Automated and Foundational Verification of Low-Level Programs

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

vorgelegt von
Michael Sammler

Saarbrücken, 2023

Tag des Kolloquiums
4. Dezember 2023

Dekan der Fakultät für Mathematik und Informatik
Prof. Dr. Jürgen Steimle

Prüfungsausschuss

Vorsitzender:	Prof. Dr. Holger Hermanns
Gutachter:	Prof. Dr. Derek Dreyer
	Prof. Dr. Deepak Garg
	Prof. Dr. Ranjit Jhala
	Dr. Viktor Vafeiadis
Akademischer Mitarbeiter:	Dr. Aïna Linn Georges

Abstract

Formal verification is a promising technique to ensure the reliability of low-level programs like operating systems and hypervisors, since it can show the absence of whole classes of bugs and prevent critical vulnerabilities. However, to realize the full potential of formal verification for real-world low-level programs one has to overcome several challenges, including: (1) dealing with the complexities of *realistic* models of real-world programming languages; (2) ensuring the *trustworthiness* of the verification, ideally by providing foundational proofs (*i.e.*, proofs that can be checked by a general-purpose proof assistant); and (3) minimizing the manual effort required for verification by providing a high degree of *automation*.

This dissertation presents multiple projects that advance formal verification along these three axes: *RefinedC* provides the first approach for verifying C code that combines foundational proofs with a high degree of automation via a novel refinement and ownership type system. *Islaris* shows how to scale verification of assembly code to realistic models of modern instruction set architectures—in particular, Armv8-A and RISC-V. *DimSum* develops a decentralized approach for reasoning about programs that consist of components written in multiple different languages (*e.g.*, assembly and C), as is common for low-level programs. RefinedC and Islaris rest on *Lithium*, a novel proof engine for separation logic that combines automation with foundational proofs.

Zusammenfassung

Formale Verifikation ist eine vielversprechende Technik, um die Verlässlichkeit von grundlegenden Programmen wie Betriebssystemen sicherzustellen. Um das volle Potenzial formaler Verifikation zu realisieren, müssen jedoch mehrere Herausforderungen gemeistert werden: Erstens muss die Komplexität von realistischen Modellen von Programmiersprachen wie C oder Assembler gehandhabt werden. Zweitens muss die Vertrauenswürdigkeit der Verifikation sichergestellt werden, idealerweise durch maschinenüberprüfbare Beweise. Drittens muss die Verifikation automatisiert werden, um den manuellen Aufwand zu minimieren.

Diese Dissertation präsentiert mehrere Projekte, die formale Verifikation entlang dieser Achsen weiterentwickeln: *RefinedC* ist der erste Ansatz für die Verifikation von C Code, der maschinenüberprüfbare Beweise mit einem hohen Grad an Automatisierung vereint. *Islaris* zeigt, wie die Verifikation von Assembler zu realistischen Modellen von modernen Befehlssatzarchitekturen wie Armv8-A oder RISC-V skaliert werden kann. *DimSum* entwickelt einen neuen Ansatz für die Verifizierung von Programmen, die aus Komponenten in mehreren Programmiersprachen bestehen (z.B., C und Assembler), wie es oft bei grundlegenden Programmen wie Betriebssystemen der Fall ist. RefinedC und Islaris basieren auf *Lithium*, eine neue Automatisierungstechnik für Separationslogik, die maschinenüberprüfbare Beweise und Automatisierung verbindet.

Acknowledgements

The past years in which I worked towards this dissertation have been an incredibly rewarding journey and I had the joy to meet and work with many great people—this dissertation would not have been possible without their support.

First and foremost I would like to thank my advisors Derek Dreyer and Deepak Garg, who helped me grow at every step of the way, from not knowing much about research to writing this dissertation. Out of many things, I, in particular, would like to thank Derek for teaching me that the best technical results are not worth much without a clear and understandable presentation—even though I might not have felt that way when rewriting an introduction for the fourth time—and I would like to thank Deepak for sharing his broad knowledge and encouraging me to look at the broader picture—his ability to point out deep connections to other work after some technical presentation always impressed me. I am also thankful for their patience whenever I excitedly shared the sketches of some new idea and for guiding me towards turning these ideas into clear research directions.

Next, I thank my examiners, Ranjit Jhala and Viktor Vafeiadis, for reviewing my dissertation, and Holger Hermanns and Aïna Linn Georges for serving as chair and academic member of my committee, respectively. I also thank Simon Spies, Rose Hoberman, Thibault Dardinier, Mete Polat, Ike Mulder, Kimaya Bedarkar, Laila Elbeheiry, and Sacha-Élie Ayoun for reading and giving feedback on this dissertation. Additionally, I would like to thank the MPI-SWS office and IT staff for always being available to help when I encountered organizational or IT-related problems. I also thank Tadeusz Litak for the SemProg course at FAU and the subsequent supervision that got me started on the path towards this dissertation by sparking my interest in the Coq proof assistant.

The work described in this dissertation would not have been possible without the work of my amazing collaborators Rodolphe Lepigre, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Peter Sewell, Angus Hammond, Kayvan Memarian, Brian Campbell, and Jean Pichon-Pharabod. Additionally, I would like to thank Peter Sewell for initiating and leading the pKVM verification project that motivated much of the work in this dissertation.

I also would like to express my gratitude to the students that welcomed me at MPI—I cherished our common lunches or extended chats over tea. In particular, to Ralf Jung, for his work on Rust and RustBelt, which is the reason I got me interested in the MPI and doing a PhD in the first

place, and for his mentorship when I started my PhD; to Rodolphe Lepigre, for his OCaml hacking skills that made many of the projects described in this dissertation possible; to Jan-Oliver Kaiser, for sharing his vast knowledge of Coq with me; to Hai Dang, for showing me what complicated proofs really look like; to Anjo Vahldiek-Oberwagner, for introducing me to systems research; and to David Swasey, for providing the work that my first project was built on. Additionally, I would like to thank Simon Spies for supporting me with his writing and presentation skills on many occasions and making travel to conferences much more enjoyable (*e.g.*, by preventing me from being arrested by CBP), and all the other students, postdocs, and interns, including Lennard Gäher, Youngju Song, Emanuele D'Osualdo, Jan Menz, Kimaya Bedarkar, Laila Elbeheiry, Andrew Hirsch, Hei Li, Paul Zhu, George Pîrlea, Ignacio Tiraboschi, Johannes Hostert, Niklas Mück, Aïna Linn Georges, Benjamin Peters, Janine Lohse, Neven Villani, Vincent Lafeychine, and all others who made the MPI a fun place—be it through joint work on interesting projects, discussions at tea times, or explorations of the culinary offerings of Saarbrücken.

Last, but surely not least, I thank my friends and family for their unwavering support, for making sure that we stay connected across the distance and a pandemic, for all the fun visits of Saarland (with and without rain), for the board game afternoons, and for the joint travel throughout Europe. My biggest thanks goes to my partner Kristin—for exploring the Saarland and the world with me, for her support when I dive into work, while making sure that I resurface, and for so much more. I am looking forward to the many exciting adventures that await us in the future. I dedicate this dissertation to my mother Ruth, who gave me my love for books and learning.

Michael Sammler
Saarbrücken, December 2023

Contents

Chapter 1

Introduction

Low-level systems software like operating systems and hypervisors forms the foundation of modern computer systems. Such systems software often provides critical components that ensure the reliability and security of the overall system. On the flip side, this means that bugs and vulnerabilities in such low-level systems software can lead to catastrophic failures and security problems. As a consequence, detecting and preventing bugs in low-level systems software is crucial for providing a reliable basis that the modern computer-based world can build upon.

So how can one ensure reliability of systems software? One way to find bugs is to use testing, but "testing shows the presence, not the absence of bugs".[1] This dissertation instead focuses on another approach, *formal verification*, as verification can ensure the absence of whole classes of bugs and vulnerabilities.

It is well-understood that formal verification is a useful technique for increasing the reliability of low-level systems software, so it is not surprising that verification of systems code is a large and active area of research. Highlights include the verification of the seL4[2] and CertiKOS[3] kernels, push-button verification of systems code,[4] and various tools and frameworks for modular interactive and automated verification of C code.[5]

While the previous research achieved impressive results, verification of systems code is far from being a "solved problem". In particular, we[6] want to highlight three important challenges when verifying systems code: Dealing with *realistic* systems code and programming languages, ensuring the *trustworthiness* of the verification technique, and conducting the verification as *automatically* as possible. Let us consider each of these challenges in more detail:

Challenge #1: Realism. The first step of any verification is to (formally or informally) model the system to verify. The challenge in this step is to *realistically* model the complexities of real-world languages and how they are used. In particular, there is a wide range of possible ways to model a program for verification. On the one hand, a language with local variables, if statements, and while loops is sometimes called a "C-like" language.[7] On the other hand, we have the complex definition of C according to the C standard,[8] with features like goto, the ability to take the address of local variables, integer-pointer-casts, and concurrency. Thus, verification techniques need to find a balance between making modeling tractable

[1] Buxton and Randell, "Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO", 1970 [BR70].

[2] Klein et al., "seL4: Formal Verification of an OS Kernel", 2009 [Kle+09]; Klein et al., "Comprehensive Formal Verification of an OS Microkernel", 2014 [Kle+14].

[3] Gu et al., "Building Certified Concurrent OS Kernels", 2019 [Gu+19].

[4] Nelson et al., "Scaling symbolic evaluation for automated verification of systems code with Serval", 2019 [Nel+19].

[5] Cohen et al., "VCC: A Practical System for Verifying Concurrent C", 2009 [Coh+09]; Rondon et al., "Low-Level Liquid Types", 2010 [RKJ10]; Appel, *Program Logics for Certified Compilers*, 2014 [App14]; Cao et al., "VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs", 2018 [Cao+18]; Cuoq et al., "Frama-C: A Software Analysis Perspective", 2012 [Cuo+12]; Jacobs et al., "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java", 2011 [Jac+11].

[6] As is standard scientific practice, the first person plural is used to refer to work done by the author, independent of whether it was done in collaboration with others. For further details, see §1.6.

[7] Wang et al., "Compiler Verification Meets Cross-Language Linking via Data Abstraction", 2014 [WCC14].

[8] How much the C standard agrees with how C is used in practice is still a topic of active discussion, see *e.g.*, Memarian et al., "Into the Depths of C: Elaborating the De Facto Standards", 2016 [Mem+16].

via simplifying assumptions and capturing the relevant aspects of the program to verify.

Challenge #2: Trustworthiness. With the model of the program in hand, the verification technique should provide a high degree of confidence in its *trustworthiness*, *i.e.*, one should be able to trust that a successful verification actually guarantees the desired properties hold about (the model of) the program. The gold standard for increasing this confidence is via *foundational* proofs, which allow a general-purpose proof assistant (like Coq[9] or Isabelle[10]) to check the result of the verification. With foundational proofs, one only needs to trust the correctness of the core of the proof assistant instead of the verification process that generated the foundational proof. However, generating such proofs is not easy (*e.g.*, the SMT solvers that underlie many verification tools rely on complex heuristics that do not come with foundational correctness proofs) and thus verification techniques might not provide foundational proofs for all parts of the verification.

[9] Coq team, *The Coq proof assistant*, 2023 [Coq23].

[10] Isabelle team, *The Isabelle proof assistant*, 2023 [Isa23].

Challenge #3: Automation. Another challenge when verifying low-level systems code is achieving a high degree of *automation*. Since program verification in general is undecidable, completely automatic verification is impossible. Instead, verification techniques often rely on the user to help complete the verification. This help can take the form of, for example, annotations in the source code (*e.g.*, for preconditions, postconditions, or loop invariants) or more detailed instructions (*e.g.*, an interactive proof in a proof assistant). While more help from the user simplifies the task of the verification technique, it places additional burden on the user. Thus, verification techniques strive to reduce the amount of help required by providing a high degree of automation.

Contributions of this dissertation. This dissertation presents new approaches for the verification of low-level systems code that push the boundaries on the three dimensions described above. Concretely, this dissertation describes three projects, covering three different aspects of the verification of low-level systems code:

1. RefinedC:[11] Automating the foundational verification of C code with refined ownership types
2. Islaris:[12] Verification of machine code against authoritative ISA semantics
3. DimSum:[13] A decentralized approach to multi-language semantics and verification

Let us now introduce these projects in more detail.

[11] Sammler et al., "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types", 2021 [Sam+21b].

[12] Sammler et al., "Islaris: Verification of Machine Code Against Authoritative ISA Semantics", 2022 [Sam+22].

[13] Sammler et al., "DimSum: A Decentralized Approach to Multi-language Semantics and Verification", 2023 [Sam+23].

1.1 *RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types*

The first project, *RefinedC*, provides the first approach for verifying C code that combines foundational proofs with a high degree of automation.

Previous approaches for verifying C code either provide automatic proofs but without a foundational guarantee of correctness,[14] or obtain foundational proofs by requiring the user to conduct interactive proofs in a proof assistant.[15] RefinedC addresses this gap by providing a type system combining ownership types (for modular reasoning about shared state) with refinement types (for encoding functional correctness specifications). This type system is build on top of the Iris program logic[16] in the Coq proof assistant and is automated using *Lithium*, a novel language for building automated and foundational verification tools. RefinedC has been validated on a range of interesting C code, including code from real-world hypervisors.

[14] Cohen et al., "VCC: A Practical System for Verifying Concurrent C", 2009 [Coh+09]; Cuoq et al., "Frama-C: A Software Analysis Perspective", 2012 [Cuo+12]; Jacobs et al., "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java", 2011 [Jac+11].

[15] Appel, *Program Logics for Certified Compilers*, 2014 [App14]; Klein et al., "seL4: Formal Verification of an OS Kernel", 2009 [Kle+09]; Gu et al., "Building Certified Concurrent OS Kernels", 2019 [Gu+19].

[16] Jung et al., "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning", 2015 [Jun+15]; Jung et al., "Higher-Order Ghost State", 2016 [Jun+16]; Krebbers et al., "The Essence of Higher-Order Concurrent Separation Logic", 2017 [Kre+17]; Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b]; Jung, "Understanding and Evolving the Rust Programming Language", 2020 [Jun20].

[17] Armstrong et al., "ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS", 2019 [Arm+19a]; Armstrong et al., *Sail ARMv8.5-A ISA model*, 2019 [Arm+19b]; Mundkur et al., *Sail RISC-V ISA model*, 2021 [Mun+21].

1.2 Islaris: Verification of Machine Code Against Authoritative ISA Semantics

The aim of the second project, *Islaris*, is to scale verification of assembly code to realistic models of modern instruction set architectures (ISAs). Recent work used Sail to develop formal models of large real-world architectures, including Armv8-A and RISC-V.[17] These models are comprehensive (complete enough to boot an operating system or hypervisor) and authoritative (automatically derived from the Arm internal model and validated against the Arm validation suite, and adopted as the official formal specification by RISC-V International, respectively). But the scale and complexity of these models makes them challenging to use as a basis for verification. Islaris is the first system to support verification of machine code against these real-world ISA models. Islaris achieves this goal via a novel combination of SMT-based symbolic execution (to prune irrelevant parts of the model) with automated reasoning in an Iris-based foundational program logic (to verify code against the relevant parts of the model). Islaris can handle Armv8-A and RISC-V machine-code exercising a wide range of system features, including installing and calling exception vectors, unaligned access faults, and compiled C code using inline assembly and function pointers.

1.3 DimSum: A Decentralized Approach to Multi-language Semantics and Verification

The previous two projects show how to verify C and assembly code in isolation, but systems programs often combine C with assembly components. The third project, *DimSum*, proposes a new *decentralized* approach to reasoning about such multi-language programs. Decentralization means that one can define and reason about languages independently, but also relate different languages when necessary. This allows a tool like RefinedC to purely focus on verification of code at the C level, while being able to relate the C code to assembly code in an independent step. DimSum's idea is to take inspiration from process calculi and model program components in a multi-language program as independent modules that communicate via events. These modules are manipulated using combinators that can translate modules from one set of events to another set of events and link

modules with the same events. DimSum has been applied to idealized versions of C and assembly, which capture the core challenges that arise when building and verifying multi-language programs.

1.4 Overview

This dissertation is composed of four parts:

Part I gives a detailed explanation of the Lithium programming language that forms the basis of the automation for RefinedC and Islaris.

The remaining parts—Part II, Part III, and Part IV— describe the three main projects of this dissertation—RefinedC, Islaris and DimSum—respectively.

1.5 Publications

This dissertation contains the work of the following papers:

1. Sammler et al., "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types", 2021 [Sam+21b], appeared in PLDI 2021
2. Sammler et al., "Islaris: Verification of Machine Code Against Authoritative ISA Semantics", 2022 [Sam+22], appeared in PLDI 2022
3. Sammler et al., "DimSum: A Decentralized Approach to Multi-language Semantics and Verification", 2023 [Sam+23], appeared in POPL 2023

This dissertation reuses much of the text of these papers, but adapts it into a consistent presentation. In particular, the following describes the provenance of the text of this dissertation:

- Part I contains a new presentation of Lithium, which underlies the automation of RefinedC and Islaris. The text in this part is new, except for §4.1, §4.2, and some parts of §5, which are adapted from the RefinedC paper.
- Part II is based on the RefinedC paper and its appendix.[18] §9.2 is new text and §9.3 has been adapted for the new presentation of Lithium.
- Part III is based on the Islaris paper. §16.3 has been adapted for the new presentation of Lithium.
- Part IV is based on the DimSum paper.
- §26 is mostly new text, but some parts are based on the future work sections of the RefinedC, Islaris, and DimSum papers.

[18] Sammler et al., *Artifact and Appendix of "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types"*, 2021 [Sam+21a].

1.6 Collaborations

The projects presented in this dissertation are the result of many productive and inspiring collaborations. Even though I[19] led all these collaborations, these projects would not have been possible without my collaborators. In the following, I detail my contributions to each of them:

[19] In this section, I use the first person singular to clearly distinguish my contributions from those of my collaborators.

For RefinedC, I designed and implemented Caesium (in discussion with Robbert Krebbers), Lithium (both its original form and its new presentation used in this dissertation, the latter inspired by a discussion with Lennard Gäher), and the type system, and led the writing. The implementation of the frontend was spearheaded by Rodolphe Lepigre with support from Kayvan Memarian. The verification of the case studies was a joint effort led by me and Rodolphe Lepigre.

For Islaris, the original idea of using Isla to combine the Sail models with Iris came from Angus Hammond. Angus Hammond and I jointly designed and implemented the operational semantics of ITL. I developed the automation using Lithium, the adequacy theorem, conducted the translation validation of Isla against the Sail-generated Coq code for RISC-V, and led the writing (the last together with Peter Sewell and Angus Hammond). Rodolphe Lepigre (together with Angus Hammond) developed the frontend, while Brian Campbell adapted Isla to support Islaris. The pKVM exception handler case study was verified by Angus Hammond (including adding the support for parametric traces), while the rest of the case studies were a collaborative effort of Rodolphe Lepigre, Angus Hammond, and me.

For DimSum, I developed the original idea, designed the notion of modules and refinement (with Simon Spies, Derek Dreyer, and me jointly simplifying my original definition of refinement to the equivalent, but simpler definition presented in this dissertation), developed the combinators, implemented the languages, implemented and verified the compiler (except for the *Mem2Reg* pass, which was implemented and verified by Simon Spies), proved the meta-level properties of the combinators, and verified the examples (Youngju Song suggested the coroutine example and implemented a first version of it). The writing was led jointly by me and Simon Spies.

All projects presented in this dissertation are open-source. The implementations and evaluations of the projects can be found at the following URLs:

- Lithium (Part I): `https://gitlab.mpi-sws.org/iris/refinedc` (maintained together with RefinedC)
- RefinedC (Part II): `https://gitlab.mpi-sws.org/iris/refinedc`
- Islaris (Part III): `https://github.com/rems-project/islaris`
- DimSum (Part IV): `https://gitlab.mpi-sws.org/iris/dimsum`

Part I

Lithium

Chapter 2

Introduction

This dissertation tackles the challenge of building techniques for verifying realistic, low-level programs. However, before we can dive into the details of these techniques—in particular, RefinedC (Part II) and Islaris (Part III)—in the later parts of this dissertation, we first need to introduce the foundations on which these techniques are built.

The first part of these foundations is *separation logic*.[1] Separation logic was designed as a logic for *modular* reasoning about *pointer manipulating programs*—an ideal fit for our setting, as low-level programs naturally contain a lot of pointer manipulation and modular reasoning enables composing the verification of large code-bases from the independent verifications of individual functions. Separation logic achieves this via its key notion of *ownership*: Propositions in separation logic are not just true or false, but one can acquire and give up ownership of propositions. This feature allows separation logic propositions to describe changes in the program state during verification. For example, verifying a write to memory in separation logic requires giving up ownership of a "points-to" proposition that owns the memory and afterwards, one acquires a points-to proposition for the same memory updated to the new value.[2] Separation logic has been applied in many contexts and its principle of ownership has proven crucial to the verification of many complex programs.[3] So, separation logic is an ideal starting point for the verification techniques described in this dissertation.[4]

Recent work on using separation logic for program verification has made advances in two different directions: On the one hand, tools like Viper[5] or VeriFast[6] tackle the challenge of *automatically* verifying programs based on specifications and other annotations given by the user. On the other hand, formalizations of separation logic in proof assistants show that separation logic enables building complex reasoning principles—*e.g.*, for reasoning about challenging aspects of real-world languages like C[7] or for verifying subtle concurrent algorithms[8]—that are *trustworthy* since the soundness of the reasoning principles is ensured by a foundational (*i.e.*, machine-checked) proof.

These two directions have been studied independently—*e.g.*, VeriFast and Viper don't provide foundational proofs and verification using proof assistants heavily relies on manual guidance by the user. However, building verification tools for realistic low-level languages using separation logic requires a foundation that provides both a high degree of automation and trustworthiness. To provide this foundation, we developed *Lithium*.

[1] O'Hearn et al., "Local Reasoning about Programs that Alter Data Structures", 2001 [ORY01]; Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures", 2002 [Rey02].

[2] §3.5 describes this reasoning principle in more detail.

[3] For example, Dang et al., "Compass: Strong and Compositional Library Specifications in Relaxed Memory Separation Logic", 2022 [Dan+22]; Sprenger et al., "Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification", 2020 [Spr+20].

[4] An introduction to separation logic as used in this dissertation can be found in §2.1.

[5] Müller et al., "Viper: A Verification Infrastructure for Permission-Based Reasoning", 2016 [MSS16b].

[6] Jacobs et al., "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java", 2011 [Jac+11].

[7] Appel, *Program Logics for Certified Compilers*, 2014 [App14].

[8] Jung et al., "The Future is Ours: Prophecy Variables in Separation Logic", 2020 [Jun+20]; Dang et al., "Compass: Strong and Compositional Library Specifications in Relaxed Memory Separation Logic", 2022 [Dan+22].

Lithium and its philosophy. Lithium is the first programming language for writing separation logic verification tools that are both automated and foundational. At the core of Lithium is the philosophy that the best way to solve a large problem—like verifying a C function—is to break the large problem up into many small problems until each problem is so small that its solution is straightforward. Concretely, building a verification tool using Lithium consists of writing a set of *Lithium rules* that each handle one step of the verification—*e.g.*, decomposing an expression or verifying a specific construct of the object language.[9] Lithium then automatically combines these rules when verifying a concrete program.

This decomposition-based approach is what enables Lithium to combine automated and foundational verification: Since the rules only consider one step of the verification, a small set of rules can cover a large variety of programs, as the rules can be combined in many ways. In particular, Lithium uses the structure of the program to automatically compose the rules in different ways, and thus Lithium-based verification can cover a broad range of (syntactically different) variants of a program. Additionally, this rule-based approach enables Lithium to automatically construct a soundness proof along with the verification by combining the soundness proofs of the individual rules.

The decomposition bottoms out in a set of primitive instructions provided by Lithium. These primitives are carefully chosen such that they, on the one hand, can be efficiently automated, and, on the other hand, are expressive enough to serve as the foundation for tools like RefinedC or Islaris. Each of these primitives fulfills one specific purpose—*e.g.*, acquiring or releasing separation logic ownership, proving pure side conditions, or splitting the verification into multiple cases.[10] The purpose of these primitives is *not* to provide a powerful proof search procedure for complex separation logic entailments *on their own*, but instead to give the user of Lithium the right tools to implement such a proof search procedure for the object language they care about. In particular, the primitives of Lithium are independent of the object language and the separation logic assertions used to verify programs written in the object language.[11] We will see in later parts of this dissertation how this flexibility enables building both RefinedC and Islaris on top of Lithium, even though these tools consider very different languages (C vs. assembly) and implement different strategies for proof search (type-based vs. direct separation logic).

Lithium is implemented as a shallowly embedded language in the Coq proof assistant based on the Iris framework for separation logic[12] and comes with an interpreter (written in the Ltac tactic language[13]) for executing Lithium programs.

§3 gives an overview of Lithium by showing how to use Lithium to build a verification tool for a simple language. The formal definition and semantics of Lithium are described in §4. But before we dive into the details of Lithium, we introduce the basics of separation logic that are used in the rest of this dissertation.

[9] It is important to distinguish the object language in which the programs to verify are written (*e.g.*, C for RefinedC), from the language provided by Lithium, and from the meta-level language that Lithium is embedded in (*i.e.*, Coq).

[10] The full list of Lithium primitive instructions is shown in §4.

[11] In particular, Lithium does not have a built-in notion of the points-to assertion, but the points-to assertion is treated like any other user-defined assertion.

[12] Jung et al., "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning", 2015 [Jun+15]; Jung et al., "Higher-Order Ghost State", 2016 [Jun+16]; Krebbers et al., "The Essence of Higher-Order Concurrent Separation Logic", 2017 [Kre+17]; Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b]; Jung, "Understanding and Evolving the Rust Programming Language", 2020 [Jun20].

[13] Delahaye, "A Tactic Language for the System Coq", 2000 [Del00].

$$\begin{aligned}\text{base logic: } P, Q ::= {} & \mathsf{True} \mid \mathsf{False} \mid P * Q \mid P \mathrel{-\!\!*} Q \mid P \wedge Q \mid P \vee Q \\ & \mid \forall x.\, P(x) \mid \exists x.\, P(x) \mid \ulcorner \phi \urcorner \mid \Box P \mid R \\ \text{program logic: } R ::= {} & l \mapsto v \mid \mathsf{wp}\, e\, \{v.P\} \mid \ldots\end{aligned}$$

Figure 2.1: Syntax of part of the Iris separation logic.

2.1 Separation Logic Primer

All the work in this dissertation is based on the formalization of separation logic provided by the Iris framework.[14] Iris is an ideal basis for the work in this dissertation since it provides an expressive, language-generic separation logic that comes with a formalization in the Coq proof assistant. The fragment of the Iris separation logic that is relevant for this dissertation is shown in Figure 2.1.[15] First, Iris provides a language-independent base logic. Iris propositions can be seen as predicates over some notion of resource (*e.g.*, memory). A separation conjunction $P * Q$ asserts that P and Q hold, but for disjoint resources. A magic wand $P \mathrel{-\!\!*} Q$ asserts that one can give up ownership of P to obtain ownership of Q. A (non-separating) conjunction $P \wedge Q$ states that both P and Q hold, but their resources are not necessarily disjoint. A disjunction $P \vee Q$ asserts that either P or Q holds. Iris also provides higher-order universal and existential quantification ($\forall x.\, P(x)$ and $\exists x.\, P(x)$). The $\ulcorner \phi \urcorner$ connective embeds the pure Coq assertion ϕ (*i.e.*, an assertion independent of resources) into the separation logic. The persistent modality $\Box P$ states that P holds for resources that are not exclusive, which means that $\Box P$ can be duplicated, *i.e.*, we have $\Box P \vdash P$ and $\Box P \vdash \Box P * \Box P$.

In addition to the Iris base logic, Iris also provides the ability to create a program logic for a specific object language. The exact connectives of the program logic depend on the object language, but one common connective is the points-to connective $l \mapsto v$ that asserts ownership of the memory location l and states that the location l contains the value v. The workhorse of Iris-based program verification is the weakest precondition $\mathsf{wp}\, e\, \{v.P\}$: It asserts that the expression e has been verified and that after executing e the postcondition P holds for the resulting value v. The goal of an Iris-based program verification is to prove such a weakest precondition for the program and then use a general adequacy statement to lift this Iris proof to a pure (Coq) proof that states that the program is well-behaved.[16] Since this lifted proof is independent of Iris, one only needs to trust the Coq proof assistant, but not Iris itself.

[14] Jung et al., "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning", 2015 [Jun+15]; Jung et al., "Higher-Order Ghost State", 2016 [Jun+16]; Krebbers et al., "The Essence of Higher-Order Concurrent Separation Logic", 2017 [Kre+17]; Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b]; Jung, "Understanding and Evolving the Rust Programming Language", 2020 [Jun20].

[15] A more in depth description of the Iris separation logic is provided by Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

[16] Usually, this means that e does not trigger undefined behavior.

