pp. 312–325, ISBN: 3-540-55511-0. DOI: 10.1007/3-540-55511-0_16.

[103] J. Midtgaard, F. Nielson, and H. R. Nielson, "A parametric abstract domain for lattice-valued regular expressions", in *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, X. Rival, Ed., ser. Lecture Notes in Computer Science, vol. 9837, Springer, 2016, pp. 338–360, ISBN: 978-3-662-53412-0. DOI: 10.1007/978-3-662-53413-7_17.

[104] ——, "Iterated process analysis over lattice-valued regular expressions", in *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, J. Cheney and G. Vidal, Eds., ACM, 2016, pp. 132–145, ISBN: 978-1-4503-4148-6. DOI: 10.1145/2967973.2968601.

[105] M. Might and P. Manolios, "A posteriori soundness for non-deterministic abstract interpretations", in *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, N. D. Jones and M. Müller-Olm, Eds., ser. Lecture Notes in Computer Science, vol. 5403, Springer, 2009, pp. 260–274, ISBN: 978-3-540-93899-6. DOI: 10.1007/978-3-540-93900-9_22.

[106] M. Might and O. Shivers, "Improving flow analyses via gammacfa: Abstract garbage collection and counting", in *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, J. H. Reppy and J. L. Lawall, Eds., ACM, 2006, pp. 13–25, ISBN: 1-59593-309-3. DOI: 10.1145/1159803.1159807.

[107] M. Might and D. Van Horn, "A family of abstract interpretations for static analysis of concurrent higher-order programs", in *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, E. Yahav, Ed., ser. Lecture Notes in Computer Science, vol. 6887, Springer, 2011, pp. 180–197, ISBN: 978-3-642-23701-0. DOI: 10.1007/978-3-642-23702-7_16.

[108] M. S. Miller, E. D. Tribble, and J. S. Shapiro, "Concurrency among strangers", in *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*, R. D. Nicola and D. Sangiorgi, Eds., ser. Lecture Notes in Computer Science, vol. 3705, Springer, 2005, pp. 195–229, ISBN: 3-540-30007-4. DOI: 10.1007/11580850_12.

[109] M. S. Miller, K.-P. Yee, J. Shapiro, *et al.*, "Capability myths demolished", Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals, Tech. Rep., 2003.

[110] A. Miné, "Relational thread-modular static value analysis by abstract interpretation", in *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, K. L. McMillan and X. Rival, Eds., ser. Lecture Notes in Computer Science, vol. 8318,

Springer, 2014, pp. 39–58, ISBN: 978-3-642-54012-7. DOI: 10.1007/978-3-642-54013-4_3.

[111] ——, "Static analysis of run-time errors in embedded real-time parallel C programs", *Logical Methods in Computer Science*, vol. 8, no. 1, 2012. DOI: 10.2168/LMCS-8(1:26)2012.

[112] A. Miné and D. Delmas, "Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software", in *2015 International Conference on Embedded Software, EMSOFT 2015, Amsterdam, Netherlands, October 4-9, 2015*, A. Girault and N. Guan, Eds., IEEE, 2015, pp. 65–74, ISBN: 978-1-4673-8079-9. DOI: 10.1109/EMSOFT.2015.7318261.

[113] R. Monat and A. Miné, "Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions", in *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, A. Bouajjani and D. Monniaux, Eds., ser. Lecture Notes in Computer Science, vol. 10145, Springer, 2017, pp. 386–404, ISBN: 978-3-319-52233-3. DOI: 10.1007/978-3-319-52234-0_21.

[114] D. Mostrous and V. T. Vasconcelos, "Session typing for a featherweight erlang", in *Coordination Models and Languages - 13th International Conference, COORDINATION 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, W. De Meuter and G. Roman, Eds., ser. Lecture Notes in Computer Science, vol. 6721, Springer, 2011, pp. 95–109, ISBN: 978-3-642-21463-9. DOI: 10.1007/978-3-642-21464-6_7.

[115] P. Müller, Ed., *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, vol. 74, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, ISBN: 978-3-95977-035-4.

[116] M. Naik, C. Park, K. Sen, and D. Gay, "Effective static deadlock detection", in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, IEEE, 2009, pp. 386–396, ISBN: 978-1-4244-3452-7. DOI: 10.1109/ICSE.2009.5070538.

[117] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco, "Communicating state transition systems for fine-grained concurrent resources", in *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, Z. Shao, Ed., ser. Lecture Notes in Computer Science, vol. 8410, Springer, 2014, pp. 290–310, ISBN: 978-3-642-54832-1. DOI: 10.1007/978-3-642-54833-8_16.