Chapter 3

Lithium by Example

This chapter explains Lithium on the example of building an automated verifier for the language depicted in Figure 3.1.

$$\begin{array}{rl}
\mathsf{Literal} \ni t ::= & z \mid b \mid l \mid \mathsf{NULL} \qquad \mathsf{Val} \ni v ::= \#t \mid (v_1, v_2) \mid \mathsf{fn}\ f(\mathrm{x}) \triangleq e \\
\mathsf{BinOp} \ni \oplus ::= & + \mid - \mid = \mid (_, _) \qquad \mathsf{UnOp} \ni u ::= \mathsf{fst}(_) \mid \mathsf{snd}(_) \\
\mathsf{Expr} \ni e ::= & v \mid \mathrm{x} \mid e_1\ e_2 \mid u\ e \mid e \oplus e \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{assert}(e) \mid \mathsf{alloc} \mid\ !e \mid e_1 \leftarrow e_2 \\
& \mathsf{let}\ \mathrm{x} := e_1\ \mathsf{in}\ e_2 \triangleq (\mathsf{fn}\ _(\mathrm{x}) \triangleq e_2)\ e_1 \qquad e_1; e_2 \triangleq \mathsf{let}\ _ := e_1\ \mathsf{in}\ e_2 \\
& (e_1, e_2) \triangleq e_1\ (_, _)\ e_2 \qquad \mathsf{fst}(e) \triangleq \mathsf{fst}(_)\ e \qquad \mathsf{snd}(e) \triangleq \mathsf{snd}(_)\ e
\end{array}$$

Figure 3.1: Grammar of the language.

This language[1] is a simple functional language with integers z, Booleans b, and locations l. Locations refer to memory on the heap allocated with alloc and can be read via $!e$ and assigned via $e_1 \leftarrow e_2$. The $\mathsf{assert}(e)$ statement checks that e evaluates to true and raises an error otherwise.[2]

Overview of the chapter. This chapter first describes the basics of Lithium (§3.1), introduces the operational semantics of Lithium (§3.2), shows how the primitives of Lithium can be used to conduct modular verification (§3.3), and explains the continuation passing style used by Lithium (§3.4). Afterwards, we turn to separation logic: §3.5 introduces the basics of reasoning about separation logic propositions using Lithium, and then we see how Lithium can be used to reason about abstract predicates (§3.6) and higher-order functions (§3.7). Finally, §3.8 describes the semantic model of Lithium along with the guarantees provided by a successful verification.

Formalization. A formalization of the language and the Lithium verifier presented in this chapter can be found in the RefinedC repository at the following URL: `https://gitlab.mpi-sws.org/iris/refinedc/-/tree/ci/lithium-dissertation-sammler/tutorial/lithium`. The version at this URL corresponds to the version of Lithium at the time of writing this dissertation. Subsequent updates to the formalization will be published on the main branch of the RefinedC repository (`https://gitlab.mpi-sws.org/iris/refinedc`).

[1] This language is an adapted version of Iris's HeapLang language (Jung et al., "Higher-Order Ghost State", 2016 [Jun+16]) and the SimpLang language (Chajed, *SimpLang*, 2023 [Cha23]).

[2] More precisely, failed asserts cause undefined behavior.

3.1 Lithium Basics

Let us get started with verifying our first program using Lithium. Concretely, we tackle the problem of proving that $1 + 1 = 2$ by verifying that the assert in the following program assert_two succeeds:

$$\mathsf{assert\_two} \triangleq \mathsf{let}\ \mathrm{x} := 1\ \mathsf{in}\ \mathsf{let}\ \mathrm{y} := \mathrm{x} + 1\ \mathsf{in}\ \mathsf{assert}(\mathrm{y} = 2)$$

To verify this program, we first declare a Lithium function exprOk e for verifying the expression e and returning a (potentially symbolic) value representing the result of e. To verify assert_two, we then just need to run the following Lithium program:

$$\_ \leftarrow \mathsf{exprOk}\ \mathsf{assert\_two};\ \mathbf{done}$$

This program first calls exprOk on assert_two and then successfully terminates using **done**. However, if we now run this program using the Lithium interpreter, nothing happens (*i.e.*, the Lithium interpreter immediately gets stuck) since we have not yet defined what exprOk should do.

EXPR-LET

1: exprOk (let x := e_1 in e_2) G :-
2: $\quad v_1 \leftarrow$ exprOk e_1;
3: $\quad v_2 \leftarrow$ exprOk (e_2[x $\mapsto v_1$]);
4: $\quad$ **return**$_G$ v_2

EXPR-BINOP

1: exprOk ($e_1 \oplus e_2$) G :-
2: $\quad v_1 \leftarrow$ exprOk e_1;
3: $\quad v_2 \leftarrow$ exprOk e_2;
4: $\quad v \leftarrow$ binopOk $v_1 \oplus v_2$;
5: $\quad$ **return**$_G$ v

EXPR-ASSERT

1: exprOk assert(e) G :-
2: $\quad v \leftarrow$ exprOk e;
3: $\quad$ **exhale** $\ulcorner v = \#\mathsf{true}\urcorner$;
4: $\quad$ **return**$_G$ 0

exprOk v G :- **return**$_G$ v

binopOk $z_1 + z_2$ G :- **return**$_G$ $\#(z_1 + z_2)$

binopOk $z_1 - z_2$ G :- **return**$_G$ $\#(z_1 - z_2)$

binopOk $z_1 = z_2$ G :- **return**$_G$ $\#(z_1 = z_2)$

Figure 3.2: Basic rules for exprOk and binopOk.

We define exprOk by giving rules for specific program constructs as shown in Figure 3.2.[3] Function definitions in Lithium are based on pattern-matching: When Lithium needs to evaluate a call to a user-declared function like exprOk it searches for a matching rule (*i.e.*, where the call matches the part of the rule before :-) and then executes the body of the rule (*i.e.*, the part of the rule after :-).[4] When the execution reaches a **return**$_G$ instruction, it continues with the code after the call. (The continuation G and the details of this call mechanism are explained in §3.4.) For example, EXPR-LET states that to verify let x := e_1 in e_2, we first verify e_1 and then e_2 with the variable x replaced by the resulting value v_1 of e_1. Verifying a binary operation $e_1 \oplus e_2$ (rule EXPR-BINOP) is defined as verifying the expressions e_1 and e_2 and then calling a new Lithium function binopOk with the resulting values. binopOk is defined by mapping the binary operations of the language to their mathematical counterparts.[5] The most interesting rule is the rule EXPR-ASSERT for assert(e): Here we encode the check that all assertions succeed by using Lithium's **exhale** statement to tell Lithium to prove that the value returned by e is equal to true.[6] In the case of assert_two, this **exhale** results in the side condition $\#(1 + 1 = 2) = \#\mathsf{true}$ that can be trivially discharged.[7] Running the Lithium program from above with the rules from Figure 3.2 succeeds, which means we have verified our first program using Lithium!

[3] Constructs provided by Lithium like **done** or **return** are typeset in **blue, bold**, while definitions specific to this example like exprOk are depicted in black.

[4] This style of defining functions is reminiscent of predicate declarations in logic programming, but with the important difference that Lithium uses the first matching rule and *does not* backtrack on this choice.

[5] The $\#t$ operator lifts the literal t to a value, see Figure 3.1.

[6] We need to turn the pure proposition $v = \#\mathsf{true}$ into a separation logic proposition using $\ulcorner \ldots \urcorner$ since **exhale** is defined on a separation logic propositions (see §3.5).

[7] Lithium is generic over how to solve pure side conditions—here the standard Coq `done` tactic suffices.

3.2 Operational Model

Before we discuss more of the features provided by Lithium, let us look at how Lithium programs are executed by the Lithium interpreter. The state of the Lithium interpreter is given by the judgment

$$\Gamma; \Delta \Vdash \exists \vec{x}.\, G(\vec{x})$$

The *goal* G contains the Lithium program that needs to be executed. We have already seen some goals:[8]

$$G ::= x \leftarrow F\mathbf{;}\ G \mid \mathbf{exhale}\ H\mathbf{;}\ G \mid \dots$$

The goal can depend on a (potentially empty) list of existentially quantified variables $\vec{x}$.[9] In the following, we call such a variable in $\vec{x}$ a *(Lithium) existential*.[10] The pure context Γ tracks binders x and pure facts ϕ, while the resource context Δ contains separation logic propositions A and persistent separation logic propositions $\square A$, *i.e.*, we have

$$\Gamma ::= \emptyset \mid \Gamma, x \mid \Gamma, \phi \qquad \Delta ::= \emptyset \mid \Delta, A \mid \Delta, \square A$$

The Lithium interpreter is defined via a small-step transition relation $\Gamma_1; \Delta_1 \Vdash \exists \vec{x}_1.\, G_1(\vec{x}_1) \Rightarrow \Gamma_2; \Delta_2 \Vdash \exists \vec{x}_2.\, G_2(\vec{x}_2)$.[11,12]

$$\frac{\Gamma \vdash \phi}{\Gamma; \Delta \Vdash \mathbf{exhale}\ \ulcorner \phi \urcorner\mathbf{;}\ G \Rightarrow \Gamma; \Delta \Vdash G}\ \textsc{li-exhale-pure}$$

For example, exhaling a pure proposition ϕ is handled by the rule LI-EXHALE-PURE, which uses a solver to prove ϕ using the assumptions in Γ and then continues with the rest of the program. This rule only applies when ϕ does not depend on existentials (note the absence of $\exists \vec{x}.$ in LI-EXHALE-PURE).[13] §4.2 describes how Lithium handles side conditions that contain existentials.

The Lithium interpreter executes the Lithium program using the $\Rightarrow$ relation until it (a) reaches **done** and the verification succeeds or (b) it gets stuck and the verification fails.

3.3 Modular Verification via Inhale, Exhale, and Quantifiers

Now that the basics of Lithium are out of the way, let us introduce the most important primitives that Lithium provides: We have already seen **exhale** $\ulcorner \phi \urcorner$ for asserting that Lithium should prove the proposition ϕ. Dually, the **inhale** $\ulcorner \phi \urcorner$ instruction adds ϕ as an assumption for the further verification. These primitives correspond to "assert" and "assume" primitives in other tools that only deal with pure propositions ϕ. However, as we will see in §3.5, **exhale** and **inhale** work not only on pure propositions, but also allow adding and removing (separation logic) ownership, so we follow the terminology of Chalice[14] and Viper.[15] Additionally, Lithium allows introducing and providing (meta-level) variables using $\forall$ and $\exists$. The operational semantics of these instructions are given in Figure 3.3. The rule LI-EXIST turns an existential quantifier into an existential that is instantiated by the further proof search.[16]

[8] The complete list of goals can be found in §4.3. The left goals H are introduced in §3.5.

[9] We omit $\exists \vec{x}.$ when $\vec{x}$ is empty.

[10] Note that these existentials are different from the evars ?x of the meta-logic.

[11] In the Coq implementation, these transitions are implemented in Ltac.

[12] A Lithium transition can also step to multiple Lithium states, which are then executed independently. This feature is used by Lithium's branching constructions, introduced in §3.4 and §3.7.

[13] In the case that there are existentials $\vec{x}$ but ϕ does not depend on them, Lithium automatically commutes $\exists \vec{x}.$ into the goal G.

[14] Leino and Müller, "A Basis for Verifying Multi-threaded Programs", 2009 [LM09].

[15] Müller et al., "Viper: A Verification Infrastructure for Permission-Based Reasoning", 2016 [MSS16b].

[16] §4.2 describes Lithium's handling of existentials in more detail.

LI-INHALE-PURE
$$\Gamma; \Delta \Vdash \textbf{inhale}\ \ulcorner\phi\urcorner; G \Rightarrow \Gamma, \phi; \Delta \Vdash G$$

LI-ALL
$$\Gamma; \Delta \Vdash \forall x.\, G(x) \Rightarrow \Gamma, x; \Delta \Vdash G(x)$$

LI-EXIST
$$\Gamma; \Delta \Vdash \exists \vec{x}.\, \exists x.\, G(x, \vec{x}) \Rightarrow \Gamma; \Delta \Vdash \exists x, \vec{x}.\, G(x, \vec{x})$$

Figure 3.3: Operational semantics for **inhale**, $\forall$, and $\exists$.

To illustrate these operations, let us now see how these primitives can be used to enable modular verification. For this, consider the following simple function:

$$\mathsf{fn}\ \mathsf{add1}(\mathrm{x}) \triangleq \mathrm{x} + 1$$

First, let us consider how we would specify this function using standard Hoare triples:

$$\forall v_{arg}\ z.\, \{\ulcorner v_{arg} = \#z\urcorner\}(\mathsf{fn}\ \mathsf{add1}(\mathrm{x}) \triangleq \ldots)\ v_{arg}\{v_{ret}.\, \ulcorner v_{ret} = \#(z+1)\urcorner\}$$

This specification says that if the add1 function is applied to an argument v_{arg} that is equal to some integer z, then it will execute safely and its return value v_{ret} will be equal to $z + 1$. However, this is not the only way one could write this specification, *e.g.*, one could swap the quantifiers in the beginning or directly specialize v_{arg} to $\#z$. Since dealing with many specification formats automatically is challenging, we fix one format by defining a new fnOk predicate to specify a function v:[17]

$$\mathsf{fnOk}_A\ \{pre\}\ v\ \{post\} \triangleq \forall v_{arg}\ a.\, \{pre\ a\ v_{arg}\}\ v\ v_{arg}\ \{v_{ret}.\ post\ a\ v_{ret}\}$$

With this predicate, we can specify add1 as follows:

$$\mathsf{fnOk}_{\mathbb{Z}}\ \{z\ v_{arg}.\, \ulcorner v_{arg} = \#z\urcorner\}\ (\mathsf{fn}\ \mathsf{add1}(\mathrm{x}) \triangleq \ldots)\ \{z\ v_{ret}.\, \ulcorner v_{ret} = \#(z{+}1)\urcorner\}$$

$\mathsf{fnOk}_A\ \{pre\}\ v\ \{post\}$ asserts that the value v is a function that is safe to call with an argument v_{arg} satisfying the precondition *pre* and the resulting value v_{ret} will satisfy the postcondition *post*. The pre- and postconditions are also parametrized by a shared parameter a of type A[18] that corresponds to a universal quantifier spanning both pre- and postcondition (*i.e.*, z of type $\mathbb{Z}$ in the Hoare triple shown initially). This parameter can be used to transfer information from the precondition to the postcondition. In the example above, this parameter is used to make the value of the argument available to the postcondition: the specification says that add1 must be called with an integer argument z and then returns the value $z + 1$.

To verify that the add1 function actually satisfies this specification, we use the rule FNOK (Figure 3.4):[19] To show that a function satisfies a specification, we are first given an arbitrary parameter a, argument v_{arg}, and value for the recursive call v_f —expressed using $\forall a.\, \forall v_{arg}.\, \forall v_f.$ —and we assume that they satisfy the precondition *pre*—expressed using **inhale**. Additionally, we can assume that the recursive occurrences of f already satisfy the specification—expressed by the second **inhale**. Then, we use exprOk to verify the body of the function with the argument x substituted with v_{arg} and the recursive occurrence f substituted with v_f. Finally,

[17] Technically, we need to use a slightly different definition to support verifying recursive functions, but the intuition given by this definition suffices for our purposes here.

[18] In the following, we write fnOk instead of fnOk_A when A can be inferred from the context.

[19] FNOK is not a rule that is used by Lithium automatically since fnOk is not a Lithium function, but we use it manually to create the Lithium program for verifying a function. §3.7 shows how to integrate this rule into the Lithium automation.

FNOK

```
1: fnOk {pre} fn f(x) ≜ e {post} :-
2:     ∀a. ∀v_arg. ∀v_f.
3:     inhale pre a v_arg;
4:     inhale fnOk {pre} v_f {post};
5:     v_ret ← exprOk (e[x ↦ v_arg][f ↦ v_f]);
6:     exhale post a v_ret;
7:     done
```

EXPR-APP

```
1: exprOk (e_1 e_2) G :-
2:     v_arg ← exprOk e_2;
3:     v ← exprOk e_1;
4:     pre, post ← find fnOk {−} v {−};
5:     ∃a. exhale pre a v_arg;
6:     ∀v_ret. inhale post a v_ret;
7:     return_G v_ret
```

```
1: find fnOk {−} v {−} G :- pattern pre post. fnOk {pre} v {post};
2:     return_G pre, post
```

Figure 3.4: Lithium rules for verifying and calling a function.

we have to show that the resulting value v_{ret} satisfies the postcondition *post*—expressed using **exhale**.

When we call a function, the verification behaves dually as shown by the rule EXPR-APP: After verifying e_1 and e_2 and finding a specification for v in the context,[20] one first needs to provide the parameter a—expressed using $\exists a.$ —and show the precondition—expressed using **exhale**. Then one gets an arbitrary return value—expressed using $\forall v_{ret}.$ —and can assume the postcondition—expressed using **inhale**.[21]

With the rules in Figure 3.4, Lithium can automatically verify that add1 satisfies the specification given above. We can then use this specification to verify an adapted version of assert_two using the following Lithium program:

```
∀v_add1. inhale fnOk {z v. ⌜v = #z⌝} v_add1 {z v'. ⌜v' = #(z + 1)⌝};
_ ← exprOk (let x := 1 in let y := v_add1 x in assert(y = 2)); done
```

Note that the verification of this variant of assert_two does not depend on the body of add1, but instead universally quantifies over it as v_{add1}. This demonstrates how Lithium can support modular verification.

3.4 Continuations

Let us now come back to a question that was left open in §3.1: What is the G parameter of exprOk? The answer is that function definitions in Lithium like exprOk are written in a *continuation passing style*: G is a continuation that contains the Lithium program that should be executed after exprOk returns. Formally, this is expressed by the following rules:

LI-FN

$$\frac{F\ (x.\ G) \text{ :- } G'}{\Gamma; \Delta \Vdash x \leftarrow F; G \Rightarrow \Gamma; \Delta \Vdash G'}$$

LI-RETURN

$$\Gamma; \Delta \Vdash \mathbf{return}_G\ x \Rightarrow \Gamma; \Delta \Vdash G(x)$$

Calling a user-defined function F (LI-FN) replaces the program with the body of a matching clause for F and the previous goal G is passed to F as a continuation. Returning from a function (LI-RETURN) calls the continuation with the return value.

To see why continuations are treated as explicit parameters in Lithium instead of being handled implicitly, consider the rules for verifying an if expression in Figure 3.5. We introduce a new Lithium function ifOk v for verifying an if expression on v. Importantly, ifOk is parametrized by

[20] **find** is discussed in §3.5.

[21] This use of **exhale** and **inhale** to encode the verification of function specifications is standard, see *e.g.*, Heule et al., "Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions", 2013 [Heu+13] or Vogels et al., "Featherweight VeriFast", 2015 [VJP15].

EXPR-IF
1: exprOk (if e_1 then e_2 else e_3) G :-
2: $v \leftarrow$ exprOk e_1;
3: ifOk v (exprOk e_2 G) (exprOk e_3 G)

IF-BOOL
1: ifOk $\#b$ G_1 G_2 :-
2: if b = true then
3: return$_{G_1}$
4: else
5: return$_{G_2}$

Figure 3.5: Rules for if expressions.

two continuations G_1 and G_2—G_1 for the then branch and G_2 for the else branch. EXPR-IF instantiates these continuations with exprOk for the corresponding expressions. Calling ifOk in EXPR-IF is not handled by LI-FN since it is not stated using the sequence operator ;, so we introduce a new Lithium step LI-FN':[22]

LI-FN'
$$\frac{F \text{ :- } G}{\Gamma; \Delta \Vdash F \Rightarrow \Gamma; \Delta \Vdash G}$$

[22] Lithium actually does not use LI-FN, but first unfolds $x \leftarrow F; G$ to $F\ (x.G)$ and then uses LI-FN', see §4.

IF-BOOL uses the two continuations: It first uses Lithium's **if** primitive to perform a case-distinction on whether b is true and then returns to G_1 or G_2 depending on the result.

LI-IF-TRUE
$$\frac{\Gamma \vdash \phi}{\Gamma; \Delta \Vdash \textbf{if } \phi \textbf{ then } G_1 \textbf{ else } G_2 \Rightarrow \Gamma; \Delta \Vdash G_1}$$

LI-IF-FALSE
$$\frac{\Gamma \vdash \neg\phi}{\Gamma; \Delta \Vdash \textbf{if } \phi \textbf{ then } G_1 \textbf{ else } G_2 \Rightarrow \Gamma; \Delta \Vdash G_2}$$

LI-IF
$$\Gamma; \Delta \Vdash \textbf{if } \phi \textbf{ then } G_1 \textbf{ else } G_2 \Rightarrow \Gamma, \phi; \Delta \Vdash G_1 \mid \Gamma, \neg\phi; \Delta \Vdash G_2$$

Figure 3.6: Operational semantics for **if**.

Figure 3.6 shows the Lithium steps for the **if** primitive. Lithium first tries to use a solver to prove which of the branches will be taken (via LI-IF-TRUE and LI-IF-TRUE). If this fails, Lithium verifies both branches with ϕ resp. $\neg\phi$ added to the context (LI-IF).[23]

[23] Note that LI-IF does not just step to one, but to two Lithium states (separated by "|") that are then executed independently.

As a concrete example, the rules for if expressions can be used to verify the following Fibonacci function:

fn fib(x) $\triangleq$ if x = 0 then 0 else if x = 1 then 1 else fib (x − 1) + fib (x − 2)

In particular, we can prove the property that calling the Fibonacci function on non-negative numbers results in non-negative numbers:

$$\text{fnOk } \{\_\ v.\ \exists z. \ulcorner v = z \urcorner * \ulcorner 0 \le z \urcorner\}\ \text{fn fib(x)} \triangleq \ldots\ \{\_\ v'.\ \exists z. \ulcorner v' = z \urcorner * \ulcorner 0 \le z \urcorner\}$$

With the rules we have seen so far (and Coq's `lia` tactic to solve the side conditions), Lithium automatically verifies this specification.[24]

[24] Technically, we also need to tell Lithium that fnOk is persistent and thus can be used multiple times.

3.5 Separation Logic

So far, we have seen how Lithium can be used to verify pure programs. Now, let us see how Lithium enables the verification of programs that interact with the heap by leveraging separation logic.

LI-EXHALE-STAR
$\Gamma; \Delta \Vdash \exists \vec{x}.\ \mathbf{exhale}\ H_1(\vec{x}) * H_2(\vec{x})\mathbf{;}\ G \Rightarrow \Gamma; \Delta \Vdash \exists \vec{x}.\ \mathbf{exhale}\ H_1(\vec{x})\mathbf{;}\ \mathbf{exhale}\ H_2(\vec{x})\mathbf{;}\ G(\vec{x})$

LI-EXHALE-EXIST
$\Gamma; \Delta \Vdash \exists \vec{x}.\ \mathbf{exhale}\ \exists x.\ H(x, \vec{x})\mathbf{;}\ G \Rightarrow \Gamma; \Delta \Vdash \exists \vec{x}.\ \exists x.\ \mathbf{exhale}\ H(x, \vec{x})\mathbf{;}\ G(\vec{x})$

LI-INHALE-STAR
$\Gamma; \Delta \Vdash \mathbf{inhale}\ H_1 * H_2\mathbf{;}\ G \Rightarrow \Gamma; \Delta \Vdash \mathbf{inhale}\ H_1\mathbf{;}\ \mathbf{inhale}\ H_2\mathbf{;}\ G$

LI-INHALE-EXIST
$\Gamma; \Delta \Vdash \mathbf{inhale}\ \exists x.\ H(x)\mathbf{;}\ G \Rightarrow \Gamma; \Delta \Vdash \forall x.\ \mathbf{inhale}\ H(x)\mathbf{;}\ G$

Figure 3.7: Operational semantics for exhaling and inhaling $H_1 * H_2$ and $\exists x.\ H(x)$.

The principal tools of Lithium to interact with the separation logic context are **exhale** H and **inhale** H, which don't just work on pure propositions, but on *left goals* H:

$$H ::= A \mid \ulcorner \phi \urcorner \mid H * H \mid \exists x.\ H(x) \mid \Box H$$

These left goals consist of pure propositions ϕ, atoms A, and are closed under separating conjunction, existential quantification and the persistence modality $\Box$. The operational semantics for existential quantification and separating conjunction (Figure 3.7) are straightforward.[25] The most interesting part of left goals are the atoms A. These atoms are picked by the user of Lithium as the predicates that the verification should manipulate.[26] In our running example, we have already seen one such atom: the function specification predicate fnOk. Another atom, taken from our next example, is the separation logic points-to assertion $v_1 \mapsto v_2$, which asserts that v_1 is a location that points to memory storing v_2.[27]

[25] The operational semantics of $\Box H$ are described in §4.3.

[26] Lithium simply treats all propositions that don't fall into any of the other categories as atoms.

[27] We use $v_1 \mapsto v_2$ instead of the standard $l \mapsto v$ for uniformity with the abstract predicate introduced in §3.6.

EXPR-ALLOC
```
1: exprOk alloc G :- ∀v. inhale v ↦ #0; return_G v
```

EXPR-LOAD
```
1: exprOk (!e1) G :-
2:     v1 ← exprOk e1;
3:     v2 ← find v1 ↦ −;
4:     inhale v1 ↦ v2;
5:     return_G v2
```

EXPR-STORE
```
1: exprOk (e1 ← e2) G :-
2:     v2 ← exprOk e2;
3:     v1 ← exprOk e1;
4:     _ ← find v1 ↦ −;
5:     inhale v1 ↦ v2;
6:     return_G v2
```

FIND-POINTS-TO
```
1: find v1 ↦ − G :- pattern v2. v1 ↦ v2; return_G v2
```

Figure 3.8: Rules for heap operations.

Figure 3.8 shows how this points-to assertion is used to define exprOk for allocations, loads and stores. The rule EXPR-ALLOC for alloc uses **inhale** to add a points-to fact for the newly allocated location to the context.

LI-INHALE-ATOM
$\Gamma; \Delta \Vdash \mathbf{inhale}\ A\mathbf{;}\ G \Rightarrow \Gamma; \Delta, A \Vdash G$

The **find** *primitive.* To discuss the rules for loads and stores, we first need to introduce Lithium's **find** primitive. The **find** primitive provides the ability to search for specific assertions in the context. This primitive

is used by the rules for loads and stores to find a points-to predicate in the context. Concretely, **find** is parametrized by a (user-defined) find function D—here $v \mapsto -$. The intuition for the find function $v \mapsto -$ is that it extracts a points-to assertion $v \mapsto v_2$ from the context and returns v_2. Formally, the behavior of a find functions is defined via Lithium **find** rules like FIND-POINTS-TO. In general, **find** rules have the following form:

find D G **:- pattern** $a_1 \dots a_n.$ A**;** $\dots$

The operational semantics of **find** are given by LI-FIND:

LI-FIND

$$\frac{\textbf{find } D\ G \textbf{ :- pattern } a.\ A(a)\textbf{; } G'(a) \qquad A(b) \in \Delta}{\Gamma; \Delta \Vdash x \leftarrow \textbf{find } D\textbf{; } G \Rightarrow \Gamma; \Delta \setminus \{A(b)\} \Vdash G'(b)}$$

When Lithium encounters **find** D, it searches for a **find** rule for D where the assertion A matches an assertion in the context. a specifies holes in A that should be filled using unification.[28] Once a **find** rule is found where the pattern $A(\_)$ matches an assertion $A(b)$ in the context,[29] Lithium removes the assertion $A(b)$ from the context and continues with the body of the **find** rule. Coming back to Figure 3.8, EXPR-LOAD and EXPR-STORE use **find** to find a point-to assertion for v_1 in the context—in the case of load, to return the value of the memory, and, in the case of store, to add an updated points-to assertion to the context. Note that since **find** removes the found assertion from the context, EXPR-LOAD also has to add the points-to assertion back to the context (using **inhale**).

[28] We use a single hole a here to avoid clutter, but the syntax supports multiple holes $a_1 \dots a_n$.

[29] The ability to match on the separation logic context is the unique features to **find** rules compared to normal Lithium rules.

Exhaling separation logic assertions. To discuss the semantics of **exhale** on atoms, let us consider an alternative, perfectly valid, version of the rule for loads that uses **exhale** instead of **find**:

EXPR-LOAD'

1: exprOk (**!**e_1) G **:-**
2: $v_1 \leftarrow$ exprOk e_1**;** $\exists v_2.$ **exhale** $v_1 \mapsto v_2$**; inhale** $v_1 \mapsto v_2$**; return**$_G$ v_2

When Lithium encounters an **exhale** A instruction (like **exhale** $v_1 \mapsto v_2$), it first tries to satisfy the **exhale** by finding and removing A from the context (*i.e.*, canceling A):

LI-EXHALE-ATOM-CANCEL

$$\frac{A \in \Delta}{\Gamma; \Delta \Vdash \textbf{exhale } A\textbf{; } G \Rightarrow \Gamma; \Delta \setminus \{A\} \Vdash G}$$

However, in the case of EXPR-LOAD', this rule does not apply since the atom $v_1 \mapsto v_2$ depends on the existential v_2.[30]

If LI-EXHALE-ATOM-CANCEL does not apply, Lithium tries to find a *related* proposition A' in the context and turning the **exhale** A into a *subsumption* $A' <: A$.[31] The notion of *related* proposition is determined by the user via a find function D. In our example, we define the propositions related to $v_1 \mapsto v_2$ as the propositions found by $v_1 \mapsto -$, *i.e.*, all points-to predicates for the same memory location. Formally, this procedure is encoded by LI-EXHALE-ATOM-SUBSUME:

LI-EXHALE-ATOM-SUBSUME

$$\frac{\forall \vec{x}.\ A(\vec{x})\ \textsf{related to}\ D \qquad \textbf{find } D\ (x.\ [\![D]\!](x) <: A\textbf{; } G) \textbf{ :- pattern } a.\ A'(a)\textbf{; } G'(a) \qquad A'(b) \in \Delta}{\Gamma; \Delta \Vdash \exists \vec{x}.\ \textbf{exhale } A(\vec{x})\textbf{; } G(\vec{x}) \Rightarrow \Gamma; \Delta \setminus \{A(b)\} \Vdash G'(b)}$$

[30] In general, Lithium is very careful about how existential quantifiers can be instantiated since a wrong instantiation quickly leads to a failed verification. Thus, LI-EXHALE-ATOM-CANCEL does not try to instantiate existentials. §4.2 describes how existentials are instantiated.

[31] We skip over an attempt to simplify the goal, which is described in §3.6.

The subsumption $[\![D]\!](x) <: A$ is part of the continuation passed to the **find** rule. Each find function D comes with an associated atom $[\![D]\!]$ (parametrized by the result of the find function). This atom is used on the left-hand side of the subsumption.[32] For example, we have $[\![v \mapsto -]\!](v_2) \triangleq v \mapsto v_2$.

A subsumption $A_1 <: A_2$ can be seen as a form of subtyping between atoms (*i.e.*, given A_1 prove A_2) and it is handled via user-defined rules. The atom A_2 can depend on the current existentials $\vec{x}$.[33] For our example, we introduce the following subsumption rule:

1: $v \mapsto v_1 <: v \mapsto v_2(\vec{x})\ G$ **:-** $\exists \vec{x}.$ **exhale** $\ulcorner v_1 = v_2(\vec{x})\urcorner$**;** $\mathbf{return}_G\ \vec{x}$

This rule reduces a subsumption between points-to predicates to proving an equality between the values they point to.[34] This equality can then be used to instantiate the existential created by EXPR-LOAD', which then behaves the same as EXPR-LOAD.

[32] This atom is also important for the semantic model of **find** described in §3.8.

[33] We leave the binder for $\vec{x}$ in A_2 implicit.

[34] The rule also places the current existentials $\vec{x}$ in the goal using $\exists$, refers to them in $v_2(\vec{x})$, and makes them available to the continuation G by returning $\vec{x}$.

3.6 Reasoning about Abstract Predicates

One important design principle of Lithium is that all atoms are treated the same way and can be manipulated with the same mechanisms we have seen in §3.5. This means that the expressive power that Lithium provides to manipulate "primitive" atoms like $v_1 \mapsto v_2$ is also available when reasoning about "derived" atoms (*e.g.*, an abstract predicate for a list). This section shows how one can leverage this to automate the reasoning about a linked list predicate.

There are many ways how Lithium can be used to reason about abstract predicates. The approach presented here is a simplified version of the approach used by Islaris (described in §16.3). RefinedC uses a more sophisticated, type-based approach (described in §9.3).

Concretely, this section describes how to use Lithium to automate reasoning about a standard $\mathsf{islist}(v, \overline{v_l})$ predicate:

$$\mathsf{islist}(v, \overline{v_l}) \triangleq (\overline{v_l} = [])\ ?\ \ulcorner v = \#\mathsf{NULL}\urcorner : \exists v'.\ v \mapsto (\mathsf{hd}(\overline{v_l}), v') * \mathsf{islist}(v', \mathsf{tl}(\overline{v_l}))$$

$\mathsf{islist}(v, \overline{v_l})$ asserts that v contains a singly-linked list, which stores the values $\overline{v_l}$.

Modes for islist. Before we can automate reasoning about the islist predicate, we have to think about the *modes* for the arguments of the islist predicate.[35] In particular, we have to decide which arguments are *inputs* and which are *outputs* following the following principle: *Given the values for the inputs, the outputs are uniquely determined.*[36] For example, consider the points-to predicate $v_1 \mapsto v_2$: For a given v_1, there can only be one v_2 since the memory at location v_1 can only store a single value—thus v_1 is an input of the points-to predicate and v_2 an output. We already used this implicitly in the rules we have seen so far: The EXPR-LOAD and EXPR-STORE first determine the input v_1 (of $v_1 \mapsto v_2$) by evaluating a subexpression and then look up the output of v_2 using $v_1 \mapsto -$. Similarly, the subsumption rule for $v_1 \mapsto v_2$ above represents the principle that if the inputs are the same, the outputs should also be equal.[37] The modes

[35] This discussion of modes is inspired by the treatment of modes in VeriFast (Jacobs et al., "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java", 2011 [Jac+11]) and CN (Pulte et al., "CN: Verifying Systems C Code with Separation-Logic Refinement Types", 2023 [Pul+23]).

[36] (Logic) programming languages with backtracking usually allow a finite set of possible outputs for given inputs. However, Lithium requires outputs to be determined uniquely since it does not backtrack.

[37] Also, in the subsumption rule the output v_2 can depend on existentials, but the input v_1 cannot.

$$\begin{aligned}
&\mathsf{fn}\ \mathsf{empty}(_) \triangleq \#\mathsf{NULL}\\
&\mathsf{fn}\ \mathsf{cons}(\mathrm{x}) \triangleq \mathsf{let}\ \mathrm{y} := \mathsf{alloc}\ \mathsf{in}\ \mathrm{y} \leftarrow \mathrm{x};\ \mathrm{y}\\
&\mathsf{fn}\ \mathsf{mklist}(_) \triangleq \mathsf{let}\ \mathrm{x} := \mathsf{empty}\ \#0\ \mathsf{in}\ \mathsf{let}\ \mathrm{x} := \mathsf{cons}\ (\#1, \mathrm{x})\ \mathsf{in}\ \mathsf{let}\ \mathrm{x} := \mathsf{cons}\ (\#2, \mathrm{x})\ \mathsf{in}\ \mathrm{x}\\
&\mathsf{fn}\ \mathsf{head}(\mathrm{x}) \triangleq \mathsf{fst}(!\mathrm{x})\\
&\mathsf{fn}\ \mathsf{length}(\mathrm{x}) \triangleq \mathsf{if}\ \mathrm{x} = \#\mathsf{NULL}\ \mathsf{then}\ \#0\ \mathsf{else}\ \mathsf{let}\ \mathrm{y} := \mathsf{snd}(!\mathrm{x})\ \mathsf{in}\ \mathsf{length}\ \mathrm{y} + \#1
\end{aligned}$$

$$\begin{aligned}
&\mathsf{fnOk}\ \{_\ _.\ \mathsf{True}\}\ \mathsf{fn}\ \mathsf{empty}(_) \triangleq \ldots\ \{_\ v'.\ \mathsf{islist}(v, [])\}\\
&\mathsf{fnOk}\ \{(v_1, \overline{v_l})\ v.\ \exists v_2.\ulcorner v = (v_1, v_2)\urcorner * \mathsf{islist}(v_2, \overline{v_l})\}\ \mathsf{fn}\ \mathsf{cons}(\mathrm{x}) \triangleq \ldots\ \{(v_1, \overline{v_l})\ v'.\ \mathsf{islist}(v', v_1 :: \overline{v_l})\}\\
&\mathsf{fnOk}\ \{_\ _.\ \mathsf{True}\}\ \mathsf{fn}\ \mathsf{mklist}(_) \triangleq \ldots\ \{_\ v'.\ \mathsf{islist}(v', [1, 2])\}\\
&\mathsf{fnOk}\ \{(v_1, \overline{v_l})\ v.\ \ulcorner v = v_1\urcorner * \mathsf{islist}(v, \overline{v_l}) * \ulcorner 0 < |\overline{v_l}|\urcorner\}\ \mathsf{fn}\ \mathsf{head}(\mathrm{x}) \triangleq \ldots\ \{(v_1, \overline{v_l})\ v'.\ \ulcorner v' = \mathsf{hd}(\overline{v_l})\urcorner * \mathsf{islist}(v_1, \overline{v_l})\}\\
&\mathsf{fnOk}\ \{(v_1, \overline{v_l})\ v.\ \ulcorner v = v_1\urcorner * \mathsf{islist}(v, \overline{v_l})\}\ \mathsf{fn}\ \mathsf{length}(\mathrm{x}) \triangleq \ldots\ \{(v_1, \overline{v_l})\ v'.\ \ulcorner v' = \#|\overline{v_l}|\urcorner * \mathsf{islist}(v_1, \overline{v_l})\}
\end{aligned}$$

Figure 3.9: List functions and specifications.

for the $\mathsf{islist}(v, \overline{v_l})$ predicate are similar: If we know v, there can only be one $\overline{v_l}$ as v together with the underlying heap (which is implicit in separation logic) determines $\overline{v_l}$. Thus, v is an input and $\overline{v_l}$ an output. We will use this observation when designing the automation for islist.[38]

One important motivation for following a consistent moding discipline is that correct modes are important to determine the right instantiation for existential quantifiers. One strategy to ensure that Lithium is able to correctly instantiate existential quantifiers, is to follow the following discipline: *Existential variables (*i.e.*, variables introduced by* $\exists$*) must appear in an output position before appearing in an input position.* The reason this strategy works is simple: Since outputs are uniquely determined by their inputs, existential variables that appear in output positions have a clear instantiation.[39]

[38] Unlike VeriFast and CN, Lithium does not provide explicit support for managing modes, but relies on the user to keep the modes in mind when writing rules.

[39] §4.2 goes into more detail of the handling existentials in Lithium and describes more complex strategies for determining their instantiation if this simple strategy fails.

Verification of functions on lists. Figure 3.9 shows the list functions with their specifications that we will verify in this section. We start by considering how we can automatically construct the $\mathsf{islist}(v, \overline{v_l})$ predicate. For this, let us consider empty (for creating an empty list), cons (for adding an element to a list), and mklist (for exercising empty and cons). First, let us consider the case of constructing islist for the empty list. Concretely, this case appears when proving the postcondition for empty where Lithium reaches the following state:

$$\emptyset; \emptyset \vdash \mathbf{exhale}\ \mathsf{islist}(\#\mathsf{NULL}, []); \mathbf{done}$$

To make progress in this state, we define the following rule:

SIMPL-LIST-EMPTY
1: $\mathsf{simplify\_goal}\ \mathsf{islist}(\#\mathsf{NULL}, \overline{v_l})\ G\ \mathbf{:-}\ \mathbf{exhale}\ \ulcorner \overline{v_l} = []\urcorner; \mathbf{return}_G$

This rules states that if Lithium needs to exhale $\mathsf{islist}(\#\mathsf{NULL}, \overline{v_l})$, it suffices to prove that $\overline{v_l} = []$. This transformation is expressed using a simplify_goal rule. simplify_goal rules are applied when neither LI-EXHALE-ATOM-CANCEL nor LI-EXHALE-ATOM-SUBSUME apply.[40] A simplify_goal rule transforms an **exhale** of an atom into a new goal. This is shown by

[40] In the implementation, one can specify if a specific simplify_goal rule should be used before LI-EXHALE-ATOM-SUBSUME or after.

LI-EXHALE-ATOM-SIMPLIFY:[41]

$$\frac{\text{LI-EXHALE-ATOM-SIMPLIFY}\quad \mathsf{simplify\_goal}\ A\ G\ \texttt{:-}\ G'}{\Gamma; \Delta \Vdash \textbf{exhale}\ A\texttt{;}\ G \Rightarrow \Gamma; \Delta \Vdash G'}$$

In summary, SIMPL-LIST-EMPTY encodes the moding disciple for islist discussed above for the NULL case: It applies if the input is determined to be #NULL and then enforces that the output $\overline{v_l}$ has the corresponding value []. Using SIMPL-LIST-EMPTY, Lithium successfully verifies empty.

FIND-LIST
1: **find** $\mathsf{islist}(v, -)\ G$ **:- pattern** $\overline{v_l}$. $\mathsf{islist}(v, \overline{v_l})$**; return**$_G$ $\mathsf{islist}(v, \overline{v_l})$

FIND-LIST-POINTS-TO
1: **find** $\mathsf{islist}(v_1, -)\ G$ **:- pattern** v_2. $v_1 \mapsto v_2$**; return**$_G$ $(v_1 \mapsto v_2)$

SUBSUME-LIST-LIST
1: $\mathsf{islist}(v, \overline{v_l}_1) <: \mathsf{islist}(v, \overline{v_l}_2(\vec{x}))\ G$ **:-** $\exists \vec{x}.$ **exhale** $\ulcorner \overline{v_l}_1 = \overline{v_l}_2(\vec{x}) \urcorner$**; return**$_G$ $\vec{x}$

SUBSUME-POINTS-TO-LIST
1: $v \mapsto v' <: \mathsf{islist}(v, \overline{v_l}(\vec{x}))\ G$ **:-** $\exists \vec{x}.\ \exists v_1.\ \exists v_2.\ \exists \overline{v_l}'.$
2: **exhale** $\ulcorner v' = (v_1, v_2) \urcorner * \mathsf{islist}(v_2, \overline{v_l}') * \ulcorner \overline{v_l}(\vec{x}) = v_1 :: \overline{v_l}' \urcorner$**; return**$_G$ $\vec{x}$

Figure 3.10: Basic Lithium rules for $\mathsf{islist}(v, \overline{v_l})$.

Now, let us consider the case of adding an element to a list. When verifying cons, Lithium reaches the following state (omitting Γ):

$$v \mapsto (v_1, v_2), \mathsf{islist}(v_2, \overline{v_l}) \Vdash \textbf{exhale}\ \mathsf{islist}(v, v_1 :: \overline{v_l})\texttt{;}\ \textbf{done}$$

Here Lithium is stuck, since none of its rules apply. In particular, Lithium does *not* automatically try to unfold the islist predicate.[42] Instead, we, as the users of Lithium, have to tell Lithium how it should handle this case. In this case, we don't know anything about the value v just from looking at the goal. Thus, to get more information about v from the context, we use Lithium's subsumption mechanism.[43] Concretely, we introduce a find function $\mathsf{islist}(v, -)$ that we use to find propositions related to $\mathsf{islist}(v, \_)$.[44] The corresponding Lithium rules are shown in Figure 3.10. FIND-LIST defines that a $\mathsf{islist}(v, \_)$ predicate in the context is related to a $\mathsf{islist}(v, \_)$ predicate in the goal, and SUBSUME-LIST-LIST reduces the resulting subsumption to proving equality of the lists (*i.e.*, the outputs). (This subsumption is used during the verification of mklist.) FIND-LIST-POINTS-TO states that a $v \mapsto \_$ is related to $\mathsf{islist}(v, \_)$ and SUBSUME-POINTS-TO-LIST reduces the resulting subsumption to proving that the $\mathsf{islist}(v, \_)$ predicate is in the cons case. All these rules follow the moding discipline since they use the input v to find a related predicate in the context and then determine the output based on what they find. With these rules, Lithium can automatically verify cons and mklist.[45]

Now, let us turn to automatically destructing the islist predicate. This is necessary to verify the head function. Concretely, when Lithium reaches the !x in head, it has to prove **find** $v \mapsto -$ (from EXPR-LOAD), but FIND-POINTS-TO does not apply since the context contains $\mathsf{islist}(v, \overline{v_l})$ instead of a $v \mapsto \_$ predicate. To solve this problem, we leverage Lithium's extensibility and the fact that the islist predicate can be manipulated

[41] We omit the handling of existentials here. The full rule with existentials is shown in §4.3.

[42] This is a deliberate choice. Deciding when to unfold predicates is tricky, so Lithium aims to behave predictably and lets the user define the heuristics for when to unfold which predicates.

[43] We could add a heuristic to unfold the islist predicate when the list is a cons, but this would unfold islist to eagerly, *e.g.*, during the verification of mklist, and violate our moding discipline which says that we should key the automation on the inputs, not the outputs.

[44] Note that the input v is passed to $\mathsf{islist}(v, -)$, but the output $\overline{v_l}$ is not.

[45] We omit the trivial rule creating the pair in mklist.

BINOP-EQ-NULL
1: binopOk $v = \#\mathsf{NULL}\ G$:- $b \leftarrow$ **find** v null?; **return**$_G$ b

FIND-NULL-POINTS-TO
1: **find** v null? G :- **pattern** v_2. $v \mapsto v_2$; **inhale** $v \mapsto v_2$; **return**$_G$ false

FIND-NULL-LIST
1: **find** v null? G :- **pattern** $\overline{v_l}$. islist$(v, \overline{v_l})$; **inhale** islist$(v, \overline{v_l})$; **return**$_G$ $\overline{v_l} = []$

Figure 3.11: Lithium rules for comparison with NULL.

using the same techniques as the $v_1 \mapsto v_2$ predicate. Concretely, we add a **find** $v \mapsto -$ rule for islist$(v, \overline{v_l})$:

FIND-POINTS-TO-LIST
1: **find** $v \mapsto -$ G :- **pattern** $\overline{v_l}$. islist$(v, \overline{v_l})$;
2: **exhale** $\ulcorner 0 < |\overline{v_l}| \urcorner$;
3: $\forall v_1.\, \forall v_2.\, \forall \overline{v_l}'.$ **inhale** $\ulcorner \overline{v_l} = v_1 :: \overline{v_l}' \urcorner *$ islist$(v_2, \overline{v_l}')$;
4: **return**$_G$ (v_1, v_2)

Intuitively, FIND-POINTS-TO-LIST says that if we want to read from a value that stores a list with values $\overline{v_l}$, the list must be non-empty (*i.e.*, $0 < |\overline{v_l}|$) and then the resulting value will be a pair (v_1, v_2) where v_1 is the head of $\overline{v_l}$ and v_2 is a list containing the tail of $\overline{v_l}$ (following the definition of islist). With FIND-POINTS-TO-LIST, Lithium can automatically verify head.[46]

The example of $v \mapsto -$ shows how Lithium rules can be extended to handle new abstract predicates like islist. Another example of this pattern is shown in Figure 3.11 for comparing a value with NULL (as used by length). BINOP-EQ-NULL uses a new find function v null? to allow overloading of the null check based on the context. This enables specialized rules for custom abstract predicates like islist. FIND-NULL-LIST uses this to express that a value storing a list is NULL exactly if the list is empty. With these rules, Lithium can automatically verify the length function.

[46] We omit the trivial rules for fst(_) and snd(_).

Summary. With the rules from this section Lithium is able to automatically verify all functions in Figure 3.9 without explicit unfold or fold annotations provided by the user as sometimes required by other automated verification tools. Lithium does *not* achieve this by having complex heuristics for abstract predicates built-in. Instead, Lithium relies on the user to provide the heuristics when to unfold abstract predicates like islist. This approach provides significant flexibility to the user and makes the behavior of Lithium simple and predictable.

3.7 Verifying Higher-order Functions

This section demonstrates the flexibility of Lithium by showing how it can be used to automatically verify higher-order functions. In fact, the Lithium primitives we have seen so far are almost enough achieve this task, we just need to introduce two more simple primitives: G_1 **and** G_2 for splitting the verification into the two subcases G_1 and G_2 and **drop_spatial** to remove all spatial (*i.e.*, non-persistent) propositions from the context.

Verifying contains. As a concrete example for a higher-order function, we verify the contains function shown in Figure 3.12. contains takes two

$\forall\phi.\ \{(v_1, \overline{v_l})\ v.\exists\ v_2.\ulcorner v = (v_1, v_2)\urcorner * \mathsf{islist}(v_1, \overline{v_l})$
$\quad * \mathsf{fnOk}\ \{v_1'\ v'.\ \ulcorner v_1' = v'\urcorner * \ulcorner v' \in \overline{v_l}\urcorner\}\ v_2\ \{v_1'\ v'.\ \ulcorner v' = \#(\phi\ v_1')\urcorner\}\}$
$\mathsf{fn}\ \mathsf{contains}(\mathrm{x}) \triangleq \mathsf{let}\ \mathrm{y} := \mathsf{fst}(\mathrm{x})\ \mathsf{in}\ \mathsf{let}\ \mathrm{z} := \mathsf{snd}(\mathrm{x})\ \mathsf{in}$
$\qquad\qquad \mathsf{if}\ \mathrm{y} = \#\mathsf{NULL}\ \mathsf{then}\ \#\mathsf{false}$
$\qquad\qquad \mathsf{else\ if}\ \mathrm{z}\ \mathsf{fst}(!\mathrm{y})\ \mathsf{then}\ \#\mathsf{true}$
$\qquad\qquad \mathsf{else}\ \mathsf{contains}\ (\mathsf{snd}(!\mathrm{y}), \mathrm{z})$
$\{(v_1, \overline{v_l})\ v'.\ \ulcorner v' = \#(\exists v \in \overline{v_l}.\ \phi\ v)\urcorner * \mathsf{islist}(v_1, \overline{v_l})\}$

Figure 3.12: contains function and specification.

arguments: a list (stored in y) and a callback (stored in z). contains then recursively iterates over the list until it finds an element where the callback returns true. If it finds such an element, contains returns true, otherwise false. The pre- and postcondition of contains are shown above and below contains in Figure 3.12. The specification of contains is parametrized by a pure predicate ϕ. The most interesting part of the specification is the fnOk precondition that states that the callback v_2 passed as the second argument returns whether ϕ holds for its argument as a Boolean value. The specification of v_2 uses the $\ulcorner v' \in \overline{v_l}\urcorner$ precondition to state that v_2 is only called on values from $\overline{v_l}$. The postcondition of contains states that it returns a Boolean that reflects whether there exists a $v \in \overline{v_l}$ such that $\phi\ v$ holds.

The challenge when verifying contains is handling the fnOk assumption. Using the assumption to justify the call to the callback is already handled by the standard rule for calls EXPR-APP since the fnOk assumption in the precondition is treated like any other fnOk assumption. However, *proving* the fnOk when performing the recursive call is non-trivial since the specification of the callback changes between the recursive calls. Concretely, Lithium reaches the following state when proving the precondition of the recursive call:

$$\ldots, \mathsf{fnOk}\ \{\ldots * \ulcorner v' \in v_1 :: \overline{v_l}\urcorner\}\ v_2\ \{\ldots\} \Vdash \mathbf{exhale}\ \mathsf{fnOk}\ \{\ldots * \ulcorner v' \in \overline{v_l}\urcorner\}\ v_2\ \{\ldots\}\mathbf{;}\ \ldots$$

The fnOk assertions in the context and goal *almost* match up, except that the list $\overline{v_l}$ becomes smaller during the recursive call. Thus, we cannot use LI-EXHALE-ATOM-CANCEL to make progress. Instead, we introduce a subsumption rule between function specifications (after declaring that fnOk is related to a fnOk for the same value). Luckily, Lithium is expressive enough to express a strong subsumption rule between function specifications: SUBSUME-FN in Figure 3.13.

First, let us introduce the new Lithium primitives used by this rule: The G_1 **and** G_2 primitive splits the verification into the two branches G_1 and G_2. Here, we use this primitive to spawn a subproof for proving the subsumption on line 3 and continue with G in the main branch. At the start of this subproof on line 3, we use the **drop_spatial** instruction to remove all non-persistent propositions from the context. This is necessary, since fnOk is persistent, so we cannot use any non-persistent resources to prove it. Formally, Lithium handles these primitives as follows:

LI-AND
$\Gamma; \Delta \Vdash G_1\ \mathbf{and}\ G_2 \Rightarrow \Gamma; \Delta \Vdash G_1 \,|\, \Gamma; \Delta \Vdash G_2$

LI-DROP-SPATIAL
$\Gamma; \Delta \Vdash \mathbf{drop\_spatial;}\ G \Rightarrow \Gamma; \{\square\, A \mid \square\, A \in \Delta\} \Vdash G$

SUBSUME-FN

```
1: fnOk {pre_1} v {post_1} <: fnOk {pre_2} v {post_2} G :-
2:     do
3:         drop_spatial;
4:         ∀a. ∀v_arg. inhale pre_2 a v_arg;
5:         ∃b. exhale pre_1 b v_arg;
6:         ∀v_ret. inhale post_1 b v_ret;
7:         exhale post_2 a v_ret;
8:         done
9:     and
10:        ∃x⃗. G x⃗
```

SIMPL-FN

```
1: simplify_goal fnOk {pre} fn f(x) ≜ e {post} G :-
2:     do
3:         drop_spatial;
4:         ∀a. ∀v_arg. ∀v_f.
5:         inhale pre a v_arg;
6:         inhale fnOk {pre} v_f {post};
7:         v_ret ← exprOk (e[x ↦ v_arg][f ↦ v_f]);
8:         exhale post a v_ret;
9:         done
10:    and
11:        ∃x⃗. G x⃗
```

Figure 3.13: Rules for fnOk.

The heart of SUBSUME-FN are lines 4-7: To prove fnOk $\{pre_2\}\ v\ \{post_2\}$ from fnOk $\{pre_1\}\ v\ \{post_1\}$, we can first assume the precondition pre_2 and then have to prove the precondition pre_1. Then, we can assume the postcondition pre_1 and have to prove the postcondition pre_2.[47] With SUBSUME-FN, Lithium is able to automatically verify contains.[48]

[47] Note that SUBSUME-FN is a lot stronger than required for this example, *e.g.*, it allows framing ownership from pre_2 to $post_2$.

[48] We also need a strong enough solver for the pure side conditions about lists. All side conditions can be solved with one line of Coq tactics.

Verifying a client. Next, we see how we can use Lithium to verify a client of contains, concretely the following function contains_one, which uses contains to check if a list of integers contains the number 1:

$$\{(v_1, \overline{z})\ v.\ \ulcorner v = v_1 \urcorner * \mathsf{islist}(v, \overline{\#z})\}$$
$$\quad \mathsf{fn}\ \mathsf{contains\_one}(\mathrm{x}) \triangleq \mathsf{contains}\ (\mathrm{x}, (\mathsf{fn}\ \_(\mathrm{y}) \triangleq \mathrm{y} = \#1))$$
$$\{(v_1, \overline{z})\ v'.\ \ulcorner v' = \#(1 \in \overline{z}) \urcorner * \mathsf{islist}(v_1, \overline{\#z})\}$$

To verify contains_one, we use the specification of contains instantiated with $\phi\ v \triangleq v = \#1$.[49] The challenge when verifying contains_one is proving the fnOk precondition for contains. Concretely, Lithium reaches the following state:[50]

[49] Lithium does not try to infer specifications, so we have to manually provide ϕ.

$$\ldots \Vdash \mathbf{exhale}\ \mathsf{fnOk}\ \{v_1'\ v'.\ \ulcorner v_1' = v' \urcorner * \ulcorner v' \in \overline{\#z} \urcorner\}\ \mathsf{fn}\ \_(\mathrm{y}) \triangleq \mathrm{y} = \#1\ \{v_1'\ v'.\ \ulcorner v' = \#(v_1' = \#1) \urcorner\}; \ \ldots$$

But we have already seen in §3.3 that we can prove such a fnOk in Lithium using FNOK! All we need to do is tell Lithium to automatically apply FNOK by rephrasing it as a simplify_goal rule (using **and** and **drop_spatial** in the same way as SUBSUME-FN). This strategy is encoded by SIMPL-FN in Figure 3.13. With SIMPL-FN, Lithium is able to automatically verify contains_one.[51]

[50] Note that the $\ulcorner v' \in \overline{\#z} \urcorner$ precondition is important for this goal to be provable as the language only allows to compare integers to other integers (*e.g.*, comparing a pointer to an integer causes undefined behavior).

[51] We need a strong enough solver for the pure side conditions about lists here as well.

Summary. We have seen how Lithium can be used to automatically verify higher-order functions. It is important to note that Lithium is completely oblivious of the notion of higher-order functions, but its primitives can be composed to build a verification algorithm for higher-order functions. This fact that Lithium does not rely on a built-in notion of function means that it can be reused for different languages with different notions of functions—*e.g.*, for functions with local variables as in RefinedC or for assembly instructions with preconditions as in Islaris.

3.8 Foundational Proofs via a Semantic Model

So far we have seen how Lithium provides automated verification. However, this is only one of the two main features of Lithium discussed in §2. So now let us see how Lithium is able to provide foundational proofs for each successful verification.