[118] M. Nash and W. Waldron, *Applied Akka Patterns: A Hands-on Guide to Designing Distributed Applications*. O'Reilly Media, Inc., 2016.

[119] R. Neykova and N. Yoshida, "Multiparty session actors", in *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, E. Kühn and R. Pugliese, Eds., ser. Lecture Notes in Computer Science, vol. 8459, Springer, 2014, pp. 131–146, ISBN: 978-3-662-43375-1. DOI: 10.1007/978-3-662-43376-8_9.

[120] N. Ng and N. Yoshida, "Static deadlock detection for concurrent go by global session graph synthesis", in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, A. Zaks and M. V. Hermenegildo, Eds., ACM, 2016, pp. 174–184, ISBN: 978-1-4503-4241-4. DOI: 10.1145/2892208.2892232.

[121] J. Nicolay, Q. Stiévenart, W. De Meuter, and C. De Roover, "Purity analysis for javascript through abstract interpretation", *Journal of Software: Evolution and Process*, e1889–n/a, 2017, e1889 smr.1889, ISSN: 2047-7481. DOI: 10.1002/smr.1889.

[122] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999, ISBN: 978-3-540-65410-0. DOI: 10.1007/978-3-662-03811-6.

[123] P. W. O'Hearn, "Resources, concurrency, and local reasoning", *Theor. Comput. Sci.*, vol. 375, no. 1-3, pp. 271–307, 2007. DOI: 10.1016/j.tcs.2006.12.035. [Online]. Available: https://doi.org/10.1016/j.tcs.2006.12.035.

[124] B. C. Pierce, *Types and programming languages*. MIT Press, 2002, ISBN: 978-0-262-16209-8.

[125] K. Sagonas, "Detecting defects in erlang programs using static analysis", in *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 14-16, 2007, Wroclaw, Poland*, M. Leuschel and A. Podelski, Eds., ACM, 2007, p. 37, ISBN: 978-1-59593-769-8. DOI: 10.1145/1273920.1273926.

[126] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller, "Automated type-based analysis of data races and atomicity", in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, K. Pingali, K. A. Yelick, and A. S. Grimshaw, Eds., ACM, 2005, pp. 83–94, ISBN: 1-59593-080-9. DOI: 10.1145/1065944.1065956.

[127] A. Scalas, O. Dardha, R. Hu, and N. Yoshida, "A linear decomposition of multiparty sessions for safe distributed programming", in *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, P. Müller, Ed., ser. LIPIcs, vol. 74, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 24:1–24:31, ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.24.

[128] K. Sen, "Concolic testing", in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds., ACM, 2007, pp. 571–572, ISBN: 978-1-59593-882-4. DOI: 10.1145/1321631.1321746.

[129] K. Sen and G. Agha, "A race-detection and flipping algorithm for automated testing of multi-threaded programs", in *Hardware and Software, Verification and Testing, Second International Haifa Verification Conference, HVC 2006, Haifa, Israel, October 23-26, 2006. Revised Selected Papers*, E. Bin, A. Ziv, and S. Ur, Eds., ser. Lecture Notes in Computer Science, vol. 4383, Springer, 2006, pp. 166–182, ISBN: 978-3-540-70888-9. DOI: 10.1007/978-3-540-70889-6_13.

[130] ——, "Automated systematic testing of open distributed programs", in *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, L. Baresi and R. Heckel, Eds., ser. Lecture Notes in Computer Science, vol. 3922, Springer, 2006, pp. 339–356, ISBN: 3-540-33093-3. DOI: 10.1007/11693017_25.

[131] ——, "CUTE and jcute: Concolic unit testing and explicit path model-checking tools", in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, T. Ball and R. B. Jones, Eds., ser. Lecture Notes in Computer Science, vol. 4144, Springer, 2006, pp. 419–423, ISBN: 3-540-37406-X. DOI: 10.1007/11817963_38.

[132] K. Sen and G. A. Agha, "Concolic testing of multithreaded programs and its application to testing security protocols", Tech. Rep., 2006.

[133] I. Sergey, A. Nanevski, and A. Banerjee, "Mechanized verification of fine-grained concurrent programs", in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, D. Grove and S. Blackburn, Eds., ACM, 2015, pp. 77–87, ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737964.

[134] N. Shavit and D. Touitou, "Software transactional memory", *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997. DOI: 10.1007/s004460050028.