To obtain foundational proofs, Lithium takes inspiration from the semantic typing approach:[52] In addition to the operational model described in §3.2, Lithium also comes with a *semantic model* that translates each Lithium construct G to a separation logic formula $[\![G]\!]$. Additionally, each step of Lithium comes with a proof that it maintains provability, *i.e.*, for each Lithium step $\Gamma_1; \Delta_1 \Vdash \exists \vec{x}_1.\, G_1(\vec{x}_1) \Rightarrow \Gamma_2; \Delta_2 \Vdash \exists \vec{x}_2.\, G_2(\vec{x}_2)$ we have a proof of the following:[53]

$$\frac{\Gamma_2; \Delta_2 \vdash \exists \vec{x}_2.\, [\![G_2(\vec{x}_2)]\!]}{\Gamma_1; \Delta_1 \vdash \exists \vec{x}_1.\, [\![G_1(\vec{x}_1)]\!]}$$

These proofs are then transitively composed during the execution of Lithium. When Lithium reaches **done**, it can conclude the proof since we define $[\![\textbf{done}]\!] \triangleq \mathsf{True}$. Overall, we obtain the following soundness theorem for Lithium that shows that we can prove a separation logic assertion $[\![G]\!]$ by running Lithium:[54]

Theorem 1 (Soundness of Lithium)

$$\frac{\Gamma; \Delta \Vdash G \Rightarrow \ldots \Rightarrow \Gamma'; \Delta' \Vdash \textbf{done}}{\Gamma; \Delta \vdash [\![G]\!]}$$

Semantics of goals. Let us now see how $[\![G]\!]$ is defined. Figure 3.14 shows the semantics of most of the Lithium constructs we have seen in this chapter.

$$[\![\textbf{exhale}\ H;\ G]\!] \triangleq H * [\![G]\!] \qquad [\![\forall x.\, G]\!] \triangleq \forall x.\, [\![G]\!] \qquad [\![\textbf{done}]\!] \triangleq \mathsf{True} \qquad [\![x \leftarrow \textbf{find}\ D;\ G]\!] \triangleq \exists x.\, [\![D]\!](x) * [\![G]\!]$$
$$[\![\textbf{inhale}\ H;\ G]\!] \triangleq H \mathrel{-\!\!*} [\![G]\!] \qquad [\![\exists x.\, G]\!] \triangleq \exists x.\, [\![G]\!] \qquad [\![\textbf{return}_G\ x]\!] \triangleq [\![G(x)]\!] \qquad [\![x \leftarrow F;\ G]\!] \triangleq [\![F(x.\, G)]\!]$$

Figure 3.14: (Partial) definition of $[\![G]\!]$.

The semantics of **exhale** is defined as a separating conjunction since **exhale** H requires proving H. Dually, the semantics of **inhale** is defined as a magic wand since **inhale** H adds H as an assumption.[55] The quantifiers of Lithium ($\exists$ and $\forall$) map to the corresponding quantifiers of separation logic. As we have already seen before, **done** is defined as True and thus allows to trivially conclude the verification. As discussed in §3.4, $\textbf{return}_G$ calls the continuation G. The semantics of **find** is defined using the semantics of the find function D.[56] The semantics of user-defined functions F need to be provided by the user.[57]

With this definition of $[\![G]\!]$ in hand, we can already check that many of the Lithium steps we have seen so far are sound. For example, the soundness of LI-EXHALE-ATOM-CANCEL reduces to the following derivable

[52] Milner, "A Theory of Type Polymorphism in Programming", 1978 [Mil78]; Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language", 2018 [Jun+18a].

[53] Note that use of the separation logic entailment $\vdash$ instead of the Lithium entailment $\Vdash$.

[54] In case the execution contains branching, all branches must end with **done**.

[55] This follows the definition of **exhale** and **inhale** of Parkinson and Summers, "The Relationship Between Separation Logic and Implicit Dynamic Frames", 2012 [PS12].

[56] As seen in §3.5, $[\![D]\!]$ is also used when introducing a subsumption.

[57] We will see examples for this shortly.

rule of separation logic:

$$\frac{A \in \Delta \qquad \Gamma; \Delta \setminus \{A\} \vdash [\![G]\!]}{\Gamma; \Delta \vdash A * [\![G]\!]}$$

As another example, li-exhale-star is sound because of the associativity of the separating conjunction:

$$\frac{\Gamma; \Delta \vdash \exists \vec{x}.\, H_1(\vec{x}) * (H_2(\vec{x}) * [\![G(\vec{x})]\!])}{\Gamma; \Delta \vdash \exists \vec{x}.\, (H_1(\vec{x}) * H_2(\vec{x})) * [\![G(\vec{x})]\!]}$$

Similar arguments apply to the other rules in Figure 3.7.

Semantics of functions. Next, let us consider Figure 3.15 which shows the definition of $[\![F]\!]$ for the functions we have seen in this chapter.

$$[\![\mathsf{exprOk}\ e\ G]\!] \triangleq \mathsf{wp}\ e\ \{v.\, [\![G(v)]\!]\} \qquad [\![\mathsf{simplify\_goal}\ A\ G]\!] \triangleq A * [\![G]\!]$$
$$[\![\mathsf{binopOk}\ v_1 \oplus v_2\ G]\!] \triangleq \mathsf{wp}\ v_1 \oplus v_2\ \{v.\, [\![G(v)]\!]\} \qquad [\![A_1 <: A_2\ G]\!] \triangleq A_1 \mathrel{-\!\!*} (\exists \vec{x}.\, A_2(\vec{x}) * [\![G(\vec{x})]\!])$$
$$[\![\mathsf{ifOk}\ v\ G_1\ G_2]\!] \triangleq \exists b.\, \ulcorner v = \#b \urcorner * (b\ ?\ [\![G_1]\!] : [\![G_2]\!])$$

Figure 3.15: Definition of $[\![F]\!]$.

For example, the semantics of exprOk e is defined as the weakest precondition for e (introduced in §2.1).[58] This definition formalizes the intuitive notion that running exprOk e corresponds to verifying e. In particular, coming back to the first example of this chapter, we have

$$[\![\_ \leftarrow \mathsf{exprOk}\ \mathsf{assert\_two}; \mathbf{done}]\!] \triangleq \mathsf{wp}\ \mathsf{assert\_two}\ \{\_.\, \mathsf{True}\}$$

Thus, by Theorem 1 and the fact that Lithium successfully verifies this program, we obtain a proof of wp assert_two $\{\_.\, \mathsf{True}\}$, which we can then combine with the adequacy statement of Iris to obtain a foundational proof that assert_two does not cause undefined behavior, and, in particular, that the assert succeeds. This illustrates how we can use Lithium to automatically obtain foundational proofs of correctness.

Semantics of Lithium rules. One final piece of the puzzle is missing: To justify li-fn, each Lithium rule definition F **:-** G must come with a proof that the semantics of the body must entail the semantics of the function, *i.e.*, $[\![G]\!] \vdash [\![F]\!]$.[59] For example for expr-alloc, this requires showing the following entailment that corresponds to the standard allocation rule of separation logic:

$$\forall v.\, v \mapsto \#0 \mathrel{-\!\!*} [\![G(v)]\!] \vdash \mathsf{wp}\ \mathsf{alloc}\ \{v.[\![G(v)]\!]\}$$

As another example, expr-load requires a proof of the following entailment, which can be proven using the standard separation logic rules for binding a subexpression and verifying a load:

$$\mathsf{wp}\ e\ \{v_1.\exists v_2.\, v_1 \mapsto v_2 * (v_1 \mapsto v_2 \mathrel{-\!\!*} [\![G(v_2)]\!]\} \vdash \mathsf{wp}\ !e\ \{v.[\![G(v)]\!]\}$$

[58] The semantics of the other functions is straightforward, except for the semantics of $A_1 <: A_2$, which is discussed later.

[59] li-find is justified similarly, by requiring rules for find functions **find** D G **:- pattern** a. $A(a)$**;** $G'(a)$ provide a proof of $\exists a.\, A(a) * [\![G'(a)]\!] \vdash \exists x.\, [\![D]\!](x) * [\![G]\!]$.

In some sense, the soundness proofs of the Lithium rules are the "meat" of a Lithium-based verification, as they show that the way Lithium verifies different expressions is sound. The main job of Lithium is to compose these rules and their soundness proofs to construct an overall verification of the program.

Semantics of $A_1 <: A_2$. An interesting detail is the definition of $[\![A_1 <: A_2\ G]\!]$ in Figure 3.15. Ignoring existentials, we have two choices: We could define it as $A_1 \mathrel{-\!\!*} (A_2 * [\![G]\!])$ or $(A_1 \mathrel{-\!\!*} A_2) * [\![G]\!]$. Lithium uses the first choice, but why? The reason is that the first choice allows us to split the ownership contained in A_1 between proving A_2 and the remaining goal G, while the second choice does not.[60] Such a splitting of A_1 is useful when A_1 contains more ownership than what is required to prove A_2. Concretely, consider the following subsumption rule between the islist predicate and the points-to predicate:

1: $\mathsf{islist}(v, \overline{v_l}) <: v \mapsto v'(\vec{x})\ G$ **:-**
2: $\forall v_1.\, \forall v_2.\, \forall \overline{v_l}'.$ **inhale** $\ulcorner \overline{v_l} = v_1 :: \overline{v_l}' \urcorner * \mathsf{islist}(v_2, \overline{v_l}')$**;**
3: $\exists \vec{x}.$ **exhale** $\ulcorner v'(\vec{x}) = (v_1, v_2) \urcorner$**;** **return**$_G$ $\vec{x}$

This rule is only sound with the first semantics of $A_1 <: A_2$ (used by Lithium), as the **inhale** makes the ownership of the remaining list available to the goal G.

[60] Also, with the second choice it would not be clear where to place the existential quantifier for the Lithium existentials.

Chapter 4

Lithium in Detail

After seeing how Lithium works in §3, this chapter discusses two more aspects of Lithium—how Lithium avoids backtracking (§4.1) and how Lithium handles existentials (§4.2)—before describing the complete definition of Lithium in §4.3.

4.1 Avoiding Backtracking

The Lithium proof search procedure is efficient in large part because it does not backtrack. Several design choices make this possible.

First, the argument of **exhale** is limited to the form H, which cannot contain $\wedge$, $\forall$, and $-\!\!*$. Without this restriction, handling a **exhale** G_1**;** G_2 would require a two-way split of the resource context Δ to prove G_1 and G_2 simultaneously, requiring backtracking over possible splits of Δ. However, when G_1 is limited to the form H, we can reduce it in place all the way down to atoms (LI-EXHALE-EXIST and LI-EXHALE-STAR), which eliminates this form of backtracking.

Second, the argument of **inhale** in goals is also restricted to the form H. This allows us to reduce local assumptions to atoms before adding them to the context Δ (LI-INHALE-EXIST and LI-INHALE-STAR). By relying on the fact that Δ only contains atoms, we can design **find** rules such that only one of them applies at a time. For example, consider FIND-LIST and FIND-LIST-POINTS-TO: Only one of these rules can fire at a time since Δ can only contain one of $\mathsf{islist}(v, \_)$ or $v \mapsto \_$, but never both.[1]

Third, Lithium relies on the user to ensure that existentials have a unique instantiation (*e.g.*, via the moding discipline described in §3.6 or via the more advanced mechanisms presented in §4.2). Thanks to this assumption, Lithium can avoid backtracking on the instantiation of existentials.

In principle, the need for backtracking *could* arise in LI-FN when multiple Lithium rules apply for a given function. However, Lithium provides the necessary tools to write rules such that there is always enough information to uniquely determine which rule to apply. For example, consider BINOP-EQ-NULL: Since there is not enough information how to handle a comparison with NULL by just looking at the expression, we introduce a new find function v null? that can match on the context to determine how to proceed.[2]

[1] RefinedC ensures this property by using type assignments as atoms and ensure that each value has at most one type assignment at a time.

[2] Lithium also offers a way to specify priority among Lithium rules. But once a rule is chosen, Lithium does not backtrack on the choice.

4.2 Handling of Existentials

One important aspect of Lithium that we have not described so far is the handling of existentials created in li-exist. In particular, Lithium must be careful when instantiating existentials because a bad instantiation could easily make the goal unprovable. To prevent this, most parts of Lithium treat existentials as opaque variables. In fact, the only place existentials can get instantiated is when solving side conditions emitted by li-exhale-pure, at which point Lithium attempts to eliminate any existentials in the side condition using one of the following heuristics.

First, Lithium tries to find a suitable instantiation for the existentials by checking if the side condition is an equality and, if so, trying to unify both sides (potentially instantiating existentials). Though this heuristic is often effective, it may also turn a provable goal into an unprovable goal if it unifies an existential appearing as the argument of a non-injective symbol. For example, unifying $|x|$ and $|l|$ where x is an existential will lead to x being instantiated with l, whereas the correct instantiation for x might in fact be another list with the same length as l. In such cases, the user's only recourse at present is to adjust the specifications and Lithium program to generate side conditions in an order that allows correct instantiation. However, in our experience, this rarely causes problems. In particular, all of RefinedC and Islaris uses this heuristic.

Second, if Lithium cannot instantiate the existentials in the side condition, it simplifies the goal using a set of user-extensible rewriting rules and equivalences. For example, a side condition of the form $xs \neq []$ where xs is an existential is simplified to the equivalent $\exists y.\, \exists ys.\, xs = y :: ys$, which leads Lithium to turn y and ys into Lithium existentials and instantiate xs with $y :: ys$. The simplification rules are also used by li-inhale-pure to normalize assumptions introduced into the context. For example, an assumption $xs +\!\!+ ys = []$ is simplified to $xs = []$ and $ys = []$, which causes both xs and ys to be substituted with $[]$. By default, this simplification mechanism applies equivalences and thus preserves provability, but there is an escape hatch that lets one add implications (rather than equivalences) as simplification rules. (Doing so can make provable goals unprovable.)

The procedure described above is not complete as there can be a side condition for which the heuristic for existential instantiation fails and no simplification rule applies. However, the predictable nature of Lithium helps the user avoid such side conditions: since instructions in Lithium are sequenced, it is straightforward to predict in which order the side conditions will be generated. For example, when checking the precondition of a fnOk, the side conditions are generated in a left-to-right order (following li-exhale-star), so it is easy to manually order them to make sure that the existentials are instantiated correctly. As a concrete example, consider the precondition of cons (Figure 3.9 in §3.6):

$$\mathsf{fnOk}\ \{(v_1, \overline{v_l})\ v.\ \exists v_2.\, \ulcorner v = (v_1, v_2) \urcorner * \mathsf{islist}(v_2, \overline{v_l})\}\ \mathsf{fn}\ \mathsf{cons}(\mathrm{x}) \triangleq \ldots\ \{\ldots\}$$

With this phrasing of the precondition, the existential created for v_2 is trivially instantiated by the equality (assuming that cons is called with a pair for the argument v). After v_2 is instantiated, one can continue

with proving $\mathsf{islist}(v_2, \overline{v_l})$ (and instantiating $\overline{v_l}$). However, if we reorder the preconditions to the following:

$\mathsf{fnOk}\ \{(v_1, \overline{v_l})\ v.\ \exists v_2.\ \mathsf{islist}(v_2, \overline{v_l}) * \ulcorner v = (v_1, v_2)\urcorner\}\ \mathsf{fn}\ \mathsf{cons(x)} \triangleq \ldots\ \{\ldots\}$

Lithium gets stuck on the goal $\exists v_2, \overline{v_l}.\ \mathbf{exhale}\ \mathsf{islist}(v_2, \overline{v_l});\ \ldots$

4.3 Complete Definition of Lithium

$$
\begin{array}{lll}
\text{Atom} & A ::= \ldots & \\
\text{Function} & F ::= A_1 <: A_2 \mid \ldots & \\
\text{Goal} & G ::= \mathbf{exhale}\ H;\ G \mid \mathbf{inhale}\ H;\ G \mid \forall x.\ G \mid \exists x.\ G \mid \mathbf{done} \mid \mathbf{false} \mid & \\
 & \quad x \leftarrow F;\ G \mid x \leftarrow \mathbf{find}\ D;\ G \mid \mathbf{return}_G\ x \mid \mathbf{if}\ \phi\ \mathbf{then}\ G_1\ \mathbf{else}\ G_2 \mid & \\
 & \quad G_1\ \mathbf{and}\ G_2 \mid k,\ v \leftarrow \mathbf{and\_map}\ m;\ G \mid \mathbf{drop\_spatial};\ G \mid x' \leftarrow \mathbf{destruct}\ x;\ G \mid & \\
 & \quad x \leftarrow \mathbf{tactic}\ t;\ G \mid x \leftarrow \mathbf{accu};\ G \mid \mathbf{trace}\ a;\ G \mid x \leftarrow \mathbf{iter}\ l\ \mathbf{with}\ a\ \{G_1\};\ G_2 & \\
\text{Left-goal} & H ::= A \mid \ulcorner \phi \urcorner \mid H * H \mid \exists x.\ H(x) \mid \square H & \\
\text{Contexts} & \Gamma ::= \emptyset \mid \Gamma, x \mid \Gamma, \phi \qquad \Delta ::= \emptyset \mid \Delta, A \mid \Delta, \square A &
\end{array}
$$

Figure 4.1: Syntax of Lithium.

$$
\begin{array}{lll}
[\![\mathbf{exhale}\ H;\ G]\!] \triangleq H * [\![G]\!] & [\![\mathbf{inhale}\ H;\ G]\!] \triangleq H \mathrel{-\!\!*} [\![G]\!] & [\![\forall x.\ G]\!] \triangleq \forall x.\ [\![G]\!] \\
[\![\mathbf{done}]\!] \triangleq \mathsf{True} & [\![\mathbf{false}]\!] \triangleq \mathsf{False} & [\![\exists x.\ G]\!] \triangleq \exists x.\ [\![G]\!] \\
[\![x \leftarrow F;\ G]\!] \triangleq [\![F(x.\ G)]\!] & [\![x \leftarrow \mathbf{find}\ D;\ G]\!] \triangleq \exists x.\ [\![D]\!](x) * [\![G]\!] & [\![\mathbf{return}_G\ x]\!] \triangleq [\![G(x)]\!] \\
[\![G_1\ \mathbf{and}\ G_2]\!] \triangleq [\![G_1]\!] \wedge [\![G_2]\!] & [\![\mathbf{drop\_spatial};\ G]\!] \triangleq \square [\![G]\!] & [\![x' \leftarrow \mathbf{destruct}\ x;\ G]\!] \triangleq [\![G(x)]\!] \\
[\![x \leftarrow \mathbf{tactic}\ t;\ G]\!] \triangleq t\ [\![G]\!] & [\![x \leftarrow \mathbf{accu};\ G]\!] \triangleq \exists P.\ P * [\![G(P)]\!] & [\![\mathbf{trace}\ a;\ G]\!] \triangleq [\![G]\!]
\end{array}
$$

$$
[\![\mathbf{if}\ \phi\ \mathbf{then}\ G_1\ \mathbf{else}\ G_2]\!] \triangleq (\ulcorner \phi \urcorner \mathrel{-\!\!*} [\![G_1]\!]) \wedge (\ulcorner \neg \phi \urcorner \mathrel{-\!\!*} [\![G_2]\!])
$$

$$
[\![x \leftarrow \mathbf{iter}\ b :: l\ \mathbf{with}\ a\ \{G_1\};\ G_2]\!] \triangleq [\![a' \leftarrow G_1\ b\ a;\ \ x \leftarrow \mathbf{iter}\ l\ \mathbf{with}\ a'\ \{G_1\};\ G_2]\!]
$$

$$
[\![x \leftarrow \mathbf{iter}\ []\ \mathbf{with}\ a\ \{G_1\};\ G_2]\!] \triangleq [\![G_2(a)]\!]
$$

$$
[\![k,\ v \leftarrow \mathbf{and\_map}\ m;\ G]\!] \triangleq \bigwedge_{k,v \in m} [\![G(k, v)]\!]
$$

Figure 4.2: Full definition of $[\![G]\!]$.

Figure 4.1 shows the complete syntax of Lithium. The semantic model of Lithium is depicted in Figure 4.2, while the full operational semantics of Lithium can be found in Figure 4.3 and Figure 4.4.[3] When multiple stepping rules apply to the same goal, Lithium picks the first such rule (in left to right, top to bottom order). Let us now discuss the Lithium primitives that were not explained in §3.

Exhaling and inhaling persistent left-goals. §3.5 did not discuss how $\square H$ is handled by **exhale** and **inhale**—we remedy this now. The handling of $\square H$ in **inhale** is straightforward: Lithium splits up H using LI-INHALE-PERSISTENT-PURE, LI-INHALE-PERSISTENT-STAR, and LI-INHALE-PERSISTENT-EXIST and introduces atoms into the context using LI-INHALE-PERSISTENT-ATOM. **exhale** $\square H$ is covered by LI-EXHALE-PERSISTENT, which creates a separate subgoal for proving H after dropping all non-persistent assertions. This step is sound since $\square P * Q$ is equivalent to $\square P \wedge Q$ in separation logic.

[3] Lithium also implicitly performs meta-level reduction (using Coq's `simpl` tactic), so Lithium programs can also contain meta-level constructs like matches as long as they are reduced away when the interpreter reaches them. We will see a use of this feature in §14.3.

LI-EXHALE-ATOM-CANCEL

$$\frac{A \in \Delta}{\Gamma; \Delta \Vdash \textbf{exhale}\ A; G \Rightarrow \Gamma; \Delta \setminus \{A\} \Vdash G}$$

LI-EXHALE-ATOM-SUBSUME

$$\frac{\forall \vec{x}.\ A(\vec{x})\ \text{related to}\ D \qquad \textbf{find}\ D\ (x.\ [\![D]\!](x) <: A; G)\ \textbf{:- pattern}\ a.\ A'; (a); G'(a) \qquad A'(b) \in \Delta}{\Gamma; \Delta \Vdash \exists \vec{x}.\ \textbf{exhale}\ A(\vec{x}); G(\vec{x}) \Rightarrow \Gamma; \Delta \setminus \{A(b)\} \Vdash G'(b)}$$

LI-EXHALE-ATOM-SIMPLIFY

$$\frac{\forall \vec{x}.\ \mathsf{simplify\_goal}\ A(\vec{x})\ G(\vec{x})\ \textbf{:-}\ G'(\vec{x})}{\Gamma; \Delta \Vdash \exists \vec{x}.\ \textbf{exhale}\ A(\vec{x}); G(\vec{x}) \Rightarrow \Gamma; \Delta \Vdash \exists \vec{x}.\ G'(\vec{x})}$$

LI-EXHALE-PURE

$$\frac{\Gamma \vdash \phi}{\Gamma; \Delta \Vdash \textbf{exhale}\ \ulcorner \phi \urcorner; G \Rightarrow \Gamma; \Delta \Vdash G}$$

LI-EXHALE-STAR

$$\Gamma; \Delta \Vdash \exists \vec{x}.\ \textbf{exhale}\ H_1(\vec{x}) * H_2(\vec{x}); G(\vec{x}) \Rightarrow \Gamma; \Delta \Vdash \exists \vec{x}.\ \textbf{exhale}\ H_1(\vec{x}); \textbf{exhale}\ H_2(\vec{x}); G\ (\vec{x})$$

LI-EXHALE-EXIST

$$\Gamma; \Delta \Vdash \exists \vec{x}.\ \textbf{exhale}\ \exists x.\ H(x, \vec{x}); G(\vec{x}) \Rightarrow \Gamma; \Delta \Vdash \exists \vec{x}.\ \exists x.\ \textbf{exhale}\ H(x, \vec{x}); G(\vec{x})$$

LI-EXHALE-PERSISTENT

$$\Gamma; \Delta \Vdash \textbf{exhale}\ \square H; G \Rightarrow \Gamma; \Delta \Vdash (\textbf{drop\_spatial; exhale}\ H; \textbf{done})\ \textbf{and}\ G$$

LI-INHALE-ATOM

$$\Gamma; \Delta \Vdash \textbf{inhale}\ A; G \Rightarrow \Gamma; \Delta, A \Vdash G$$

LI-INHALE-PURE

$$\Gamma; \Delta \Vdash \textbf{inhale}\ \ulcorner \phi \urcorner; G \Rightarrow \Gamma, \phi; \Delta \Vdash G$$

LI-INHALE-STAR

$$\Gamma; \Delta \Vdash \textbf{inhale}\ H_1 * H_2; G \Rightarrow \Gamma; \Delta \Vdash \textbf{inhale}\ H_1; \textbf{inhale}\ H_2; G$$

LI-INHALE-EXIST

$$\Gamma; \Delta \Vdash \textbf{inhale}\ \exists x.\ H(x); G \Rightarrow \Gamma; \Delta \Vdash \forall x.\ \textbf{inhale}\ H(x); G$$

LI-INHALE-PERSISTENT-ATOM

$$\Gamma; \Delta \Vdash \textbf{inhale}\ \square A; G \Rightarrow \Gamma; \Delta, \square A \Vdash G$$

LI-INHALE-PERSISTENT-PURE

$$\Gamma; \Delta \Vdash \textbf{inhale}\ \square\ \ulcorner \phi \urcorner; G \Rightarrow \Gamma, \phi; \Delta \Vdash G$$

LI-INHALE-PERSISTENT-STAR

$$\Gamma; \Delta \Vdash \textbf{inhale}\ \square(H_1 * H_2); G \Rightarrow \Gamma; \Delta \Vdash \textbf{inhale}\ \square H_1; \textbf{inhale}\ \square H_2; G$$

LI-INHALE-PERSISTENT-EXIST

$$\Gamma; \Delta \Vdash \textbf{inhale}\ \square(\exists x.\ H(x)); G \Rightarrow \Gamma; \Delta \Vdash \forall x.\ \textbf{inhale}\ \square H(x); G$$

LI-ALL

$$\Gamma; \Delta \Vdash \forall x.\ G(x) \Rightarrow \Gamma, x; \Delta \Vdash G(x)$$

LI-EXIST

$$\Gamma; \Delta \Vdash \exists \vec{x}.\ \exists x.\ G(x, \vec{x}) \Rightarrow \Gamma; \Delta \Vdash \exists x, \vec{x}.\ G(x, \vec{x})$$

LI-FN-UNFOLD

$$\Gamma; \Delta \Vdash x \leftarrow F; G \Rightarrow \Gamma; \Delta \Vdash F\ (x.\ G)$$

LI-FN'

$$\frac{F\ \textbf{:-}\ G}{\Gamma; \Delta \Vdash F \Rightarrow \Gamma; \Delta \Vdash G}$$

LI-FIND

$$\frac{\textbf{find}\ D\ G\ \textbf{:- pattern}\ a.\ A(a); G'(a) \qquad A(b) \in \Delta}{\Gamma; \Delta \Vdash x \leftarrow \textbf{find}\ D; G \Rightarrow \Gamma; \Delta \setminus \{A(b)\} \Vdash G'(b)}$$

LI-RETURN

$$\Gamma; \Delta \Vdash \textbf{return}_G\ x \Rightarrow \Gamma; \Delta \Vdash G(x)$$

LI-IF-TRUE

$$\frac{\Gamma \vdash \phi}{\Gamma; \Delta \Vdash \textbf{if}\ \phi\ \textbf{then}\ G_1\ \textbf{else}\ G_2 \Rightarrow \Gamma; \Delta \Vdash G_1}$$

LI-IF-FALSE

$$\frac{\Gamma \vdash \neg\phi}{\Gamma; \Delta \Vdash \textbf{if}\ \phi\ \textbf{then}\ G_1\ \textbf{else}\ G_2 \Rightarrow \Gamma; \Delta \Vdash G_2}$$

LI-IF

$$\Gamma; \Delta \Vdash \textbf{if}\ \phi\ \textbf{then}\ G_1\ \textbf{else}\ G_2 \Rightarrow \Gamma, \phi; \Delta \Vdash G_1 | \Gamma, \neg\phi; \Delta \Vdash G_2$$

Figure 4.3: Operational semantics of Lithium, part 1.

LI-FALSE
$$\frac{\Gamma \vdash \mathsf{False}}{\Gamma; \Delta \Vdash \textbf{false} \Rightarrow \Gamma; \Delta \Vdash \textbf{done}}$$

LI-AND
$$\Gamma; \Delta \Vdash G_1 \ \textbf{and}\ G_2 \Rightarrow \Gamma; \Delta \Vdash G_1 \,|\, \Gamma; \Delta \Vdash G_2$$

LI-AND-MAP-INSERT
$$\Gamma; \Delta \Vdash k,\ v \leftarrow \textbf{and\_map}\ m[k' \mapsto v']\textbf{;}\ G \Rightarrow \Gamma; \Delta \Vdash G(k', v') \,|\, \Gamma; \Delta \Vdash k,\ v \leftarrow \textbf{and\_map}\ m\textbf{;}\ G$$

LI-AND-MAP-EMPTY
$$\Gamma; \Delta \Vdash k,\ v \leftarrow \textbf{and\_map}\ \emptyset\textbf{;}\ G \Rightarrow \Gamma; \Delta \Vdash \textbf{done}$$

LI-DROP-SPATIAL
$$\Gamma; \Delta \Vdash \textbf{drop\_spatial;}\ G \Rightarrow \Gamma; \{\Box\, A \mid \Box\, A \in \Delta\} \Vdash G$$

LI-DESTRUCT
$$\frac{\text{destruct } x \text{ into } x'}{\Gamma; \Delta \Vdash x' \leftarrow \textbf{destruct}\ x\textbf{;}\ G \Rightarrow \Gamma; \Delta \Vdash G(x')}$$

LI-TACTIC
$$\frac{\text{run } t \text{ with } G \text{ resulting in } G'}{\Gamma; \Delta \Vdash x \leftarrow \textbf{tactic}\ t\textbf{;}\ G \Rightarrow \Gamma; \Delta \Vdash G'}$$

LI-ACCU
$$\Gamma; \Delta \Vdash x \leftarrow \textbf{accu;}\ G \Rightarrow \Gamma; \{\Box\, A \mid \Box\, A \in \Delta\} \Vdash G(\mathop{\Large *}_{A \in \Delta} A)$$

LI-TRACE
$$\frac{\text{invoke tracing hook with } a}{\Gamma; \Delta \Vdash \textbf{trace}\ a\textbf{;}\ G \Rightarrow \Gamma; \Delta \Vdash G}$$

LI-ITERATE-CONS
$$\Gamma; \Delta \Vdash x \leftarrow \textbf{iter}\ b :: l\ \textbf{with}\ a\ \{G_1\}\textbf{;}\ G_2 \Rightarrow \Gamma; \Delta \Vdash a' \leftarrow G_1\ b\ a\textbf{;}\ \ x \leftarrow \textbf{iter}\ l\ \textbf{with}\ a'\ \{G_1\}\textbf{;}\ G_2$$

LI-ITERATE-NIL
$$\Gamma; \Delta \Vdash x \leftarrow \textbf{iter}\ []\ \textbf{with}\ a\ \{G_1\}\textbf{;}\ G_2 \Rightarrow \Gamma; \Delta \Vdash G_2(a)$$

Figure 4.4: Operational semantics of Lithium, part 2.

The **false** *primitive.* The **false** primitive can be used to mark branches of the verification where further steps are impossible. In this case, Lithium tries to conclude the verification by asking the solver to prove a contradiction from the assumptions in the pure context Γ (LI-FALSE).

The **and_map** *primitive.* The $k,\ v \leftarrow \textbf{and\_map}\ m\textbf{;}\ G$ primitive is similar to **and**, except that it spawns a new goal for each key k and value v in the map m (LI-AND-MAP-INSERT and LI-AND-MAP-EMPTY). The map m must be a concrete map, *i.e.*, syntactically of the form $(\emptyset[k_n \mapsto v_n]) \ldots)[k_1 \mapsto v_1]$. **and_map** is used by RefinedC to create a new case for each branch of a C switch statement.

The **destruct** *primitive.* The $x' \leftarrow \textbf{destruct}\ x\textbf{;}\ G$ primitive destructs the Coq expression x, similar to the Coq `destruct` tactic, *i.e.*, it generates a separate goal for each constructor that could be used to create x. x' is filled with the constructor corresponding to the goal.

The **tactic** *primitive.* The $x \leftarrow \textbf{tactic}\ t\textbf{;}\ G$ primitive allows the user to invoke an arbitrary Coq tactic described by t. Concretely, Lithium uses Coq's typeclass mechanism to find a tactic registered for t and then uses the tactic to generate the next Lithium goal (together with a proof that the Lithium step is sound). For example, Islaris uses **tactic** to normalize arithmetic operations that arise from instructions that access memory.

The **accu** *primitive.* The $x \leftarrow \textbf{accu;}\ G$ primitive removes all non-persistent atoms A from Δ and instantiates x with a big separating conjunction

of these atoms (li-accu). (Persistent atoms $\square A$ remain in Δ.) This behavior is similar to the Iris `iAccu` tactic. This primitive is used by RefinedC to extend a loop invariant with all atoms that are left-over after establishing the user-annotated loop invariant.[4]

[4] This is an experimental feature of RefinedC that is not enabled by default, but needs to be activated via a special annotation.

The **trace** *primitive.* The **trace** a**;** G primitive invokes a user-defined tracing hook with the argument a. RefinedC uses **trace** to track the case distinctions that have been performed in the proof so far for debugging purposes.

The **iter** *primitive.* The $x \leftarrow$ **iter** l **with** a $\{G_1\}$**;** G_2 primitive iterates over l and runs G_1 for each element of l (li-iterate-cons). a is an iteration variable that can be updated by each iteration. The result x of **iter** is the final value of a (li-iterate-nil).[5] The list l must be a concrete list, *i.e.*, syntactically of the form $[a_1, \ldots, a_n]$. RefinedC uses **iter**, for example, to iterate over all fields of a C struct or all arguments to a function.

[5] The implementation of Lithium does not directly implement li-iterate-cons and li-iterate-nil, but instead uses Coq's reduction mechanism to reduce **iter**.

Chapter 5

Related Work

VeriFast. The most closely related work to Lithium is probably the symbolic execution approach of the VeriFast verifier,[1] described by Vogels et al.[2] They define a symbolic execution function that maps constructs of the programming language (annotated with pre- and postconditions and loop invariants) to a language of "outcomes". These outcomes allow demonic and angelic choice (corresponding to ∀ and ∃ in Lithium) and are defined over a notion of state containing separation logic propositions (called heap chunks), which Vogels et al. use to define "produce" and "consume" functions (corresponding to **inhale** and **exhale**). These primitives are then used to define the symbolic execution of the constructs of the language, reminiscent of the definition of Lithium functions like exprOk in §3. However, while there are some similarities, the fundamental goals of the approach that Vogels et al. describe and Lithium are quite different: The first one is a description of the inner workings of one particular verifier—VeriFast—, while the second aims to be a domain-specific language for writing multiple, different verifiers. This shows for example in the fact that the symbolic execution of Vogels et al. is specific to the underlying language, while Lithium is independent of it. (For example, the produce and consume operations are specific to the notion of heap chunks.) Also, while Vogels et al.[3] provide a formalization of their description of the core VeriFast approach that is foundationally sound, the actual implementation of VeriFast does not produce foundational proofs of correctness (unlike Lithium). Another point of difference is that the symbolic execution only supports angelic choice over the empty set (corresponding to **false** in Lithium), but not its general form (∃ in Lithium). Instead, they tie existential quantifiers directly to output positions of predicates to make sure that they can always be instantiated—this avoids the complications described in §4.2, at the cost of flexibility.

Viper. Viper[4] is an intermediate language for developing verification tools. Viper is a simple imperative language with a heap and provides a native notion of separation logic[5] with specifications for functions (and loops) and custom statements for manipulating separation logic assertions.[6] Viper comes with powerful SMT-based automation for verifying these specifications.[7] While Viper currently does not generate foundational proofs (unlike Lithium), there is ongoing work to change this.[8]

To implement a verification tool using Viper, one builds a frontend to translate the object language (usually extended with a syntax for specifica-

[1] Jacobs et al., "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java", 2011 [Jac+11].

[2] Vogels et al., "Featherweight VeriFast", 2015 [VJP15].

[3] Vogels et al., "Featherweight VeriFast", 2015 [VJP15].

[4] Müller et al., "Viper: A Verification Infrastructure for Permission-Based Reasoning", 2016 [MSS16b].

[5] Technically, Viper uses a variant of separation logic called *implicit dynamic frames*, see Parkinson and Summers, "The Relationship Between Separation Logic and Implicit Dynamic Frames", 2012 [PS12].

[6] In particular, Viper provides exhale and inhale statements similar to Lithium.

[7] Heule et al., "Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions", 2013 [Heu+13]; Schwerhoff, "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution", 2016 [Sch16].

[8] Parthasarathy et al., "Formally Validating a Practical Verification Condition Generator", 2021 [PMS21].

tions) to the Viper intermediate language. Viper has many such frontends, for example, for Go,[9] Python,[10] or Rust.[11] Lithium follows a different approach: instead of providing one concrete intermediate language and relying on a frontend to translate programs into this intermediate language, Lithium provides a language for writing verification tools that work directly on the object language (like the example language from §3). As a consequence, Lithium is able to support languages with very different memory object models,[12] while verification using Viper has mostly focused on languages with similar (abstract) memory object models. Another benefit of Lithium's direct approach is that a Lithium rule can both look at the current expression of the object language and the separation logic state to decide how to proceed with the verification, while in Viper's approach the frontend only sees the code of the object language, but not the separation logic state, and vice versa for the underlying proof automation of the Viper language.[13] In particular, this means that Lithium can support rules to automatically reason about abstract predicates (see §3.6, *e.g.*, FIND-POINTS-TO-LIST), while Viper relies on explicit fold and unfold statements (which can sometimes be automatically inferred by the frontend[14]). In the future, it would be interesting to see if some of Viper's automation, in particular, its support for iterated conjunction[15] or for reasoning about fractional permissions,[16] could also be implemented in Lithium.

Diaframe. Diaframe[17] takes inspiration from Lithium to enable automated verification of concurrent programs using Iris. Diaframe automates reasoning about Iris entailments involving Iris concurrency primitives (*e.g.*, view shifts, invariants, and custom ghost state) and combines this with an automated application of weakest-precondition rules. Instead, Lithium focuses on providing a foundation for building verification tools for detailed models of real-world languages. In particular, Diaframe aims to integrate with verification (of concurrent algorithms) as it is usually conducted in Iris, while Lithium is targeted at building larger verification tools like RefinedC or Islaris and thus expects more upfront definitions (*e.g.*, in the form of Lithium functions or atoms). This difference shows in ways these tools can be extended by the user: Diaframe's hint format has rich support for Iris primitives like view shifts or invariants and supports universal quantifiers and magic wands in the context, while Lithium requires the user to hide such connectives inside atoms (like the fnOk atom), but then gives the user full control on how these atoms are manipulated. In particular, rules for Lithium functions like subsumption can use the full syntax of Lithium including arbitrary nesting of **exhale** and **inhale**, or can spawn new subgoals using **and** as shown by SUBSUME-FN, while Diaframe's hint can manipulate the goal in a more restricted way (roughly corresponding to **exhale** followed by **inhale**).[18]

Katamaran. Keuchel et al.[19] present an alternative approach to building an automated and foundational verification tool and implement it in the Katamaran verifier. Their approach avoids the use of meta-programming by defining the verifier as a (Gallina) function that works on a deeply

[9] Wolf et al., "Gobra: Modular Specification and Verification of Go Programs", 2021 [Wol+21].

[10] Eilers and Müller, "Nagini: A Static Verifier for Python", 2018 [EM18].

[11] Astrauskas et al., "Leveraging Rust Types for Modular Specification and Verification", 2019 [Ast+19].

[12] For example, a flat memory model for assembly or a C memory model with both structured and byte-level views of memory.

[13] A Viper frontend can inspect the separation logic state *indirectly* via Viper's permission introspection feature that allows the generated Viper code to branch based on the current separation state, but the two stages remain separate.

[14] Astrauskas et al., "Leveraging Rust Types for Modular Specification and Verification", 2019 [Ast+19].

[15] Müller et al., "Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution", 2016 [MSS16a].

[16] Dardinier et al., "Fractional Resources in Unbounded Separation Logic", 2022 [DMS22].

[17] Mulder et al., "Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris", 2022 [MKG22].

[18] Using **inhale** before **exhale** is for example useful when reasoning about subsumptions between arrays in RefinedC.

[19] Keuchel et al., "Verified Symbolic Execution with Kripke Specification Monads (and No Meta-programming)", 2022 [Keu+22].

embedded specification language. The advantage of this approach is that it gives good performance by leveraging reduction of the proof assistant (Coq) and that the verifier can be extracted to OCaml. However, the downside is that one cannot reuse Coq's rich infrastructure for context management, reduction, typeclasses, or solving pure goals (in particular, the solver for linear integer arithmetic implemented by the `lia` tactic). As a consequence, it is unclear how this approach can support some features of Lithium like its extensibility or the **destruct** primitive.

Gillian. Gillian[20] is a platform for developing symbolic analysis tools, including symbolic testing, (over-approximate) verification, and (under-approximate) bi-abduction. Gillian's verification, which is the aspect of Gillian most-closely related to Lithium, has been applied to complex case studies in multiple languages including C and JavaScript. Verification in Gillian is based to a core language called GIL with a language-specific memory model that describes the core predicates of the logic (similar to Lithium atoms), each of which comes with a produce (*i.e.*, **inhale**) and consume (*i.e.*, **exhale**) action. These core predicates are then lifted to an assertion language that allows the user to define composed predicates. This approach allows Gillian to analyze user-defined predicates, *e.g.*, for generating a matching plan for determining the outputs of the predicate from the inputs. However, since Gillian's core predicates are fixed upfront, it does not have Lithium's ability to add new atoms and custom proof search procedures for them (see *e.g.*, SUBSUME-FN). Additionally, unlike Lithium, Gillian does not produce foundational proofs.

Separation logic automation. The verification literature abounds in (non-foundational) automatic solvers for separation logic and frame inference.[21] These solvers are usually specialized for a certain class of atomic formulas (usually a variant of the symbolic heap fragment[22] of separation logic), rely on more sophisticated automation (*e.g.*, based on SMT solvers), and can automate more difficult reasoning patterns (*e.g.*, induction reasoning[23]) than Lithium. In contrast, proof search in Lithium is conceptually more straightforward (which makes it more predictable and amenable to implementation in a proof assistant), and has no built-in knowledge about atoms; rather, it relies on the user to extend it with domain-specific atoms and Lithium functions. This makes Lithium extensible with custom abstractions and adaptable to many reasoning patterns.

Logic programming languages for linear and separation logic. When unfolding the definition of Lithium into their model, Lithium can be seen as a logic programming language for separation logic.[24] Prior work on logic programming for linear or separation logic[25] focuses on identifying large subsets of the underlying logic that remain amenable to logic programming. However, these fragments need expensive techniques like backtracking. In contrast, Lithium is deliberately limited to a well-behaved subset of separation logic that suffices to implement verification tools. By using the syntax of the original program to guide the proof search, Lithium

[20] Santos et al., "Gillian, Part i: A Multi-language Platform for Symbolic Execution", 2020 [San+20]; Maksimovic et al., "Gillian, Part II: Real-World Verification for JavaScript and C", 2021 [Mak+21].

[21] Piskac et al., "Automating Separation Logic with Trees and Data", 2014 [PWZ14]; Lee and Park, "A Proof System for Separation Logic with Magic Wand", 2014 [LP14]; Reynolds et al., "A Decision Procedure for Separation Logic in SMT", 2016 [Rey+16]; Le et al., "Frame Inference for Inductive Entailment Proofs in Separation Logic", 2018 [LSQ18]; Ta et al., "Automated Lemma Synthesis in Symbolic-Heap Separation Logic", 2018 [Ta+18].

[22] Berdine et al., "A Decidable Fragment of Separation Logic", 2004 [BCO04].

[23] Chu et al., "Automatic Induction Proofs of Data-Structures in Imperative Programs", 2015 [CJT15].

[24] In fact, this is how Lithium was presented originally in Sammler et al., "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types", 2021 [Sam+21b].

[25] Andreoli, "Logic Programming with Focusing Proofs in Linear Logic", 1992 [And92]; Hodas and Miller, "Logic Programming in a Fragment of Intuitionistic Linear Logic", 1991 [HM91]; Harland et al., "Programming in Lygon: An Overview", 1996 [HPW96]; Armelín and Pym, "Bunched Logic Programming", 2001 [AP01].

can avoid backtracking, which makes it easier to implement a certifying interpreter for it in Coq.

Part II

RefinedC

Chapter 6

Introduction

Despite numerous advances in programming language technology over the past several decades, a great deal of safety- and security-critical systems software is still programmed in C. The C language remains widely used in large part because it provides fine-grained control over management of resources, which is indispensable to many systems programs. However, this control comes at the steep cost of regularly introducing serious and sometimes catastrophic bugs into code. It has thus long been one of the grand challenges of programming languages research to develop scalable formal methods that can help programmers build C code that is functionally correct, and verifiably so.[1]

Existing tools for formal verification of C programs come in two varieties: *automated* or *foundational*.

On the one hand, automated tools like VeriFast,[2] VCC,[3] and MatchC[4] use a variety of techniques (including both off-the-shelf SMT solvers and bespoke separation-logic solvers) to verify correctness of C programs with minimal user intervention. With these tools, the user still needs to write specifications and provide some annotations (*e.g.*, loop invariants) to aid the proof search, but the verification is otherwise automatic. However, automated tools have a sizable *trusted computing base*: one must trust that the often-sophisticated logic underpinning them is sound—and implemented correctly—since the tools do not provide any form of independently checkable proof.

On the other hand, foundational tools like VST,[5] as well as major verification efforts like CertiKOS[6] and seL4,[7] embed expressive frameworks for verifying C code within a pre-existing logical foundation, typically a general-purpose theorem prover such as Coq or Isabelle/HOL. Foundational tools have the key advantage of a smaller trusted computing base: one need only trust the proof checker of the host theorem prover and the encoding of the operational semantics of C, but not the particular logic or implementation of the tool itself. However, the use of foundational tools typically requires significant manual proof effort: although these frameworks provide tactical support for hiding tedious proof steps, the user must still guide the proof process—*e.g.*, manipulating the proof context, applying lemmas, performing case distinctions, unfolding definitions, instantiating quantifiers—by hand. One exception is Bedrock,[8] which provides much more powerful tactic-based automation. However, Bedrock does not handle many complexities of C, instead targeting a custom

[1] See *e.g.*, NMW02; Con+07; Con+09; Coh+09; RKJ10; Chl11; Chl15; Jac+11; Gre+14; Kre15; Cuo+12; Ell+18; App14; FGK19; Lor+20; Gu+19; Ste14; Sha+05; Fen+08.

[2] Jacobs et al., "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java", 2011 [Jac+11].

[3] Cohen et al., "VCC: A Practical System for Verifying Concurrent C", 2009 [Coh+09].

[4] Stefanescu, "MatchC: A Matching Logic Reachability Verifier Using the K Framework", 2014 [Ste14].

[5] Appel, *Program Logics for Certified Compilers*, 2014 [App14]; Cao et al., "VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs", 2018 [Cao+18].

[6] Gu et al., "Building Certified Concurrent OS Kernels", 2019 [Gu+19]; Gu et al., "Certified Concurrent Abstraction Layers", 2018 [Gu+18]; Gu et al., "Deep Specifications and Certified Abstraction Layers", 2015 [Gu+15].

[7] Klein et al., "seL4: Formal Verification of an OS Kernel", 2009 [Kle+09].

[8] Chlipala, "Mostly-Automated Verification of Low-Level programs in Computational Separation Logic", 2011 [Chl11]; Chlipala, "The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier", 2013 [Chl13]; Chlipala, "From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification", 2015 [Chl15]; Malecha et al., "Compositional Computational Reflection", 2014 [MCB14].

```
struct [[rc::refined_by("a : nat")]] mem_t {
  [[rc::field("a @ int<size_t>")]] size_t len;
  [[rc::field("&own<uninit<a>>")]] unsigned char* buffer;
};

[[rc::parameters("a : nat", "n : nat", "p : loc")]]
[[rc::args("p @ &own<a @ mem_t>", "n @ int<size_t>")]]
[[rc::returns("{n ≤ a} @ optional<&own<uninit<n>>, null>")]]
[[rc::ensures("own p : {n ≤ a ? a - n : a} @ mem_t")]]
void* alloc(struct mem_t* d, size_t sz) {
  if(sz > d->len) return NULL;
  d->len -= sz;
  return d->buffer + d->len;
}
```

Figure 6.1: Memory allocator example in RefinedC.

assembly-like language with a simplified memory model that prohibits many of the optimizations performed by modern C compilers.[9]

In this part of the dissertation, we present **RefinedC**, a new approach to verifying C code that is both automated *and* foundational, while at the same time handling a range of low-level programming idioms including pointer arithmetic, uninitialized memory, and concurrency with data races.

To support *automated* verification, RefinedC employs a novel type system combining *refinement types* and *ownership types*. Refinement types let us express precise invariants on C data types and strong Hoare-style specifications for C functions. Ownership types let us reason modularly about shared state and concurrency by controlling ownership of memory à la Rust.[10] Moreover, RefinedC's type-based approach has the benefit of offering a predictable, syntax-directed approach to automated verification.

To support *foundational* verification, RefinedC follows the *semantic typing* approach of RustBelt:[11] we give meaning to RefinedC's types by interpreting them in the higher-order concurrent separation logic Iris[12] embedded in Coq. The typing rules of RefinedC thus simply become lemmas about our separation-logic model of types, whose soundness we establish (using Iris) in Coq. Separation logic is a natural fit for modeling RefinedC types because (a) it provides a built-in account of ownership-based reasoning, and (b) Iris provides features like invariants and ghost state, which are useful for justifying more sophisticated typing rules concerning shared state and concurrency.

Motivating example. Figure 6.1 shows a concrete example of RefinedC in action. The type `struct mem_t` represents the state of a memory allocator: a block of memory pointed to by `buffer`, whose size is `len`. The `alloc` function tries to allocate `sz` bytes of memory from a `struct mem_t`. It first checks, using `len`, that enough memory is available, and returns `NULL` otherwise. If `buffer` is large enough, then its *last* `sz` bytes are allocated using pointer arithmetic, and `len` is updated accordingly.

The `[[rc::...]]` blocks in Figure 6.1 represent RefinedC annotations,[13] which express a refined version of `mem_t` and a behavioral specification

[9] Chlipala, "The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier", 2013 [Chl13].

[10] Rust team, *The Rust programming language*, 2023 [Rus23].

[11] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language", 2018 [Jun+18a]; Jung et al., "Safe Systems Programming in Rust", 2021 [Jun+21]; Jung, "Understanding and Evolving the Rust Programming Language", 2020 [Jun20].

[12] Jung et al., "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning", 2015 [Jun+15]; Jung et al., "Higher-Order Ghost State", 2016 [Jun+16]; Krebbers et al., "The Essence of Higher-Order Concurrent Separation Logic", 2017 [Kre+17]; Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

[13] Annotations use C2x attributes syntax supported by recent C compilers.

of `alloc` for RefinedC to verify automatically. Here, the refined `mem_t` is indexed by a natural number `a`, the number of bytes available from the allocator. This number must match the value stored in the `len` field as enforced using `a @ int<size_t>`, the singleton type of the `size_t` integer `a`.[14] The `buffer` field is given the type `&own<uninit<a>>`, indicating that it is a pointer to an *owned* block of memory of size `a`. Taken as a whole, the refined `mem_t` encodes the *invariant* that the `len` field contains the length of the owned block pointed to by the `buffer` field.

[14] The unrefined version `int<size_t>` is inhabited by all `size_t` integers.

The specification for `alloc` assumes (in its `rc::args` clause) that the argument `d` points to a `struct mem_t` with `a` available bytes, and that the argument `sz` is equal to some integer value `n`. The `rc::returns` clause specifies the refined type of the value that `alloc` returns: in this case, an `optional` value, which points to an uninitialized block of length `n` if the refinement $n \leq a$ is true, and is `NULL` otherwise. Finally, the `rc::ensures` clause specifies that, upon returning, `alloc` gives back the ownership of `p` (the pointer passed in as the argument `d`), now pointing to a `mem_t` of the appropriately reduced size.

Key idea. One may wonder how the checking of richly-typed specifications like the one for `alloc` can be performed automatically. The key idea is that, even though RefinedC's refinement types encode deep (undecidable) specifications, their syntactic structure serves to judiciously and predictably guide the proof search in a syntax-directed manner. A concrete example of this is the type `b @ optional<T1,T2>` (as seen in the `rc::returns` clause in line 8 of Figure 6.1). Semantically, in our Iris model of RefinedC types, this type corresponds to a *disjunction* (untagged union) between the cases where `b` is true or false; and in general, searching for proofs of disjunctions is difficult because one may make incorrect choices, leading to backtracking. However, as we explain in §9.3, the *syntactic* structure of the program and refinement types provide crucial information that we use to make a definite choice, thus *avoiding backtracking*.

Formally speaking, in order to ensure that RefinedC's typing rules lead to a non-backtracking proof search, we insist that they can be expressed as rules in **Lithium** (introduced in Part I). As a consequence, we directly obtain an automated and foundational method for checking C programs against RefinedC types, by running the Lithium interpreter with the RefinedC typing rules. Additionally, this method is inherently extensible (*e.g.*, to handle new C programming idioms) since it is encoded as an open set of Lithium rules.

The RefinedC toolchain. Figure 6.2 depicts the complete toolchain of RefinedC. Developers write standard C code as they would without RefinedC. To this, they add a functional *specification* in the form of RefinedC's (refinement) types and standard annotations like loop invariants. After this, RefinedC takes over. First, in step (A), a *frontend* that we have created (based on the frontend of Cerberus[15]) translates the C code to a deep embedding of C in Coq, called **Caesium**, and translates the annotations to RefinedC's abstract syntax to Coq. Next, in step (B), Lithium automatically executes RefinedC's typing rules (represented as a Lithium

[15] Memarian et al., "Exploring C Semantics and Pointer Provenance", 2019 [Mem+19].

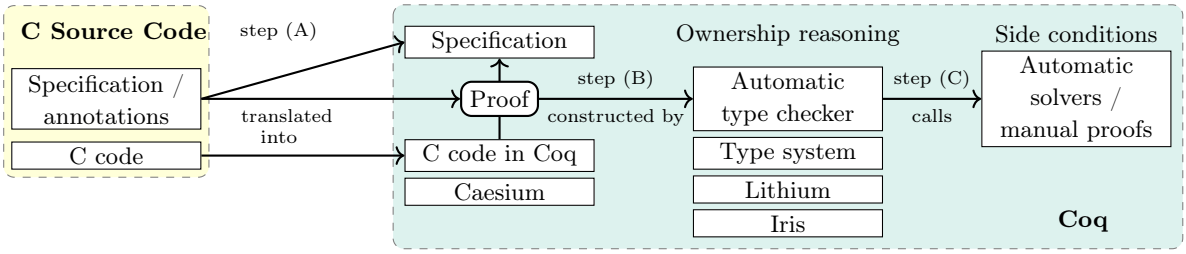


Figure 6.2: The architecture of RefinedC.

program) on the Caesium code to produce a typing derivation proving the specification in Coq. During this process, verification conditions—which are *pure* Coq propositions—are generated. These are mostly automatically discharged using a library of Coq tactics (step (C)), but they can also be discharged by custom (*e.g.*, domain-specific) solvers, or manual proofs.

Under the hood, hidden from the ordinary C programmer, lie RefinedC's types and typing rules, which have been defined ahead of time, in Lithium, by an expert. The expert must define types semantically (as explained above), and prove typing rules sound in Iris against the Caesium C semantics.

Contributions of RefinedC. In this part of the dissertation, we describe the following contributions:

- RefinedC: A foundationally sound and automatic approach to functional verification of idiomatic C code based on refinement and ownership types (§9).
- A frontend translating annotated C code into Caesium, a deep embedding of C in Coq (§8).
- An evaluation of the RefinedC approach using case studies of varying complexity, which demonstrate RefinedC's handling of common low-level C idioms (§10).

Chapter 7

RefinedC by Example

In this chapter, we use motivating examples to introduce RefinedC from the user's point of view. First, we go back in more detail to the example of Figure 6.1 (§7.1). Then, we see how we can make this allocator thread-safe by using a spinlock abstraction (§7.2). Finally, we verify the deallocation mechanism of a more complex allocator relying on a linked list of free chunks, which requires a recursive refinement type and a loop invariant (§7.3).

7.1 A Simple Memory Allocator

```
1  struct [[rc::refined_by("a : nat")]] mem_t {
2    [[rc::field("a @ int<size_t>")]] size_t len;
3    [[rc::field("&own<uninit<a>>")]] unsigned char* buffer;
4  };
5
6  [[rc::parameters("a : nat", "n : nat", "p : loc")]]
7  [[rc::args("p @ &own<a @ mem_t>", "n @ int<size_t>")]]
8  [[rc::returns("{n ≤ a} @ optional<&own<uninit<n>>, null>")]]
9  [[rc::ensures("own p : {n ≤ a ? a - n : a} @ mem_t")]]
10 void* alloc(struct mem_t* d, size_t sz) {
11   if(sz > d->len) return NULL;
12   d->len -= sz;
13   return d->buffer + d->len;
14 }
```

Figure 7.1: Memory allocator example in RefinedC.

As shown in §6, the RefinedC annotations on `struct mem_t` in Figure 7.1 (repeated from Figure 6.1) define a new RefinedC type called `mem_t`, which is parametric in a natural number `a` representing the number of available bytes. We emphasize the difference between the C type `struct mem_t` and the RefinedC type `mem_t`: The C type only specifies the *physical layout*—*e.g.*, the names and the offsets of the fields, which are used by the compiler to generate field accesses—but does not give meaningful correctness guarantees. For example, the C type does not enforce that `len` is a valid integer: it could very well be uninitialized. The RefinedC type `mem_t` captures the invariant satisfied by `struct mem_t` values on which `alloc` operates. Note that RefinedC specifications are purely logical: they do not influence the program's compilation or its runtime behavior.