[135] O. Shivers, "Control-flow analysis of higher-order languages", PhD thesis, Carnegie-Mellon University, 1991.

[136] M. Sirjani, F. S. de Boer, and A. Movaghar-Rahimabadi, "Modular verification of a component-based actor language", *J. UCS*, vol. 11, no. 10, pp. 1695–1717, 2005. DOI: 10.3217/jucs-011-10-1695.

[137] M. Sirjani and M. M. Jaghoori, "Ten years of analyzing actors: Rebeca experience", in *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, G. Agha, O. Danvy, and J. Meseguer, Eds., ser. Lecture Notes in Computer Science, vol. 7000, Springer, 2011, pp. 20–56, ISBN: 978-3-642-24932-7. DOI: 10.1007/978-3-642-24933-4_3.

[138] S. Srinivasan and A. Mycroft, "Kilim: Isolation-typed actors for java", in *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, J. Vitek, Ed., ser. Lecture Notes in Computer Science, vol. 5142, Springer, 2008, pp. 104–128, ISBN: 978-3-540-70591-8. DOI: 10.1007/978-3-540-70592-5_6.

[139] K. Stadtmüller, M. Sulzmann, and P. Thiemann, "Static trace-based deadlock analysis for synchronous mini-go", in *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, A. Igarashi, Ed., ser. Lecture Notes in Computer Science, vol. 10017, 2016, pp. 116–136, ISBN: 978-3-319-47957-6. DOI: 10.1007/978-3-319-47958-3_7.

[140] Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover, "A general method for rendering analyses for diverse concurrency models modular", *Submitted on Feb. 22, 2018 to Journal of Systems and Software*,

[141] ——, "Building a modular static analysis framework in scala (tool paper)", in *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, A. Biboudis, M. Jonnalagedda, S. Stucki, and V. Ureche, Eds., ACM, 2016, pp. 105–109, ISBN: 978-1-4503-4648-1. DOI: 10.1145/2998392.3001579.

[142] ——, "Detecting concurrency bugs in higher-order programs through abstract interpretation", in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, M. Falaschi and E. Albert, Eds., ACM, 2015, pp. 232–243, ISBN: 978-1-4503-3516-4. DOI: 10.1145/2790449.2790530.

[143] ——, "Mailbox abstractions for static analysis of actor programs", in *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, P. Müller, Ed., ser. LIPIcs, vol. 74, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 25:1–25:30, ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.25.

[144] ——, "Poster: Static analysis of concurrent higher-order programs", in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds., IEEE Computer Society, 2015, pp. 821–822. DOI: 10.1109/ICSE.2015.265.

[145] ——, "Smap: Scalable modular static analysis of actor programs", *Submitted on Oct. 17, 2017 to Information & Software Technology*,

[146] Q. Stiévenart, M. Vandercammen, W. De Meuter, and C. De Roover, "Scala-am: A modular static analysis framework", in *16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, October 2-3, 2016*, IEEE Computer Society, 2016, pp. 85–90, ISBN: 978-1-5090-3848-0. DOI: 10.1109/SCAM.2016.14.

[147] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications", *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955.

[148] S. Tasharofi, P. Dinges, and R. E. Johnson, "Why do scala developers mix the actor model with other concurrency models?", in *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, G. Castagna, Ed., ser. Lecture Notes in Computer Science, vol. 7920, Springer, 2013, pp. 302–326, ISBN: 978-3-642-39037-1. DOI: 10.1007/978-3-642-39038-8_13.

[149] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha, "Transdpor: A novel dynamic partial-order reduction technique for testing actor programs", in *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, H. Giese and G. Rosu, Eds., ser. Lecture Notes in Computer Science, vol. 7273, Springer, 2012, pp. 219–234, ISBN: 978-3-642-30792-8. DOI: 10.1007/978-3-642-30793-5_14.

[150] M. Ujma and N. Shafiei, "Jpf-concurrent: An extension of java pathfinder for java.util.concurrent", *CoRR*, vol. abs/1205.0042, 2012. arXiv: 1205.0042.

[151] A. Valmari, "The state explosion problem", in *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, W. Reisig and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 1491, Springer, 1996, pp. 429–528, ISBN: 3-540-65306-6. DOI: 10.1007/3-540-65306-6_21.

[152] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter, "Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks", in *XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), 8-9 November 2007, Iquique, Chile*, IEEE Computer Society, 2007, pp. 3–12, ISBN: 978-0-7695-3017-8. DOI: 10.1109/SCCC.2007.4.