Specification of `alloc`. We now turn to the annotations assigning a type (*i.e.*, a specification) to the `alloc` function. Our specification introduces a number of logical variables (`rc::parameters` on line 6). Parameters are universally quantified in the specification and, like refinements on a `struct` given with `rc::refined_by`, range over arbitrary mathematical domains (*i.e.*, Coq types). The `alloc` function has three parameters: the natural numbers `a` and `n` representing the number of available bytes and the amount requested by the caller respectively, and the location `p` at which the allocator state is stored. These parameters connect the refinements in the argument and return types, as well as possible pre- and postconditions. The types of the arguments are specified using `rc::args` on line 7. The type `p @ &own<a @ mem_t>` specifies that the first argument of `alloc` is an owned pointer to an allocator state with `a` available bytes, stored at location `p`. The singleton type `n @ int<size_t>` specifies that the second argument of `alloc`—the requested allocation size—is the `size_t` integer with value `n`.

Next, the return type of `alloc` is specified using `rc::returns` on line 8. The return value is an owned pointer if the allocation succeeds, otherwise it is `NULL`. These two possibilities are captured by the type `b @ optional<&own<...>, null>` that represents an owned pointer if the refinement `b` is true, and `null` (the singleton type containing only `NULL`) if the refinement `b` is false. The refinement $\texttt{n} \leq \texttt{a}$[1] checks whether allocation will succeed (*i.e.*, if the allocator state owns enough memory).

The last part of the specification is a postcondition given via the `rc::ensures` annotation on line 9. It says that `alloc` returns the ownership of the `mem_t` (that it received through its first argument) back to its caller. The `mem_t` in the postcondition has an updated refinement since the amount of available memory decreases on a successful allocation. Note that the first argument of `alloc` and the type in the postcondition are refined by the same location `p`. This forces `alloc` to return ownership for the same pointer that it was passed. This ownership transfer pattern often occurs in RefinedC. It is inspired by Mezzo,[2] and is an alternative to Rust's mutable references.

[1] Curly braces `{...}` are used to delimit Coq code in RefinedC annotations.

[2] Pottier and Protzenko, "Programming with Permissions in Mezzo", 2013 [PP13].

Verification. RefinedC verifies the specification of `alloc` without manual intervention. In particular, RefinedC's automation picks the correct case of the returned `optional` by examining the type of the returned value (via rules S-NULL and S-OWN on page 62). It also splits the ownership associated with `buffer` into two following the pointer addition on line 13 (via rule O-ADD-UNINIT on page 62). One part of this ownership stays with `buffer` while the other part is returned to the caller. In §9.3, we explain both techniques further, as well as how the same typing rules also automatically verify a variant of `alloc` that allocates from the start of `buffer` instead of the end.

Error messages. RefinedC's syntax-directed proof search affords precise error messages. For example, suppose the programmer mistakenly writes $\texttt{n} < \texttt{a}$ instead of $\texttt{n} \leq \texttt{a}$ in the specification of `alloc` on line 8. When $\texttt{n} = \texttt{a}$, the code returns a valid pointer, while the specification expects `NULL`,

causing the verification to fail. Concretely, RefinedC fails with the error shown in Figure 7.2. This error message tells the user where in the code the verification failed (at the `return` on line 13), in which branch of the `if` statement on line 11 (the else branch), and what side condition could not be proved. Using this information, the programmer can easily debug the specification.

```
Cannot solve side condition in function "alloc"!
Location: "alloc.c" [13:2-13:28]
Case distinction (n > a) → false at "alloc.c" [11:5-11:18]
...
H3 : ¬ n > a
--------------------------------------
n < a
```

Figure 7.2: Error message for incorrect specification.

7.2 Thread-Safe Allocator Using a Spinlock

```
1  [[rc::parameters("lid : lock_id")]]
2  [[rc::global("spinlock<lid>")]]
3  struct spinlock lock;
4
5  [[rc::parameters("lid : lock_id")]]
6  [[rc::global("spinlocked<lid, {\"data\"}, mem_t>")]]
7  struct mem_t data;
8
9  [[rc::parameters("lid : lock_id", "n : nat")]]
10 [[rc::args("n @ int<size_t>")]]
11 [[rc::requires("[initialized \"lock\" lid]", "[initialized \"data\" lid]")]]
12 [[rc::returns("optional<&own<uninit<n>>, null>")]]
13 void* thread_safe_alloc(size_t sz) {
14   sl_lock(&lock);
15   rc_unwrap(data);
16   void* ret = alloc(&data, sz);
17   sl_unlock(rc_wrap(&lock));
18   return ret;
19 }
```

Figure 7.3: Thread-safe allocation function.

To see how RefinedC provides abstractions that enable reasoning about concurrent code, we consider a thread-safe wrapper of the `alloc` function. A thread can only call the `alloc` function if it has full ownership of the allocator state. And indeed, `alloc` is clearly subject to data races if used concurrently on the same `struct mem_t`. One simple solution to make the allocator thread safe is to protect its global state using a lock—this is exactly what the function `thread_safe_alloc` in Figure 7.3 does. The allocator state is stored in the global variable `data` (line 7), which is protected by spinlock `lock` (line 3). The `thread_safe_alloc` function then simply acquires and releases the lock using `sl_lock` and `sl_unlock` around the call to `alloc` on `data`.[3]

[3] The `rc_unwrap` and `rc_wrap` macros expand to RefinedC annotations, and they are no-ops as far as C is concerned. Moreover, the `rc_wrap` macro is only explicitly included for clarity: it is automatically inserted by RefinedC.

Global variables. Before introducing the spinlock abstraction, we need to take a detour to explain the handling of global variables in RefinedC. Much like function arguments or `struct` fields, global variable are annotated with a type. This type may (again) depend on logical variables specified with `rc::parameters`, and it is itself specified using `rc::global`. However, global variables are special in the sense that their specification (*i.e.*, their type) in only satisfied once they have been explicitly initialized (*e.g.*, by the `main` function).

As a consequence, when a function relies on some global variable being initialized, this fact must be made explicit in its specification with a precondition using the `rc::requires` annotation. Indeed, `thread_safe_alloc` has such a precondition for both global variables `lock` and `data` on line 11. Here, the separation logic assertions `initialized "lock" lid` and `initialized "data" lid`[4] specify that the variables have been initialized, and they also tie the `lid` parameter of the function to the parameter of the same name in the specification of both global variables. This enforces that the two global variables satisfy their specification for the *same* lock identifier.

[4] Inside RefinedC annotations square brackets `[...]` delimit quoted Iris propositions.

Spinlock abstraction. The locking mechanism used in `thread_safe_alloc` is a simple spinlock that was previously verified in RefinedC, and that is used here as a library. The spinlock interface relies on two abstract types `spinlock<...>` and `spinlocked<...>`. The former is the type of a spinlock, and it is parameterized by a `lock_id`, *i.e.*, a unique identifier for a particular spinlock instance. The latter corresponds to the type of a value (whose type is given as third argument) that is protected by the lock identified by the first argument.[5] The main idea for using a lock is that the protected data can only be accessed (*i.e.*, the `spinlocked<...>` type stripped from their type) if a token associated to the lock has been obtained. This token is logically returned by `sl_lock` through a postcondition, and it must be given up when calling `sl_unlock` as it is required as a precondition. It is worth pointing out that this spinlock interface is more general than the standard specification for locks in higher-order concurrent separation logic[6] in that our `spinlocked` type allows adding resources to a lock after it has been allocated.

[5] The second argument of `spinlock<...>` is a string that uniquely identifies the object that is being protected. Indeed, with our spinlock abstraction *one* lock can protect, *e.g.*, multiple global variables.

[6] Hobor et al., "Oracle Semantics for Concurrent Separation Logic", 2008 [HAN08]; Svendsen and Birkedal, "Impredicative Concurrent Abstract Predicates", 2014 [SB14].

Verification. The specification of `thread_safe_alloc` is similar to `alloc` with the exception that `thread_safe_alloc` cannot give any guarantees whether it will succeed or not due to concurrent allocations. Thus, its return type (specified by `rc::returns`) does not give a refinement on the `optional<...>`.

The main challenge in automatically verifying `thread_safe_alloc` is dealing with `spinlocked<...>`. After the lock has been acquired, the `spinlocked<...>` type constructor must be stripped away before one can use the data protected by the spinlock. Moreover, it must be reinstated before releasing the lock. Concretely, the question is: how can the type system decide when to remove and introduce the `spinlocked<...>` type? The answer is not straightforward as there may be several resources protected by the same lock, and not all of them may need to be unwrapped.

```
1 typedef struct
2 [[rc::refined_by("s : {gmultiset nat}")]]
3 [[rc::typedef("chunks_t : {s ≠ ∅} @ optional<&own<...>, null>")]]
4 [[rc::exists("n : nat", "tail : {gmultiset nat}")]]
5 [[rc::size("n")]]
6 [[rc::constraints("{s = {[n]} ⊎ tail}", "{∀ k, k ∈ tail → n ≤ k}")]]
7 chunk {
8   [[rc::field("n @ int<size_t>")]] size_t size;
9   [[rc::field("tail @ chunks_t")]] ~struct chunk* next;~
10 }* chunks_t;
11
12 [[rc::parameters("s : {gmultiset nat}", "p : loc", "n : nat")]]
13 [[rc::args("p @ &own<s @ chunks_t>", "&own<uninit<n>>", "n @ int<size_t>")]]
14 [[rc::requires("{sizeof(struct_chunk) ≤ n}")]]
15 [[rc::ensures ("own p : {{[n]} ⊎ s} @ chunks_t")]]
16 [[rc::tactics ("all: multiset_solver.")]]
17 void free(chunks_t* list, void* data, size_t sz) {
18   chunks_t* cur = list;
19   [[rc::exists("cp : loc", "cs : {gmultiset nat}")]]
20   [[rc::inv_vars("cur : cp @ &own<cs @ chunks_t>")]]
21   [[rc::inv_vars("list : p @ &own<wand<{cp ◁ₗ ({[n]} ⊎ cs) @ chunks_t}, {{[n]} ⊎ s} @ chunks_t>>")]]
22   while(*cur != NULL) {
23     if(sz <= (*cur)->size) break;
24     cur = &(*cur)->next;
25   }
26   chunks_t entry = data;
27   entry->size = sz; entry->next = *cur;
28   *cur = entry;
29 }
```

Figure 7.4: Example of an allocator with a freelist.

Also, the `spinlocked<...>` type may be hidden away behind abstractions. Hence, to keep the system as flexible as possible, it is the responsibility of the programmer to guide the type system using annotations. For this purpose, RefinedC provides the `rc_unwrap` and `rc_wrap` macros in the implementation of `thread_safe_alloc`. With these annotations, RefinedC can automatically verify the `thread_safe_alloc` function.

7.3 Deallocation Using a List of Free Chunks

Next, consider the memory deallocation function `free` in Figure 7.4. This function inserts a chunk of memory that is being freed into a linked list of free memory chunks. When in the list, the initial bytes of a chunk are occupied by a `struct chunk`, which is a header that contains the chunk's `size` (line 8), and a pointer to the `next` chunk (line 9) if there is one, or `NULL` otherwise. The remaining bytes of the chunk can be arbitrary.

It is an invariant of `free` that the chunk list is always sorted in increasing order of chunk size. Hence, `free` has a loop to find where to insert the new chunk (lines 22-25).

Recursive type definition. Figure 7.4 defines two C types: `struct chunk` of chunk headers and `chunks_t` of pointers to such headers. The type `chunks_t` (not `struct chunk`) is refined by the RefinedC type `chunks_t`,

which is defined on line 3. The annotation `rc::typedef` indicates that the defined RefinedC type refines the type of a *pointer* to the surrounding `struct`, not the `struct` itself. The ellipsis in the definition of `chunks_t` is a placeholder for the RefinedC type of the struct.

Note that `chunks_t` is a recursive type: The annotation on the `next` field mentions `chunks_t` again. Unfolding of recursive types is handled by RefinedC automatically; no extra annotations are required to indicate when to unfold.

Multiset and invariant. We explain the type `chunks_t` further. This type is refined by a multiset of natural numbers `s` on line 2. This multiset contains the sizes of all chunks in the list. When `chunks_t` is an owned pointer (*i.e.*, when `s` is not the empty set), the `struct` that it points to is parameterized by the size of the first chunk `n` and the multiset `tail` refining the rest of the list. These two parameters are existentially quantified in the rest of the type (`rc::exists` annotation). A constraint (`rc::constraints` annotation) relates `n` and `tail` to `s`. A second constraint says that `n` is less than or equal to all elements of `tail`, which implies that the list of chunks is sorted. The last interesting point about `chunks_t` is the `rc::size` annotation on line 5. This annotation means that the chunk actually occupies `n` bytes in memory, of which the C type (`struct chunk`) only describes the initial part. In other words, the chunk is of size `n` bytes and a `struct chunk` (the header) is *overlaid* at its beginning. The remaining bytes of the chunk are treated as uninitialized by RefinedC.

Loop invariant and verification. The formal specification of `free` should be unsurprising. It says that when `free` is passed a free list with chunks of sizes `s` and a pointer to an owned chunk of size `n` (this is the block to be freed), then at the end of `free`, the free list contains chunks of sizes `{[n]} ⊎ s` (using Coq multiset operation notations). Importantly, `free` has a precondition (line 14) that the block being added to the free list is large enough to fit the `struct chunk` header.

Verifying `free` in RefinedC requires an explicit loop invariant (lines 19-21). Loop invariants are described using three kinds of annotations: `rc::exists` introduces local, existentially quantified logical variables, `rc::inv_vars` specifies RefinedC types of relevant program variables at the start of each loop iteration, and `rc::constraints` lists additional assertions. (This example does not need `rc::constraints`.)

The loop invariant tracks the ownership of the list as it is traversed. Logically, the list has two parts: the suffix that has not yet been traversed and the prefix that has already been traversed. These two parts are pointed to by the local variable `cur` and the argument variable `list`, respectively. The loop invariant associates *ownership* of the list's two parts to these two variables. Specifically, it introduces a multiset variable `cs` corresponding to the multiset refinement of the suffix and asserts that `cur` points to an owned list of chunk sizes from `cs`. Next, it asserts that *if* this ownership extended with a chunk of size `n` (the new chunk) is combined with the ownership associated with `list`, *then* one obtains ownership of the entire output list (sizes from multiset `{[n]} ⊎ s`). This

if-then relation is conveniently expressed using the `wand<...>` type using a standard technique for expressing partial data structures via the magic wand of separation logic.[7]

Finally, the annotation `rc::tactics` on line 16 instructs RefinedC to use the multiset solver from the std++ Coq library[8] for proving the side conditions in this example that RefinedC's default solver cannot prove.

[7] Cao et al., "Proof Pearl: Magic Wand as Frame", 2019 [Cao+19].

[8] Coq-std++ team, *An extended "standard library" for Coq*, 2020 [Coq20].

Chapter 8

RefinedC Frontend and Caesium

Before a C program can be verified by RefinedC, it is elaborated by the RefinedC frontend to a core language we call **Caesium**. This language is control-flow graph-based, and given a formal semantics through a deep embedding in Coq. The core of this semantics is a low-level memory model that is roughly based on that of CompCert.[1] Caesium provides both sequentially consistent and non-atomic memory accesses, and assigns undefined behavior to data races following the semantics of RustBelt.[2] Caesium supports many low-level idioms like pointer arithmetic, the address-of operator (also on local variables), access to representation bytes, fixed-size integers, goto (including unstructured switches, such as Duff's device), alignment checks, composite types as arguments and return values, uninitialized memory with poison semantics,[3] and first-class function pointers. The RefinedC frontend is implemented in OCaml and relies on the first half of the pipeline of Cerberus.[4]

Since RefinedC aims at the verification of low-level systems code (like allocators, as shown in §7), the Caesium semantics is more permissive than what the ISO C standard describes. Indeed, it is well documented that ISO C and de facto practices commonly found in low-level systems code disagree on many aspects of the C memory model.[5] Hence, the Caesium memory model has less undefined behavior than ISO C with respect to, *e.g.*, padding in structs and effective types.

Caesium lacks some features of ISO C that are subject to active research. It does not support C's loose expression evaluation ordering[6] (Caesium fixes a left-to-right ordering), lifetimes of block-scoped variables[7] (all local variables are function-scoped in Caesium), and relaxed-memory concurrency[8] (Caesium's only atomic accesses are sequentially consistent). To mitigate the first two points, the RefinedC frontend performs an over-approximating analysis that emits warnings if an expression may be non-deterministic, or if the address of a block-scoped variable could escape. Caesium has been extended with support for integer-pointer casts by Lepigre et al.[9] which is not part of this dissertation.

Trusted computing base. The trusted computing base (TCB) of RefinedC includes the implementation of the frontend, the definition of the Caesium semantics, and Coq. The frontend contains around 6000 lines of OCaml code (excluding Cerberus) that transform Cerberus's AIL intermediate language into a control-flow graph and translate AIL constructs to Caesium (almost 1-to-1). The definition of the Caesium semantics is roughly 2500

[1] Leroy and Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations", 2008 [LB08]; Leroy et al., *The CompCert Memory Model, Version 2*, 2012 [Ler+12].

[2] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language", 2018 [Jun+18a].

[3] Lee et al., "Taming Undefined Behavior in LLVM", 2017 [Lee+17].

[4] Memarian et al., "Exploring C Semantics and Pointer Provenance", 2019 [Mem+19].

[5] Wang et al., "Undefined behavior: What happened to my code?", 2012 [Wan+12]; Memarian et al., "Into the Depths of C: Elaborating the De Facto Standards", 2016 [Mem+16]; Memarian et al., "Exploring C Semantics and Pointer Provenance", 2019 [Mem+19].

[6] Hathhorn et al., "Defining the Undefinedness of C", 2015 [HER15]; Krebbers, "An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C", 2014 [Kre14]; Frumin et al., "Semi-automated Reasoning About Non-determinism in C Expressions", 2019 [FGK19].

[7] Hathhorn et al., "Defining the Undefinedness of C", 2015 [HER15]; Krebbers and Wiedijk, "Separation Logic for Non-local Control Flow and Block Scope Variables", 2013 [KW13].

[8] Batty et al., "Mathematizing C++ Concurrency", 2011 [Bat+11]; Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris", 2017 [Kai+17]; Dang et al., "RustBelt Meets Relaxed Memory", 2020 [Dan+20].

[9] Lepigre et al., "VIP: Verifying Real-World C Idioms with Integer-Pointer Casts", 2022 [Lep+22].

lines of Coq code (including some proofs) and additionally uses definitions from the Coq standard library, std++, and the language interface of Iris. The Iris logic itself is not part of the TCB since its adequacy theorem establishes a closed Coq statement that involves just the operational semantics. Similarly, the RefinedC type system and Lithium need not be trusted since they generate proofs in Iris.

Chapter 9

RefinedC Type System

This chapter describes the RefinedC type system: First, we present some commonly used RefinedC types in §9.1, then we describe the semantic model of RefinedC's types in §9.2, and, finally, we discuss some interesting RefinedC typing rules in §9.3.

9.1 RefinedC Types

Type	Intuitive semantics
n @ $\mathsf{int}(\alpha)$	C integer of type α that encodes n
ϕ @ bool	Boolean reflecting the truth of ϕ
ℓ @ $\&_{\mathsf{own}}(\tau)$	unique ownership of τ at location ℓ
$\mathsf{uninit}(n)$	n uninitialized (*i.e.*, arbitrary) bytes
null	singleton type of `NULL`
ϕ @ $\mathsf{optional}(\tau_1, \tau_2)$	if ϕ then τ_1 else τ_2
$\mathsf{wand}(H, \tau)$	τ with hole H
$\mathsf{struct}_\sigma\ \overline{\tau}$	struct with layout σ, fields of types $\overline{\tau}$
$\exists x.\, \tau(x)$	type-level existential quantifier
$\{\tau \mid \phi\}$	τ with constraint ϕ
$\mathsf{padded}(\tau, n)$	τ padded to n bytes

Figure 9.1: A selection of RefinedC types.

Several interesting RefinedC types, along with their intuitive meaning, are shown in Figure 9.1. (These types also appeared in earlier examples.) In RefinedC, most types can have a *refinement*, an optional parameter that limits values in the type. A refinement is a logical predicate on values of the type, but the meta-level sort of the refinement and the predicate vary from type to type. For example, the type $\mathsf{int}(\alpha)$ can be refined by a mathematical integer n to form the type n @ $\mathsf{int}(\alpha)$ that represents the singleton set $\{n\}$ of α-sized integers.[1] The type ϕ @ bool is the single Boolean value reflecting the validity of proposition ϕ.[2] The refinement type ℓ @ $\&_{\mathsf{own}}(\tau)$ denotes an *owned* (non-aliased) pointer and its refinement ℓ specifies the exact memory location that is owned. As examples, the annotations on `mem_t` on line 3 in Figure 6.1 use $\&_{\mathsf{own}}(\tau)$ together with $\mathsf{uninit}(n)$ to denote a pointer to a block of n bytes of uninitialized memory. The type ϕ @ $\mathsf{optional}(\tau_1, \tau_2)$ is a type-level case distinction on the validity of ϕ. It is most commonly used to represent nullable pointers (via $\&_{\mathsf{own}}(\tau)$ and null), as illustrated in §7. Another interesting type is $\mathsf{wand}(H, \tau)$,

[1] The *int type* α describes the number of bits in the integer and whether it is signed. For example, `i32` represents a signed, 32-bit integer.

[2] Technically, the bool type is also parametrized by an int type that describes its size. The implementation of RefinedC also distinguishes between strict Booleans that are either zero or one, and relaxed Booleans that are zero or non-zero.

which is used to encode partial data structures via the magic wand.[3] This type is for example used by the loop invariant of `free` in Figure 7.4.

The last four types in Figure 9.1 are most often *generated* from other annotations (although they can be used directly, too). A structure type $\mathsf{struct}_\sigma\,\overline{\tau}$ is built by combining the types given by the `rc::field` annotations on a C `struct` (*e.g.*, lines 2-3 in Figure 6.1). The types $\exists x.\,\tau(x)$ and $\{\tau \mid \phi\}$ are generated from `rc::exists` and `rc::constraints` annotations (*e.g.*, lines 4-6 of Figure 7.4). Finally, the type $\mathsf{padded}(\tau, n)$, which represents type τ padded to n bytes, is generated from `rc::size` annotations (*e.g.*, line 5 of Figure 7.4).

[3] Cao et al., "Proof Pearl: Magic Wand as Frame", 2019 [Cao+19].

Function types. Functions have RefinedC types of the form:

$$\mathsf{fn}(\forall x.\,\overline{\tau_{\mathsf{arg}}}; H_{\mathsf{pre}}) \to \exists y.\,\tau_{\mathsf{ret}}; H_{\mathsf{post}}$$

Function types are generated from the source code annotations we have already seen. For example, the annotations on `alloc` (lines 6-9 of Figure 6.1) lead to the function type $\mathsf{alloc}_{\mathsf{spec}}$ shown in Figure 9.2.

$$\begin{aligned} a \mathbin{@} \mathsf{mem\_t} &\triangleq \mathsf{struct}_{\texttt{struct mem\_t}}\,[a \mathbin{@} \mathsf{int}(\mathsf{size\_t}),\ \&_{\mathsf{own}}(\mathsf{uninit}(a))] \\ \mathsf{alloc}_{\mathsf{spec}} &\triangleq \mathsf{fn}(\forall(a, n, p).\,p \mathbin{@} \&_{\mathsf{own}}(a \mathbin{@} \mathsf{mem\_t}),\ n \mathbin{@} \mathsf{int}(\mathsf{size\_t}); \mathsf{True}) \\ &\quad \to \exists().\,(n \le a) \mathbin{@} \mathsf{optional}(\&_{\mathsf{own}}(\mathsf{uninit}(a)),\ \mathsf{null}); p \triangleleft_l ((n \le a)\ ?\ (a - n) : a) \mathbin{@} \mathsf{mem\_t} \end{aligned}$$

Figure 9.2: The formal specification of `alloc` (Figure 6.1) in RefinedC's type system.

Logical variables in the `rc::parameters` annotation (line 6) correspond to x in the function type, the annotations `rc::args` and `rc::returns` (lines 7-8) correspond to $\overline{\tau_{\mathsf{arg}}}$ and τ_{ret}, respectively, and the annotations `rc::requires` and `rc::ensures` (line 9) correspond to H_{pre} and H_{post}, respectively. Existential variables that are bound in the return type and the postconditions by `rc::exists` correspond to y. RefinedC function types are first-class: functions can be stored in memory and passed to or returned from other functions.

9.2 Model of RefinedC Types

We have seen a few examples of RefinedC types, but so far we have not answered the question: What *is* a RefinedC type? Inspired by the *semantic typing* approach of RustBelt,[4] RefinedC types are defined semantically in the Iris separation logic. Concretely, a RefinedC type $\tau \in \mathsf{type}$ is defined via the following predicates:

loc. assign.: $\ell \triangleleft_l \tau$ val. assign.: $v \triangleleft_{\mathsf{v}} \tau$ type layout: τ has layout ι

RefinedC types have two type assignments: a location type assignment $\ell \triangleleft_l \tau$[5] that states that the location ℓ has type τ and a value type assignment that states that the value v has type τ, both defined as separation logic predicates. Additionally, each RefinedC type comes with a (pure) predicate "τ has layout ι". This predicate describes whether the type τ has the layout ι.[6,7]

[4] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language", 2018 [Jun+18a]; Jung et al., "Safe Systems Programming in Rust", 2021 [Jun+21]; Jung, "Understanding and Evolving the Rust Programming Language", 2020 [Jun20].

[5] Technically, the location type assignment is parametrized by an own state β that describes whether the ownership of ℓ is exclusively owned or shared. The shared version is used for the types of global variables.

[6] A layout ι is a combination of a size $\mathsf{size}(\iota)$ and an alignment $\mathsf{align}(\iota)$.

[7] Lepigre et al., "VIP: Verifying Real-World C Idioms with Integer-Pointer Casts", 2022 [Lep+22] extend τ has layout ι from a layout to a C type to support integer pointer casts when loading from memory, but we ignore this extension in this dissertation.

All these predicates are linked by the following rules that each type as to fulfill:

TY-ALIGNED
$$\ulcorner \tau \ \mathsf{has\ layout}\ \iota \urcorner \mathrel{-\!\!*} \ell \lhd_l \tau \mathrel{-\!\!*} \ulcorner (\ell \,|\, \mathrm{align}(\iota)) \urcorner$$

TY-SIZE
$$\ulcorner \tau \ \mathsf{has\ layout}\ \iota \urcorner \mathrel{-\!\!*} v \lhd_\mathsf{v} \tau \mathrel{-\!\!*} \ulcorner |v| = \mathsf{size}(\iota) \urcorner$$

TY-DEREF
$$\ulcorner \tau \ \mathsf{has\ layout}\ \iota \urcorner \mathrel{-\!\!*} \ell \lhd_l \tau \mathrel{-\!\!*} \exists v.\, l \mapsto v * v \lhd_\mathsf{v} \tau$$

TY-REF
$$\ulcorner \tau \ \mathsf{has\ layout}\ \iota \urcorner \mathrel{-\!\!*} \ulcorner (\ell \,|\, \mathrm{align}(\iota)) \urcorner \mathrel{-\!\!*} l \mapsto v \mathrel{-\!\!*} v \lhd_\mathsf{v} \tau \mathrel{-\!\!*} \ell \lhd_l \tau$$

TY-ALIGNED (and TY-SIZE) state that the type assignments guarantee that the location (resp. value) has the alignment (resp. size) described by the predicate "τ has layout ι".[8] The rules TY-DEREF and TY-REF govern the relationship between $\ell \lhd_l \tau$ and $v \lhd_\mathsf{v} \tau$: TY-DEREF states that a location type assignment can be turned into a points-to predicate for a corresponding value type assignment and TY-REF vice versa. These properties are used when loading and storing values from and to memory.[9]

[8] $(\ell \,|\, n)$ states that the location ℓ is aligned to an n-byte boundary. $|v|$ denotes the size of the value v in memory.

[9] We omit the rules for shared ownership, *i.e.*, that owned location type assignments can be turned into shared location type assignments and that shared location type assignments are persistent.

Immovable types. TY-DEREF and TY-REF don't hold unconditionally, but only if there exists a layout ι such that τ has layout ι holds. This enables what we call *immovable types*: Immovable types are types that can only exist in memory, but not as a value and thus cannot be loaded from and stored to memory. The most common immovable type is the $\mathsf{place}(\ell)$ type that asserts that it is stored at memory location ℓ, but contains no further ownership of this location.[10] This type is useful when type checking the address-of operator as it allows to move the ownership out of a memory location by replacing the type of the memory location with $\mathsf{place}(\ell)$.

[10] $\mathsf{place}(\ell)$ is defined as $\ell' \lhd_l \mathsf{place}(\ell) \triangleq \ulcorner \ell = \ell' \urcorner$, $v \lhd_\mathsf{v} \mathsf{place}(\ell) \triangleq \mathsf{False}$, and $\mathsf{place}(\ell)\ \mathsf{has\ layout}\ \iota \triangleq \mathsf{False}$. The last definition makes all properties for a type definition hold trivially.

Refinement types. A refinement type $\hat{\tau}$ is a function from the type of the refinement A to a type, *i.e.*, we have

$$\hat{\tau} \in \mathsf{rtype}_A \triangleq A \to \mathsf{type} \qquad x \mathbin{@} \hat{\tau} \triangleq \hat{\tau}\ x$$

Thus, refinement types are just normal RefinedC types with one distinguished parameter selected as the refinement. They provide the following two benefits: (1) a refinement can be omitted to implicitly existentially quantify it, *i.e.*, a refinement type $\hat{\tau}$ that is used without a refinement is automatically elaborated to $\exists x.\, x \mathbin{@} \hat{\tau}$, (2) the type system automatically treats the refinement x as an output of the type, *i.e.*, a subsumption between two types that only differ in the refinement is reduced to an equality between the refinements.

Extensibility. All RefinedC types are defined using the semantic model described in this section. The benefit of this approach is that this makes RefinedC's type system extensible: A user can add a new type to RefinedC by giving a definition in the model described above. Additionally, the user can also add new typing rules for the types, by extending the Lithium program representing the RefinedC type checker with new clauses.

9.3 Examples of RefinedC Typing Rules

Next, we explain selected typing rules, shown in Figure 9.3. Every typing judgment in RefinedC is a Lithium function and typing rules correspond

IF-BOOL
1: $\vdash^{\Sigma}_{\text{IF}} \phi$ @ bool `then` s_1 `else` s_2 :-
2: `if` ϕ `then` $\vdash^{\Sigma}_{\text{STMT}} s_1$
3: `else` $\vdash^{\Sigma}_{\text{STMT}} s_2$

IF-INT
1: $\vdash^{\Sigma}_{\text{IF}} n$ @ int(α) `then` s_1 `else` s_2 :-
2: `if` $n \neq 0$ `then` $\vdash^{\Sigma}_{\text{STMT}} s_1$
3: `else` $\vdash^{\Sigma}_{\text{STMT}} s_2$

T-IF
1: $\vdash^{\Sigma}_{\text{STMT}}$ `if` e `then` s_1 `else` s_2 :-
2: $v, \tau \leftarrow \vdash_{\text{EXPR}} e$;
3: $\vdash^{\Sigma}_{\text{IF}} \tau$ `then` s_1 `else` s_2

T-BINOP
1: $\vdash_{\text{EXPR}} e_1 \odot e_2$ G :-
2: $v_1, \tau_1 \leftarrow \vdash_{\text{EXPR}} e_1$;
3: $v_2, \tau_2 \leftarrow \vdash_{\text{EXPR}} e_2$;
4: $v, \tau \leftarrow \vdash_{\text{BINOP}} (v_1 : \tau_1) \odot (v_2 : \tau_2)$;
5: `return`$_G$ v, τ

O-OPTIONAL-EQ
1: $\vdash_{\text{BINOP}} (v_1 : \phi$ @ optional($\&_{\text{own}}(\tau)$, null)) $= (v_2 :$ null) G :-
2: `if` ϕ `then`
3: `inhale` $v_1 \lhd_{\text{v}} \&_{\text{own}}(\tau)$; `return`$_G$ `false`, False @ bool
4: `else`
5: `inhale` $v_1 \lhd_{\text{v}}$ null; `return`$_G$ `true`, True @ bool

S-NULL
1: $v \lhd_{\text{v}}$ null $<: v \lhd_{\text{v}} \phi$ @ optional($\&_{\text{own}}(\tau)$, null) G :-
2: `exhale` $\ulcorner\neg\phi\urcorner$; `return`$_G$

S-OWN
1: $v \lhd_{\text{v}} \&_{\text{own}}(\tau_1) <: v \lhd_{\text{v}} \phi$ @ optional($\&_{\text{own}}(\tau)$, null) G :-
2: $\forall \ell$. `inhale` $\ell \lhd_l \tau_1$; `exhale` $\ulcorner\phi\urcorner * \ell \lhd_l \tau$; `return`$_G$

O-ADD-UNINIT
1: $\vdash_{\text{BINOP}} (v_1 : \&_{\text{own}}(\text{uninit}(n_1))) + (v_2 : n_2$ @ int(size_t)) G :-
2: `exhale` $\ulcorner 0 \leq n_2 \leq n_1 \urcorner$; `inhale` $v_1 \lhd_{\text{v}} \&_{\text{own}}(\text{uninit}(n_2))$; `return`$_G$ $v_1 +_l n_2, \&_{\text{own}}(\text{uninit}(n_1 - n_2))$

CAS-BOOL
1: $\vdash_{\text{CAS}}$ `CAS`$(v_1 :$ atomicbool$(H_\top, H_\bot), v_2 : \&_{\text{own}}(b_1$ @ bool$), v_3 : b_2$ @ bool) G :-
2: `do`
3: `inhale` $v_2 \lhd_{\text{v}} \&_{\text{own}}(\neg b_1$ @ bool); `return`$_G$ `false`, False @ bool
4: `and`
5: `inhale` $(b_1 \ ? \ H_\top : H_\bot)$; `exhale` $(b_2 \ ? \ H_\top : H_\bot)$;
6: `inhale` $v_2 \lhd_{\text{v}} \&_{\text{own}}(b_1$ @ bool); `return`$_G$ `true`, True @ bool

Figure 9.3: Selected RefinedC typing rules. (Simplified by, *e.g.*, omitting refinements of $\&_{\text{own}}$ and existentials $\vec{x}$.)

to rules for these functions. RefinedC's type assignments are considered Lithium atoms. For a description of Lithium and the syntax used to define the rules, see §3.

Judgment basics. RefinedC has a specialized typing judgment for each program construct, *e.g.*, $\vdash_{\text{IF}}$ for conditional statements and $\vdash_{\text{BINOP}}$ for binary operators. These judgments are parameterized by the types of the values they operate on. This ensures that Lithium's proof search does not need to backtrack since these types uniquely determine the applicable rule. For example, consider the rules IF-BOOL and IF-INT in Figure 9.3. Depending on the type of the condition (bool vs. int) a different rule applies and typing proceeds differently. Such type-based overloading allows RefinedC to handle the same program construct differently depending on the context. This is useful because, in C, the same construct may serve different purposes.

Construct-specific judgments arise in the premises of rules for general statement and expression judgments, *e.g.*, T-IF or T-BINOP. The expression judgment $\vdash_{\text{EXPR}} e$ takes an expression e and returns its inferred type (together with a symbolic value).[11] These inferred types and values are then passed to the specialized judgments like $\vdash_{\text{BINOP}}$.

[11] $\vdash_{\text{EXPR}} e$ is similar to exprOk from §3 except that $\vdash_{\text{EXPR}} e$ also returns a type.

Typing rules for optional. As demonstrated in §7.1, the optional type of RefinedC plays a key role in handling the common low-level programming pattern of encoding an error value as `NULL`. Most uses of this pattern can

be handled by three RefinedC typing rules: the rule O-OPTIONAL-EQ for comparing an optional with NULL, and the two rules S-NULL and S-OWN for introducing an optional type.

The rule O-OPTIONAL-EQ is used to handle goals of the form

$$\vdash^{\Sigma}_{\text{STMT}} \texttt{if}\ (e = \texttt{NULL})\ \texttt{then}\ s_1\ \texttt{else}\ s_2$$

To do this, Lithium first applies T-IF, which requires typing the Boolean expression $e = \texttt{NULL}$. It then applies T-BINOP, which requires typing e. Suppose Lithium infers the type $\phi \mathbin{@} \mathsf{optional}(\&_{\mathsf{own}}(\tau), \mathsf{null})$ for e. Next, Lithium types the second expression, NULL. This is trivial as NULL has type null. At this point, Lithium's goal is a judgment that matches O-OPTIONAL-EQ.

We now explain O-OPTIONAL-EQ in detail. The rule distinguishes two cases via **if**, corresponding to the cases where ϕ holds or does not hold. When ϕ holds (first case), v_1 must be an owned pointer, which cannot equal NULL, so the result of the equality check in the conclusion of the rule must be false. Accordingly, in this case, we add $v_1 \lhd_{\mathsf{v}} \&_{\mathsf{own}}(\tau)$ to the context and return `false`. When ϕ does not hold (second case), v_1 must have the type null, so v_1 must be NULL and, hence, equal to v_2. Accordingly, we add $v_1 \lhd_{\mathsf{v}} \mathsf{null}$ to the context and return `true`.

In either of these two cases, the typing of the `if` statement continues using IF-BOOL (with the meta-variable ϕ of IF-BOOL instantiated to False or True, respectively). This rule also uses Lithium's **if** primitive to perform a case distinction, but Lithium automatically selects the right branch based on the value of ϕ.

Next, we explain how Lithium establishes that a value v has type $\phi \mathbin{@} \mathsf{optional}(\&_{\mathsf{own}}(\tau), \mathsf{null})$. This corresponds to proving the following Lithium goal:[12]

$$\ldots \Vdash \mathbf{exhale}\ v \lhd_{\mathsf{v}} \phi \mathbin{@} \mathsf{optional}(\&_{\mathsf{own}}(\tau), \mathsf{null}); \ \ldots$$

[12] We omit existentials $\vec{x}$ in this chapter to avoid clutter.

As discussed in §3.5, Lithium's main method of proving such a goal is to search the context for a related atom A' and then introduce a subsumption. Concretely, a type assignment for a value v is related to a type assignment A' for v in the context.[13] Typically, A' will type v at either null or $\&_{\mathsf{own}}(\tau')$ for some τ'. In the first case, Lithium creates a new goal of the form

[13] Similarly, location type assignments are related to location type assignments for the same location.

$$\ldots \Vdash v \lhd_{\mathsf{v}} \mathsf{null} <: v \lhd_{\mathsf{v}} (\phi \mathbin{@} \mathsf{optional}(\&_{\mathsf{own}}(\tau), \mathsf{null})); \ \ldots$$

At this point, rule S-NULL is used to reduce the goal to proving $\neg\phi$ (and the continuation), which is what one expects from the intuitive meaning of the optional type. In the second case, Lithium's goal is

$$\ldots \Vdash v \lhd_{\mathsf{v}} \&_{\mathsf{own}}(\tau') <: v \lhd_{\mathsf{v}} (\phi \mathbin{@} \mathsf{optional}(\&_{\mathsf{own}}(\tau), \mathsf{null})); \ \ldots$$

Using rule S-OWN, this reduces to first introducing $\ell \lhd_l \tau'$ for some fresh location ℓ and then proving ϕ and $\ell \lhd_l \tau$, which again follows the meaning of the optional type.

Ownership reasoning. Next, we explain how program syntax guides ownership reasoning in RefinedC. Consider the expression `d->buffer + d->len`

on line 13 of Figure 6.1. Logically, this expression splits the ownership of `d->buffer` into two parts: one part that remains associated with `d->buffer`, and a second part that is returned to the caller with the allocated memory. This reasoning is performed by the rule O-ADD-UNINIT, which types the addition of an integer n_2 to a pointer to uninitialized memory of length n_1 (RefinedC type $\mathsf{uninit}(n_1)$). The rule splits $\mathsf{uninit}(n_1)$ into the smaller pieces $\mathsf{uninit}(n_2)$ and $\mathsf{uninit}(n_1 - n_2)$, after checking that $n_2 \leq n_1$. This rule is a representative instance of how RefinedC's informative types disambiguate the intended logical meaning of a commonly overloaded C operator (+ in this case).

Note that O-ADD-UNINIT can be reused in other contexts where programs add values of type $\&_{\mathsf{own}}(\mathsf{uninit}(n))$ and $\mathsf{int}(\mathsf{size\_t})$. For example, say we change the implementation of `alloc` to allocate from the beginning of `buffer` instead of the end, *i.e.*, replacing line 13 in Figure 6.1 with the following:

```
1   unsigned char *res = d->buffer;
2   d->buffer += sz;
3   return res;
```

RefinedC automatically verifies the resulting version of `alloc` without further changes since O-ADD-UNINIT is general enough to cover the type checking of + in both cases. The only difference is that the two versions distribute v_1 and $v_1 +_l n_2$ differently. In the original version, v_1 and the associated $\&_{\mathsf{own}}(\mathsf{uninit}(n_2))$ stay in `buffer`, while $v_1 +_l n_2$ is returned with $\&_{\mathsf{own}}(\mathsf{uninit}(n_1 - n_2))$. In the new version, `buffer` is updated to $v_1 +_l n_2$, while the original value v_1 is returned.[14]

[14] This variant of the example was suggested by a reviewer for Sammler et al., "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types", 2021 [Sam+21b]; it type checked without requiring any changes to RefinedC or its typing rules.

Fine-grained concurrency. RefinedC can also automatically verify fine-grained concurrent code. We illustrate this with the $\mathsf{atomicbool}(H_\top, H_\bot)$ type, which represents a Boolean that can be accessed atomically. The type holds the ownership of $H_\top$ if the Boolean is true, and of $H_\bot$ if the Boolean is false. For example, a spinlock that protects the resource H can be modeled as the type $\mathsf{atomicbool}(\mathsf{True}, H)$.

The main atomic operation supported by the atomicbool type is the `atomic_compare_exchange_strong` function, corresponding to Caesium's $\texttt{CAS}(\ell_{atom}, \ell_{exp}, v_{des})$ operation. The first argument (ℓ_{atom}) is a pointer to the value to be modified atomically, the second argument (ℓ_{exp}) is a pointer to the expected current value of ℓ_{atom}, and the third argument (v_{des}) is the value to be assigned to ℓ_{atom}. `CAS` also sets ℓ_{exp} to the previous value stored at ℓ_{atom}.

`CAS` is verified using the rule CAS-BOOL. The second and third arguments of `CAS` have singleton Boolean types that determine whether the premise uses $H_\top$ or $H_\bot$. CAS-BOOL has two cases corresponding to whether the `CAS` fails or succeeds. (First case) When `CAS` fails, the second argument is updated to $\neg b_1$, and `false` is returned. (Second case) When `CAS` succeeds, we receive ownership stored with the atomic Boolean before the `CAS`, and have to prove ownership stored after the `CAS`. Subsequently, we receive ownership of v_2, and the `CAS` returns `true`. (The implementation of the spinlock mentioned earlier uses CAS-BOOL with $b_1 \triangleq$ `false` and $b_2 \triangleq$ `true`,

which means that on a successful `CAS`, one receives the ownership of H stored in the spinlock.)

The RefinedC type atomicbool hides complex Iris concepts related to fine-grained concurrency like impredicative invariants and ghost state. These concepts show up only in proving the soundness of cas-bool, which we have done once and for all in Coq. Lithium's automation only uses the much simpler *statement* of the cas-bool rule, not its proof.

Chapter 10

Evaluation and Case Studies

Class	Test	Types used	Rules	∃	⌜ϕ⌝	Impl	Spec	Annot	Pure	Ovh
#1	Singly linked list	wand, alloc	44/613	119	47/5	106	33	24 (4/20/0)	2	∼0.2
	Queue	list segments, alloc	42/310	81	10/0	42	15	9 (9/0/0)	0	∼0.2
	Binary search	arrays, func. ptr.	40/308	68	73/6	42	16	6 (0/5/1)	19	∼0.6
#2	Thread-safe allocator	wand, padded, lock	58/319	96	28/2	68	18	21 (14/2/5)	3	∼0.4
	Page allocator	padded	40/236	60	14/0	43	14	14 (14/0/0)	0	∼0.3
#3	Bin. search tree (layered)	wand, alloc	50/964	216	50/11	133	65	22 (8/7/7)	128	∼1.1
	Bin. search tree (direct)	wand, alloc	48/977	240	47/43	115	43	17 (8/7/2)	10	∼0.2
#4	Linear probing hashmap	unions, arrays, alloc	57/1167	356	175/39	111	46	34 (14/17/3)	265	∼2.7
#5	Hafnium mpool allocator	wand, padded, lock	72/1730	515	122/11	191	53	55 (28/19/8)	5	∼0.3
#6	Spinlock	atomic Boolean	25/65	10	14/1	24	12	13 (0/1/12)	1	∼0.6
	One-time barrier	atomic Boolean	18/34	5	6/0	20	7	2 (0/0/2)	0	∼0.1

Types used: Salient type constructs used in the program. Rules: Number of distinct typing rules / number of typing rule applications. ∃: Number of automatically instantiated existential quantifiers. ⌜ϕ⌝: Number of side conditions automatically proved / manually proved. Impl: Lines of C code (counted by tokei [Tok23]). Spec: Lines of top-level (function) specification. Annot: Lines of annotation in source code (numbers in parentheses show breakdown into data structure invariants / loop annotations / other annotations). Pure: Lines of pure Coq reasoning, including definitions and lemma statements. Ovh: Sum of Annot and Pure divided by Impl.

Figure 10.1: Evaluation.

To evaluate the automation and expressiveness of RefinedC, we verified full functional correctness of six classes of programs in Figure 10.1.[1] We selected these programs to cover a wide variety of reasoning patterns ranging over standard benchmarks (#1), tricky ownership reasoning (#2), difficult side conditions (#3, #4), real-world C code (#5) and concurrent algorithms (#6).

First, the table in Figure 10.1 lists the most interesting types used by each example. This shows how RefinedC types like wand or padded are reused across different programs. Then, the table shows the number of RefinedC typing rules used in type checking each of the examples. All typing rules used by the examples are either automatically generated unfolding rules for user-defined types or they are part of the RefinedC standard library. This standard library contains around 30 types and 200 typing rules. Lithium automatically selects and applies the right typing rule from these predefined rules. Figure 10.1 shows how many such automatic rule applications Lithium performs. This number gives a sense of the automation afforded by Lithium, showing the extent to which typing rules handle tasks like ownership manipulation and unfolding of definitions

[1] The description and numbers in this chapter reflect the original evaluation in Sammler et al., "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types", 2021 [Sam+21b].

that must be performed manually in some other tools. Additionally, the table shows how many existential quantifiers are automatically instantiated via the heuristics described in §4.2. Across all programs, we had to instantiate only one existential manually (in Spinlock).

Figure 10.1 also lists how many pure side conditions RefinedC solves automatically using its default solver and how many need at least some manual help. We count these numbers very *conservatively*: In many cases, a standard solver, like `set_solver` from std++,[2] discharges several side conditions automatically, but we still count these side conditions in "manual" since the developer has to explicitly specify that the set solver must be used. Basically, any side condition that cannot be discharged by the one default solver that we wrote—which currently only targets linear arithmetic and Coq lists—is counted as manual. This default solver can definitely be improved in the future. Finally, for each example, Figure 10.1 lists the number of lines of C code, annotations, and pure Coq reasoning for manual proofs. Importantly, there is no column for the number of lines of separation logic (Iris) reasoning since the RefinedC automation is able to handle this automatically (except for the initialization function for spinlocks, which we explain later).

[2] Coq-std++ team, *An extended "standard library" for Coq*, 2020 [Coq20].

Overall, our experience is that RefinedC's automation can handle a wide variety of low-level reasoning, requiring manual input only for example-specific pure (mathematical) side conditions and only in the more challenging examples. RefinedC's relative annotation overhead is moderate—less than 0.7 for all examples that do not involve complex side conditions (which are not the focus of RefinedC's automation at present).

#1: Common case studies. The first three examples of Figure 10.1 are case studies common to many verification tools. The verification of singly-linked lists uses the representation of partial data structures with magic wand[3] illustrated in §7.3, while the verification of queues needs a more specialized notion of list segments. Both use the first allocator of #2 below for the allocation of new nodes. The five side conditions counted here as manually discharged are actually handled automatically by `set_solver` from std++. Additionally, we verified a binary search implementation using a function pointer, and a client of it. RefinedC handles this easily since function pointer types are first class. The annotation overhead for these examples is low. In addition to annotations for loops and data structure invariants, only a single annotation (to import manual proofs) is necessary.

[3] Cao et al., "Proof Pearl: Magic Wand as Frame", 2019 [Cao+19]; Charguéraud, "Higher-order Representation Predicates in Separation Logic", 2016 [Cha16].

#2: Ownership reasoning. To evaluate RefinedC's ownership reasoning, we verified two memory allocators. These examples showcase RefinedC's expressiveness, as all necessary ownership transfers can be represented using types like `padded` (`rc::size` annotation in Figure 7.4). The thread-safe allocator uses annotations to manipulate the `spinlocked` type, similar to the allocator described in §7.2. (A third memory allocator from real-world code is covered in #5 below.)

#3: Layered vs. direct verification. A popular approach to verification of low-level code is to split the verification tasks into many layers of intermediate specifications.[4] To investigate how this layered approach works in RefinedC, we verified a binary search tree first via an intermediate functional layer, and second by directly going from C to the desired specification as a functional set. Although both approaches are viable with RefinedC, the overhead of the direct approach is significantly less than the overhead of the layered approach as it does not require defining the intermediate layer. The direct approach works well because the type system cleanly separates ownership reasoning from pure functional reasoning and all except three side conditions are automatically discharged by variants of `set_solver`.

[4] Gu et al., "Certified Concurrent Abstraction Layers", 2018 [Gu+18]; Lorch et al., "Armada: Low-Effort Verification of High-Performance Concurrent Programs", 2020 [Lor+20].

#4: Complex functional reasoning. To check whether RefinedC scales to data structures with complex functional invariants, we verified a hashmap with linear probing. Verifying linear probing is non-trivial since all keys share the same array, and one has to prove that an insertion or deletion does not affect unrelated keys. The verification uses a functional version of the probing function for stating the invariant. RefinedC reduces verification to pure reasoning about this invariant, which is discharged through manual proofs in Coq.

#5: Real-world code. Our largest case study applies RefinedC to a version of the page allocator of the Hafnium hypervisor.[5] This verification combines many of the previously mentioned techniques, and shows that RefinedC can verify real-world C code. Even though this allocator is significantly more complicated than the allocators in #2, we did not have to define any new RefinedC types to automatically handle the ownership reasoning.

[5] Hafnium, *Hafnium*, 2023 [Haf23].

#6: Concurrent abstractions. The examples in this class show that RefinedC can automatically verify fine-grained concurrent code that is out of reach for many other automatic verifiers. In particular, we use the atomic Boolean type from §9 to verify two concurrent algorithms: a spinlock and a one-time barrier. This type is abstract enough to automate the verification of the acquire and release functions of the spinlock and the barrier. The initialization function needs manual proofs where it allocates a ghost token and for instantiating one existential quantifier with a newly generated ghost name. As mentioned in §7, RefinedC also provides a `spinlocked` type, which decouples the spinlock from the resources protected by it; the typing rules for `spinlocked` require 162 lines of additional Iris proofs. Altogether, the result is a reusable spinlock abstraction, which is used by several other examples in Figure 10.1 (the first allocator of #2, and the allocator of #5).

Chapter 11

Related Work

Bedrock. Like RefinedC, the Bedrock project[1] targets foundational and mostly automatic separation logic-based verification of low-level programs. However, Bedrock is based on a custom assembly-like language and custom DSLs built on top, using macros that are verified similar to compiler passes. In contrast, RefinedC applies to existing C code that can be compiled using off-the-shelf optimizing C compilers.

Another point of difference from RefinedC is that, rather than exploiting the higher-level abstractions of a refined type system to drive automation, Bedrock encodes specifications and abstract predicates in plain separation logic, for which proof automation can be extended via custom Ltac tactics and hints for unfolding abstract predicates. However, Bedrock's hint format is less expressive than Lithium, *e.g.*, it cannot represent rules like CAS-BOOL from §9.3. Also, unlike RefinedC typing rules, Bedrock hints cannot be tied to specific program constructs and, hence, cannot be directed by program syntax. Thus, for example, the verification of a singly-linked list requires four custom hints and ∼10 lines of custom Ltac in Bedrock,[2] whereas no such extra work is required in RefinedC. (Both tools require loop invariant annotations.)

VST. VST[3] is a separation logic-based framework for verifying CompCert C programs. Users of VST deploy a set of semi-automatic tactics to build functional correctness proofs in Coq,[4] or a frontend[5] that uses source code annotation to reduce verification to a set of entailments that have to be proven in Coq. However, in both cases the user needs to manually guide the proof by performing case distinctions, applying lemmas, unfolding predicates, and instantiating existential quantifiers—tasks that RefinedC's Lithium-based automation handles automatically in most cases. As a concrete example, verification of a binary search tree similar to the one in §10 by the authors of VST[6] requires manual effort for hundreds of such proof steps, which is not the case in RefinedC. (The binary tree example in RefinedC needs manual effort only for pure side conditions.)

Foundational verification of large-scale C programs. There are several projects that perform C verification at scale, most notably seL4 and CertiKOS.

seL4[7] demonstrated the first formal proof of functional correctness of a complete, general-purpose operating-system kernel and comes with a translation-validation procedure[8] to transfer the proofs to generated

[1] Chlipala, "Mostly-Automated Verification of Low-Level programs in Computational Separation Logic", 2011 [Chl11]; Chlipala, "The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier", 2013 [Chl13]; Chlipala, "From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification", 2015 [Chl15]; Malecha et al., "Compositional Computational Reflection", 2014 [MCB14].

[2] Bedrock team, *Verification of a singly linked list*, 2015 [Bed15a].

[3] Appel, *Program Logics for Certified Compilers*, 2014 [App14]; Cao et al., "VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs", 2018 [Cao+18].

[4] Cao et al., "VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs", 2018 [Cao+18].

[5] Wang and Cao, "VST-A: A Foundationally Sound Annotation Verifier", 2019 [WC19].

[6] VST team, *Verification of a binary search tree*, 2020 [VST20].

[7] Klein et al., "seL4: Formal Verification of an OS Kernel", 2009 [Kle+09]; Klein et al., "Comprehensive Formal Verification of an OS Microkernel", 2014 [Kle+14].

[8] Sewell et al., "Translation Validation for a Verified OS Kernel", 2013 [SMK13]; Myreen, "Formal verification of machine-code programs", 2009 [Myr09].

assembly code. However, most of seL4's proofs about C code are manual and rely only on basic tactic support.[9] Later work automates some but not all of the most tedious parts.[10] This automation—and the original seL4 verification—do not support some aspects of C (such as concurrency and taking addresses of local variables) that are supported by RefinedC.

CertiKOS[11] provides the first correctness proof of a general-purpose concurrent OS kernel with fine-grained locking. CertiKOS verification is integrated with the CompCert C compiler, so the proof applies to the generated assembly code. The proof technique used (called "certified abstraction layers") is based on writing programs at different layers of abstraction and proving refinements between these layers. Refinement proofs are discharged (broadly similar to VST) by manually guiding specialized tactics in Coq. As seen in §10, RefinedC does not (in most cases) require such manual guidance in Coq, and it also supports a layer-based approach (although quite different from CertiKOS's, since RefinedC's is based on layers of types vs. layers of programs in CertiKOS). However, further work is needed in order to establish the effectiveness of RefinedC at the larger scale at which seL4 and CertiKOS have been deployed.

[9] Klein et al., "Comprehensive Formal Verification of an OS Microkernel", 2014 [Kle+14]; Winwood et al., "Mind the Gap", 2009 [Win+09].

[10] Greenaway et al., "Bridging the Gap: Automatic Verified Abstraction of C", 2012 [GAK12]; Greenaway et al., "Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain", 2014 [Gre+14].

[11] Gu et al., "Building Certified Concurrent OS Kernels", 2019 [Gu+19]; Gu et al., "Certified Concurrent Abstraction Layers", 2018 [Gu+18]; Gu et al., "Deep Specifications and Certified Abstraction Layers", 2015 [Gu+15].

Non-foundational tools for verification of C. We compare RefinedC to some of the most closely related non-foundational tools for verifying C code.

CN[12] provides a separation logic refinement type system to reason about systems code written in C. Since CN does not provide foundational proofs, it can rely on an SMT solver to automate the verification conditions generated by its LiquidTypes-based refinement type system. However, CN only considers a fixed type system and does not support modularly extending the type-system with new types and rules as RefinedC.

[12] Pulte et al., "CN: Verifying Systems C Code with Separation-Logic Refinement Types", 2023 [Pul+23].

Gillian-C[13] instantiates the Gillian platform for symbolic analysis[14] with a memory model based on CompCert's Csharpminor language. Thanks to its use of SMT solvers and implementation in OCaml, it can provide a higher degree of automation than RefinedC, especially around pure properties, but it lacks RefinedC's ability to leverage definitions and lemmas from an ambient meta-logic (*i.e.*, Coq in the case of RefinedC).

[13] Maksimovic et al., "Gillian, Part II: Real-World Verification for JavaScript and C", 2021 [Mak+21].

[14] Santos et al., "Gillian, Part i: A Multi-language Platform for Symbolic Execution", 2020 [San+20].