[153] N. Van Es, J. Nicolay, Q. Stiévenart, T. D'Hondt, and C. De Roover, "A performant scheme interpreter in asm.js", in *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, S. Ossowski, Ed., ACM, 2016, pp. 1944–1951, ISBN: 978-1-4503-3739-7. DOI: 10.1145/2851613.2851748.

[154] N. Van Es, Q. Stiévenart, J. Nicolay, T. D'Hondt, and C. De Roover, "Implementing a performant scheme interpreter for the web in asm.js", *Computer Languages, Systems & Structures*, vol. 49, pp. 62–81, 2017. DOI: 10.1016/j.cl.2017.02.002.

[155] N. Van Es, M. Vandercammen, and C. De Roover, "Incrementalizing abstract interpretation", in *Proceedings of the 16th edition of the BElgian-NEtherlands software eVOLution symposium, Antwerp, Belgium, December 4-5, 2017.*, S. Demeyer, A. Parsai, G. Laghari, and B. van Bladel, Eds., ser. CEUR Workshop Proceedings, vol. 2047, CEUR-WS.org, 2017, pp. 31–35.

[156] D. Van Horn and M. Might, "Abstracting abstract machines", in *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, P. Hudak and S. Weirich, Eds., ACM, 2010, pp. 51–62, ISBN: 978-1-60558-794-3. DOI: 10.1145/1863543.1863553.

[157] M. Vandercammen and C. De Roover, "Employing run-time static analysis to improve concolic execution", in *Proceedings of the 16th edition of the BElgian-NEtherlands software eVOLution symposium, Antwerp, Belgium, December 4-5, 2017.*, S. Demeyer, A. Parsai, G. Laghari, and B. van Bladel, Eds., ser. CEUR Workshop Proceedings, vol. 2047, CEUR-WS.org, 2017, pp. 26–29.

[158] ——, "Improving trace-based JIT optimisation using whole-program information", in *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages, VMIL@SPLASH 2016, Amsterdam, Netherlands, October 31, 2016*, A. L. Hosking and W. Srisa-an, Eds., ACM, 2016, pp. 16–23, ISBN: 978-1-4503-4645-0. DOI: 10.1145/2998415.2998418.

[159] M. Vandercammen, Q. Stiévenart, W. De Meuter, and C. De Roover, "STRAF: A scala framework for experiments in trace-based JIT compilation", in *Grand Timely Topics in Software Engineering - International Summer School GTTSE 2015, Braga, Portugal, August 23-29, 2015, Tutorial Lectures*, J. Cunha, J. P. Fernandes, R. Lämmel, J. Saraiva, and V. Zaytsev, Eds., ser. Lecture Notes in Computer Science, vol. 10223, Springer, 2015, pp. 223–234. DOI: 10.1007/978-3-319-60074-1_10.

[160] C. A. Varela and G. Agha, "Programming dynamically reconfigurable open systems with SALSA", *SIGPLAN Notices*, vol. 36, no. 12, pp. 20–34, 2001. DOI: 10.1145/583960.583964.

[161] S. Weeks, S. Jagannathan, and J. Philbin, "A concurrent abstract interpreter", *Lisp and Symbolic Computation*, vol. 7, no. 2-3, pp. 173–193, 1994.

[162] A. L. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for java libraries", in *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, A. P. Black, Ed., ser. Lecture Notes in Computer Science, vol. 3586, Springer, 2005, pp. 602–629, ISBN: 3-540-27992-X. DOI: 10.1007/11531142_26.

[163] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Efficient stateful dynamic partial order reduction", in *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings*, K. Havelund, R. Majumdar, and J. Palsberg, Eds., ser. Lecture Notes in Computer Science, vol. 5156, Springer, 2008, pp. 288–305, ISBN: 978-3-540-85113-4. DOI: 10.1007/978-3-540-85114-1_20.

[164] X. Yi, J. Wang, and X. Yang, "Stateful dynamic partial-order reduction", in *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, Z. Liu and J. He, Eds., ser. Lecture Notes in Computer Science, vol. 4260, Springer, 2006, pp. 149–167, ISBN: 3-540-47460-9. DOI: 10.1007/11901433_9.

[165] A. Yonezawa, J. Briot, and E. Shibayama, "Object-oriented concurrent programming in ABCL/1", in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings.*, N. K. Mey-

rowitz, Ed., ACM, 1986, pp. 258–268, ISBN: 0-89791-204-7. DOI: 10.1145/28697.28722.