VCC[15] employs SMT solvers to verify C programs and has been used on large C programs in practice. However, it lacks good support for dynamic ownership reasoning. For example, a linked list predicate that supports member testing requires three ghost fields—all of which need to be updated manually in the `add` function.[16] No such ghost fields and annotations are necessary in RefinedC.

[15] Cohen et al., "VCC: A Practical System for Verifying Concurrent C", 2009 [Coh+09].

[16] VCC team, *Verification of a singly linked list*, 2016 [VCC16].

VeriFast[17] is an automated, separation logic-based verification tool for C and Java. It provides heuristics to automatically infer annotations to reduce the proof burden.[18] VeriFast's symbolic execution approach (of which only a core subset has been proven sound[19]) uses a fixed rule for each program construct, whereas RefinedC allows type-based overloading as described in §9.3. RefinedC also benefits from existing Coq libraries like std++: the binary search tree (layered) example from §10 requires roughly half the number of lines of pure reasoning compared to a similar

[17] Jacobs et al., "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java", 2011 [Jac+11].

[18] Vogels et al., "Annotation Inference for Separation Logic Based Verifiers", 2011 [Vog+11].

[19] Vogels et al., "Featherweight VeriFast", 2015 [VJP15].

example in VeriFast[20] by judicious use of existing lemmas and tactics. Other than this, the annotation burden is similar.

MatchC[21] is an automated verification tool for C based on the K framework and matching logic.[22] Its rewrite-based approach provides good automation for non-trivial pointer-manipulating programs and can be extended with new abstractions and custom rules like RefinedC. However, unlike RefinedC, these abstractions and their rules are not proven sound against a model, and must be trusted. MatchC also does not support concurrency.

Memory safety in low-level programming languages. RefinedC focuses on full functional verification of low-level programs. Much prior work[23] focuses instead on the different—and simpler—problem of automatically verifying memory safety. One popular approach[24] is to combine static and dynamic checks to enforce safety of C programs. In contrast, RefinedC targets verification without affecting the dynamic semantics of the program. Low-Level Liquid Types[25] verify memory safety of C code using a combination of refinement types and alias types.[26] The annotation overhead is low (*e.g.*, no loop invariants are required), but the goal is only memory safety. In contrast, RefinedC targets full functional verification, and thus requires more annotations but can also verify more programs. Finally, safety can also be attained by using a memory-safe language such as Vault, Cyclone, or Rust in place of C.[27] However, these languages rely on runtime checks, and—unlike RefinedC—cannot guarantee functional correctness.

Refinement and ownership type systems. Refinement types,[28] although originally developed for functional programs, have also been used for the safety and correctness of imperative code.[29] This line of work usually focuses on fully automatic type systems for relatively simple imperative languages. In contrast, RefinedC requires more annotations (*e.g.*, loop invariants), but can verify more complicated properties and supports a more realistic subset of C (including pointer arithmetic, uninitialized memory, and concurrency).

Foundational verification of fine-grained concurrent algorithms. There is an abundance of related work on foundational verification of fine-grained concurrent algorithms using interactive proofs, *e.g.*, in FCSL, VST, and Iris.[30] This line of work has focused on more challenging concurrent algorithms than the spinlock and barrier we have verified in RefinedC. In future work, we aim to investigate if we can develop types besides the atomic Boolean type (§9.3) that would enable automatic verification of more sophisticated concurrent algorithms. In particular, it would be interesting to see if ideas from Diaframe (discussed in §5) could be integrated into RefinedC to enable automated verification of more interesting concurrent algorithms.

Semantic typing. RefinedC's semantic typing approach—in particular, building a semantic model of types on top of Iris—is modeled after that of

[20] Verifast team, *Verification of a binary search tree*, 2019 [Ver19].

[21] Stefanescu, "MatchC: A Matching Logic Reachability Verifier Using the K Framework", 2014 [Ste14].

[22] Rosu and Serbanuta, "An Overview of the K Semantic Framework", 2010 [RS10]; Rosu et al., "Matching Logic: An Alternative to Hoare/Floyd Logic", 2010 [RES10].

[23] Berdine et al., "Smallfoot: Modular Automatic Assertion Checking with Separation Logic", 2005 [BCO05]; Yang et al., "Scalable Shape Analysis for Systems Code", 2008 [Yan+08].

[24] Necula et al., "CCured: Type-Safe Retrofitting of Legacy Code", 2002 [NMW02]; Condit et al., "Dependent Types for Low-Level Programming", 2007 [Con+07]; Elliott et al., "Checked C: Making C Safe by Extension", 2018 [Ell+18].

[25] Rondon et al., "Low-Level Liquid Types", 2010 [RKJ10].

[26] Rondon et al., "Liquid Types", 2008 [RKJ08]; Smith et al., "Alias Types", 2000 [SWM00].

[27] DeLine and Fähndrich, "Enforcing High-Level Protocols in Low-Level Software", 2001 [DF01]; Jim et al., "Cyclone: A Safe Dialect of C", 2002 [Jim+02]; Swamy et al., "Safe Manual Memory Management in Cyclone", 2006 [Swa+06].

[28] Freeman and Pfenning, "Refinement Types for ML", 1991 [FP91]; Xi, "Dependent ML: An Approach to Practical Programming with Dependent Types", 2007 [Xi07]; Rondon et al., "Liquid Types", 2008 [RKJ08].

[29] Rondon et al., "Low-Level Liquid Types", 2010 [RKJ10]; Bakst and Jhala, "Predicate Abstraction for Linked Data Structures", 2016 [BJ16]; Toman et al., "ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs", 2020 [Tom+20].

[30] Sergey et al., "Mechanized Verification of Fine-grained Concurrent Programs", 2015 [SNB15]; Mansky et al., "A Verified Messaging System", 2017 [MAN17]; Jung et al., "The Future is Ours: Prophecy Variables in Separation Logic", 2020 [Jun+20].

RustBelt.[31] However, the concrete design of RefinedC's type system differs from RustBelt in several key aspects: (1) RefinedC uses Mezzo-like[32] alias types instead of Rust's lifetimes and mutable references, (2) RefinedC includes refinement types in addition to ownership types, and (3) RefinedC supports automated type checking, which RustBelt does not.

[31] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language", 2018 [Jun+18a]; Jung, "Understanding and Evolving the Rust Programming Language", 2020 [Jun20].

[32] Pottier and Protzenko, "Programming with Permissions in Mezzo", 2013 [PP13].

Chapter 12

Limitations and Future Work

This part of this dissertation demonstrated the potential of refined ownership types to effectively automate the foundational verification of C code. However, RefinedC is still in its infancy and has a number of limitations that we plan to address in future work.

C idioms and features. RefinedC relies on an expert crafting typing rules to handle relevant programming idioms in the code one wishes to verify. Our evaluation shows that it is possible to come up with reusable typing rules for several common C programming idioms. However, there are C programming idioms that are not yet covered by our existing typing rules. One step in this direction is the work of Zhu et al.,[1] who show how to use RefinedC's extensible type system to reason about bitfield manipulating programs. It would be interesting to investigate RefinedC's capability for building reusable abstractions further.

Also, Caesium and the frontend lack support for some features of C. Lepigre et al.[2] extended RefinedC and Caesium with support for integer pointer casts. Their model is designed to by used by a verification tool and proven sound against the PNVI-ae-udi model of Memarian et al.[3] Extending Caesium with other features of C like floats would be interesting future work.

RefinedC currently does not support reasoning about external function calls and input-output behavior of programs. We believe that the automation provided by Lithium can also be useful for such I/O verification, and we plan to integrate RefinedC with DimSum (presented in Part IV) in future work.

Pure automation. So far, we have focused mainly on automating the *separation logic* aspects of reasoning. We additionally support automation for several domains of *pure* reasoning by leveraging existing solvers for *e.g.*, linear arithmetic, sets, and multisets, but this support can certainly be extended further.

Liveness properties. RefinedC only verifies partial, not total, correctness. This is mainly due to Iris's focus on verifying safety properties. However, recent work enables termination verification in Iris using transfinite step-indexing.[4] It would be interesting to combine transfinite step-indexing with RefinedC and Lithium to achieve automated and foundational verification of liveness properties.

[1] Zhu et al., "BFF: Foundational and Automated Verification of Bitfield-Manipulating Programs", 2022 [Zhu+22].

[2] Lepigre et al., "VIP: Verifying Real-World C Idioms with Integer-Pointer Casts", 2022 [Lep+22].

[3] Memarian et al., "Exploring C Semantics and Pointer Provenance", 2019 [Mem+19].

[4] Spies et al., "Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic", 2021 [Spi+21].

Part III

Islaris

Chapter 13

Introduction

Program verification can be applied at many levels, from high-level languages to low-level assembly or machine code. Low-level code verification is desirable for three reasons. First, some critical code manipulates architectural features that are not exposed in higher-level languages, *e.g.*, to access system registers to install exception vector tables, or to configure address translation; this is necessarily written in assembly. Second, machine code is the form in which programs are actually executed, so a verification can be grounded on the architecture semantics, without needing trust or verification of any compilation or assembly steps. One can moreover verify the machine code after any modifications introduced by linking or initialization (perhaps parametrically w.r.t. these). Third, some code is written in assembly for performance reasons.

In low-level code verification, it remains a grand challenge to develop tools that are demonstrably sound w.r.t. the underlying architecture and support reasoning about all of it, including all systems features. There are several aspects to this. One is the relaxed-memory concurrency exhibited by modern hardware. For this, the underlying models for user code have been clarified;[1] work on systems concurrency is in progress;[2] and researchers are starting to build low-level-code verifications targeting relaxed memory, *e.g.*, for hypervisors.[3]

Another key aspect—and the one we focus on here—is ensuring fidelity and completeness w.r.t. the underlying *instruction-set architecture* (ISA), the sequential semantics of machine instructions. Until recently, the only option was to hand-write an ISA semantics, as several verification projects did, each for the fragment of the ISA they needed.[4] These typically cover only a small user-level fragment of the ISA, are simplified in various ways, and have, at best, only limited validation with respect to the architectural intent or hardware implementations. For x86, there is a handwritten larger fragment in ACL2,[5] and empirical and handwritten models.[6] Others add models of some systems aspects, but similarly without tight connections to production architecture definitions.

In contrast to the above, recent work has established sequential ISA models for Armv8-A and RISC-V, that are both comprehensive—complete enough to boot an operating system or hypervisor—and authoritative. These are expressed in the Sail ISA definition language.[7] For Armv8-A, the Sail model is automatically derived from the Arm-internal model and tested against the Arm-internal validation suite, while for RISC-V the handwritten Sail model has been adopted as the official formal specification

[1] Sewell et al., "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors", 2010 [Sew+10]; Sarkar et al., "Understanding POWER Multiprocessors", 2011 [Sar+11]; Alglave et al., "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory", 2014 [AMT14]; Pulte et al., "Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8", 2018 [Pul+18]; Arm, *Arm Architecture Reference Manual. Armv8, for A-profile architecture profile*, 2021 [Arm21]; *The RISC-V Instruction Set Manual. Volume I: User-Level ISA; Volume II: Privileged Architecture*, 2017 [17].

[2] Simner et al., "ARMv8-A System Semantics: Instruction Fetch in Relaxed Architectures", 2020 [Sim+20]; Raad et al., "Extending Intel-x86 Consistency and Persistency: Formalising the Semantics of Intel-x86 Memory Types and Non-Temporal Stores", 2022 [RMV22]; Simner et al., "Relaxed virtual memory in Armv8-A", 2022 [Sim+22].

[3] Li et al., "A Secure and Formally Verified Linux KVM Hypervisor", 2021 [Li+21].

[4] See *e.g.*, Ama+13; Ler06; Šev+13; Mor+12; Myr09; Ken+13; JBK13; Sar+09; Gra+15; Flu+16; AMT14; FM10; LS09.

[5] Goel et al., "Engineering a Formal, Executable x86 ISA Simulator for Software Verification", 2017 [GJK17].

[6] Degenbaev, "Formal Specification of the x86 Instruction Set Architecture", 2012 [Deg12]; Heule et al., "Stratified Synthesis: Automatically Learning the x86-64 Instruction Set", 2016 [Heu+16]; Dasgupta et al., "A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture", 2019 [Das+19].

[7] Armstrong et al., "ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS", 2019 [Arm+19a].

by RISC-V International. This makes these attractive foundations for verification, providing high confidence that they accurately capture the architecture (and hence that the results of verification will hold above correct hardware implementations), and enabling verification about all aspects of the sequential ISA, especially the systems aspects that are key to security.

However, that fidelity and coverage also makes these models intimidatingly large and complex, and only sometimes practical for mechanized proof. The Sail Armv8.5-A and RISC-V models are 113k and 14k non-whitespace lines of specification, respectively. Sail generates Isabelle and Coq versions of these definitions. For Armv8-A, the former has been used for some metatheory,[8] but not for program verification, and in the Coq version even simple definition unfoldings take an unreasonably long time or fail to terminate.

To see how this complexity arises, consider the seemingly simple Armv8-A `add sp,sp,64` instruction, adding 64 to the stack-pointer register. Some handwritten Arm semantics describe this in a single line,[9] but its full Sail definition spans 146 lines in 9 functions, excerpted in Figure 14.1. These do much more than just compute the addition: they compute arithmetic flags (discarded by this particular `add` instruction); they support subtraction as well as addition (again irrelevant for this instruction); they support other registers; and `sp` is in fact a banked family of registers, selected based on the current exception level register value. A yet more extreme example is a "simple" `ldrb` instruction to load a byte. This involves over 2000 lines of specification, even without address translation, for alignment checks, big/little endianness, tagged memory, different address sizes and exception levels, and the store and prefetch instructions that are specified simultaneously.

The challenge we face, therefore, is how one can reason above such models while avoiding up-front idealization, so that we retain the ability to reason about the whole architecture, and the confidence in the authoritative model.

In this part of the dissertation, we present Islaris, a novel approach to machine-code verification that achieves the above. Our key insight is to realize that the verification problem can be split into two subtasks, separating the *irrelevant* complexity from the *inherent* complexity, so that each can then be solved by techniques well suited for the respective task: SMT-based symbolic evaluation, and a mechanized program logic.

The first step is to realize that, when verifying a concrete program under specific assumptions, many aspects of the ISA definition are irrelevant, because they do not influence the results of instructions or are ruled out by the system configuration. To handle this irrelevant complexity, we leverage and extend the Isla symbolic evaluation tool for Sail ISA specifications.[10] Isla takes an opcode and SMT constraints, *e.g.*, that the exception-level register has a specific value or some general-purpose register is aligned, and symbolically evaluates the Sail model using an SMT solver. It produces a *trace* of the instruction's register and memory accesses, constrained by SMT formulas. Crucially, this can be much

[8] Armstrong et al., "ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS", 2019 [Arm+19a]; Bauereiss et al., "Verified Security for the Morello Capability-enhanced Prototype Arm Architecture", 2022 [Bau+22].

[9] CompCert team, *CompCert Arm semantics*, 2023 [Com23].

[10] Armstrong et al., "Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models", 2021 [Arm+21].

simpler than the full Sail definition, without irrelevant and unreachable parts, and is in a much simpler language.

That leaves the inherent complexity of verification, typically including address and memory manipulations, higher-order reasoning with code-pointers, reasoning about the relevant aspects of the systems architecture, and modular reasoning about user-defined specifications. Islaris addresses these with a higher-order separation logic for the Isla traces that produces machine-checked proofs, based on Iris. The key challenge is designing proof automation that makes the verification practical. Here, Islaris adapts Lithium that was originally designed for automating for the RefinedC type system. In particular, we realize that Lithium's efficiency can be retained even without the type information relied on by RefinedC, by using the separation logic context to guide proof search. Overall, we obtain a level of proof automation comparable to previous foundational approaches,[11] but for full ISA semantics rather than a simple intermediate language.

[11] Chlipala, "Mostly-Automated Verification of Low-Level programs in Computational Separation Logic", 2011 [Chl11].

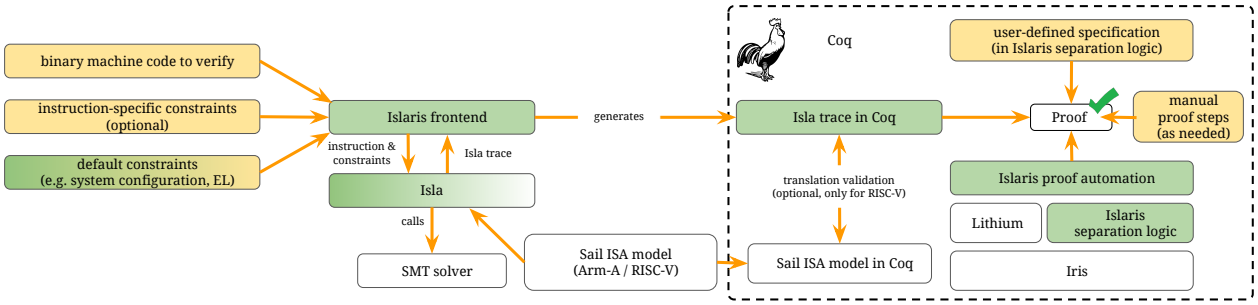


Figure 13.1: The Islaris workflow. White: from previous work. Green: new in this part of the dissertation. Yellow: provided by the Islaris user.

Overview. Figure 13.1 shows an overview of the Islaris workflow. First, the user passes the machine code to verify together with suitable constraints on the system state to the Islaris frontend, which invokes Isla to generate a trace describing the effects of the instructions based on the Sail ISA model. The generated trace has already been simplified by Isla, by pruning parts of the ISA specification that cannot be reached under the given constraints (Isla uses symbolic execution and an SMT solver for this pruning). The frontend outputs a deep embedding of this trace in Coq, which is then verified against a user-written specification using the Islaris proof automation, together with manual Coq proof if needed. For RISC-V, we also provide infrastructure to prove the Isla trace correct against the Coq ISA model generated directly from Sail.

Contributions. Our overarching contribution is this new approach for machine-code verification above complete and authoritative real-world ISA specifications, including systems features. The Islaris combination of Isla-based symbolic execution with an Iris-based program logic and Lithium-based proof automation gives us practical tooling for verification above such models, which we demonstrate on a range of examples. All this is generic in the actual Sail model, applying equally to Armv8-A and RISC-V. We give:

- Operational semantics for the Isla trace language (§15).

- An Iris-based separation logic for Isla traces with Lithium-based proof automation (§16).

- Translation validation infrastructure for RISC-V Isla traces, proving them correct with respect to the Coq model generated from Sail directly for RISC-V (§17).

- Demonstration that Islaris is able to handle Armv8-A and RISC-V machine code exercising a wide range of systems features (and interacting with many system registers), including installing and calling an exception vector, compiled C code using inline assembly and function pointers, using memory-mapped IO to interact with a UART device, and code that is parametric on a relocation address offset; the last of these is part of an exception handler from the production pKVM hypervisor under development at Google (§18).

Non-goals and limitations. The main contribution of Islaris is to make it possible to reason above authoritative ISA semantics (especially the full Armv8-A ISA model) without upfront idealization, which has not been done previously. This is important in two contexts: for the lowest, security-critical, layers of a software stack, and as a more solid foundation for large-scale verification of higher layers of a software stack. Islaris focuses on the first. When the critical code is short, e.g., the pKVM exception-handler dispatch described in §18, Islaris as presented here can be applied directly. For the second, Islaris can provide a useful building block. However, demonstrating the use of Islaris on higher layers of a software stack is future work; Islaris as presented in this dissertation is not intended or claimed to replace general-purpose verification tools for such.

Islaris targets the verification of concrete machine code, where one has specific (or highly constrained) opcodes in hand, together with constraints on the system state, as the Isla symbolic evaluation can provide substantial simplification in such situations. For proving facts about all the instructions of an architecture, one would typically want a different approach, *e.g.*, as described by Armstrong et al.[12] or Bauereiss et al.[13] For proving facts about a compiler, one might want to prove correctness of a simplified model, tuned to the subset of instructions it generates. In some cases, this could also be done using Islaris, but we do not explore this use-case here.

[12] Armstrong et al., "ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS", 2019 [Arm+19a].

[13] Bauereiss et al., "Verified Security for the Morello Capability-enhanced Prototype Arm Architecture", 2022 [Bau+22].

Ideally, the trusted computing base (TCB) would only include the ISA definitions and one proof assistant kernel. The basic Islaris approach adds Isla and the SMT solver to the TCB (but not the Islaris separation logic, which produces machine-checked proofs). We consider this a reasonable price to pay for the benefits Islaris provides. For additional assurance, we have explored post-hoc validation of the Isla output with respect to the Sail-generated Coq semantics (see §17). We have done this for RISC-V; for Armv8-A, the model size makes it challenging. *Complete* assurance is of course impossible: even the Arm-internal ISA definition, while well-validated in many ways, is surely not perfect; there is the possibility of

error in the Sail-to-Coq translations; and full verification of the underlying hardware is not yet feasible.

The other main limitation is that Islaris currently assumes single-threaded execution. This is not inherent to our approach—Isla's output is generic in the underlying memory model, and supporting a sequentially consistent concurrency semantics would not be hard. However, supporting the Armv8-A or RISC-V relaxed-memory concurrency models requires a more sophisticated separation logic, the subject of active research. Islaris also does not currently support self-modifying code or address translation, which involve additional forms of relaxed-memory concurrency, likewise subjects of active research[14] (our underlying ISA semantics includes translation-table walks, but here we only use machine configurations that turn translation off). Finally, we have focused so far on 64-bit little-endian cases, and on small but tricky examples; scaling remains future work.

[14] Simner et al., "ARMv8-A System Semantics: Instruction Fetch in Relaxed Architectures", 2020 [Sim+20]; Simner et al., "Relaxed virtual memory in Armv8-A", 2022 [Sim+22].

Chapter 14

Overview of the Islaris Approach

In this chapter, we give a high-level presentation of the Islaris approach to machine-code verification. We explain how the complexity of raw Sail models is made manageable using the Isla symbolic evaluator in §14.1, and then the rest of the chapter (starting in §14.2) shows how Islaris builds a modular verification framework on Isla.

14.1 Background: Symbolic Execution with Isla

```
function clause decode64
 ((_:bits(1) @ 0b0010001 @ _:bits(24) as opcode) if ...)={
 Rd:bits(5)=opcode[4 .. 0]; Rn=...; imm2=...; sf=...; ...
 integer_arithmetic_addsub_immediate_decode(Rd,Rn,...)}

function integer_arithmetic_addsub_immediate_decode(...)={
 let 'd = UInt(Rd); ...
 let 'datasize = if sf == 0b1 then 64 else 32;
 imm : bits('datasize) = undefined : bits('datasize); ...
 match shift {
  0b00 => { imm = ZeroExtend(imm12, datasize) },
  0b01 => { imm = ZeroExtend(imm12 @ Zeros(12), datasize)},
  0b10 => { throw(Error_See("ADDG, SUBG")) },
  0b11 => { ReservedValue() } };
 integer_arithmetic_addsub_immediate(d,datasize,imm,...) }

function integer_arithmetic_addsub_immediate (...) = {
 let op1 : bits('datasize) =
  if eq_int(n, 31) then aget_SP(__id(datasize))
                   else aget_X(__id(datasize), n); ...
 if sub_op then { op2 = not_vec(op2); carry_in = 0b1 }
           else { carry_in = 0b0 }
 let (tup__0, tup__1) = AddWithCarry(op1,op2,carry_in) in
 ...
 if setflags then
  {PSTATE={PSTATE with N=vector_subrange_A(nzcv,3,3)};...};
 if and_bool(eq_int(d,31),not_bool(setflags)) then
  { aset_SP(result) } else { aset_X(d, result) } }

function AddWithCarry (x,y,carry_in) = { ... }
```

Figure 14.1: Excerpts of the 146-line Sail definition of `add sp,sp,64`.

As already discussed in §13, in a real-world architecture the semantics of even seemingly simple instructions like an addition can be surprisingly

```
1 (trace
2  (assume-reg |PSTATE| ((_ field |EL|)) #b10)
3  (assume-reg |PSTATE| ((_ field |SP|)) #b1)
4  (read-reg |PSTATE| ((_ field |SP|)) (_ struct(|SP| #b1)))
5  (read-reg |PSTATE| ((_ field |EL|)) (_ struct(|EL| #b10)))
6  (declare-const v38 (_ BitVec 64))
7  (read-reg |SP_EL2| nil v38)
8  (define-const v61 (bvadd ((_ extract 63 0)
9   ((_ zero_extend 64) v38)) #x0000000000000040))
10 (write-reg |SP_EL2| nil v61)
11 (declare-const v62 (_ BitVec 64))
12 (read-reg |_PC| nil v62)
13 (define-const v63 (bvadd v62 #x0000000000000004))
14 (write-reg |_PC| nil v63))
```

Figure 14.2: Isla trace of `add sp,sp,64` (opcode `0x910103ff`).

complex. For example, consider the excerpt of the Sail semantics for the `add sp,sp,64` instruction in Figure 14.1. The `decode64` entry point decodes an opcode and dispatches to many auxiliary functions expressing the register and memory accesses of its semantics. This size makes direct verification against these semantics challenging, which is why Islaris uses Isla. Isla[1] takes as input an opcode and a collection of SMT constraints on the machine state, and symbolically evaluates the Sail model w.r.t. those, pruning unreachable branches using an SMT solver. The result of such a symbolic evaluation for (the opcode of) `add sp,sp,64` is the *trace* in Figure 14.2. This describes the behavior of the instruction using a small set of primitive constructs. Ignoring lines 2-5 for the moment, this trace first reads the value `v38` from the stack pointer `SP_EL2`, on lines 6-7. This read is expressed by first declaring a new 64-bit bitvector variable `v38` on line 6, and then setting it to the value of the `SP_EL2` register on line 7. Then the trace computes `v61` as the bitvector addition of `v38` and 64 (`0x40`). It might seem curious that the addition is computed on 128-bit integers (by first zero-extending `v38`) from which the lowest 64 bits are extracted as the result (via `(_ extract 63 0)`); this is a vestige of the fact that the model also computes whether this addition overflows, used for other variants of `add`, but discards that in this case. Finally, the result in `v61` is stored back into `SP_EL2`, and the `_PC` register is updated to point to the next instruction. This example shows that Isla can condense the 100+ executed lines of the original model down to the operations that one would expect of this `add` instruction: reading the stack pointer, computing the addition, writing the result back, and incrementing the program counter.

[1] Armstrong et al., "Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models", 2021 [Arm+21].

Isla has also simplified away the complexity from the banked stack pointer registers (which is not covered in some handwritten models, notably excepting the work by Fox[2]): Armv8-A has distinct exception levels for user, kernel, hypervisor, and monitor execution, and a stack pointer register for each. The stack pointer used by `add sp,sp,64` is selected based on the `EL` and `SP` fields of the `PSTATE` register, where the first gives the current exception level and the second toggles whether the multiple stack pointers are enabled (when `SP=0` all exception levels use the stack pointer of exception level 0, `SP_EL0`). Typically, the values of `EL` and `SP`

[2] Fox, "Improved Tool Support for Machine-Code Decompilation in HOL4", 2015 [Fox15].

are fixed for a given piece of code, and thus it is clear which stack pointer is used. Isla can exploit this knowledge to simplify the trace by adding constraints to the symbolic execution. Concretely, the trace in Figure 14.2 was generated with the constraints `EL=2` and `SP=1` (for code running at exception level 2 with multiple stack pointers enabled). As a consequence, the trace directly uses the stack pointer of exception level 2, `SP_EL2`, and the reads of `SP` and `EL` on lines 4-5 have been simplified to specify their concrete known values. Without these constraints, the trace distinguishes five cases (via the mechanism described in §14.4): one for `SP=0`, and one for each of the four exception levels when `SP=1`. The assumptions used by Isla are recorded in the trace via `assume-reg` on lines 2-3. These become proof obligations during verification, so one has to prove that `SP` and `EL` have their assumed values.

$$
\begin{array}{rcll}
e & ::= & v \mid \mathsf{not}(e) \mid \mathsf{bvadd}(e_1, e_2) \mid \ldots & \text{(SMT-Expr)} \\
v & ::= & b \mid \mathsf{true} \mid \mathsf{false} \mid x \mid \ldots & \text{(Val)} \\
r & ::= & \rho \mid \rho.\mathsf{i} & \text{(Reg)} \\
\tau & ::= & \mathsf{BitVec}(n) \mid \mathsf{Boolean} \mid \ldots & \text{(Type)} \\
j & ::= & \mathsf{ReadReg}(r, v) \mid \mathsf{WriteReg}(r, v) & \text{(Event)} \\
 & & \mid \mathsf{ReadMem}(v_d, v_a, n) \mid \mathsf{WriteMem}(v_a, v_d, n) & \\
 & & \mid \mathsf{AssumeReg}(r, v) \mid \mathsf{DeclareConst}(x, \tau) & \\
 & & \mid \mathsf{DefineConst}(x, e) \mid \mathsf{Assert}(e) \mid \mathsf{Assume}(e) & \\
t & ::= & [] \mid j :: t \mid \mathsf{Cases}(t_1, \ldots, t_n) & \text{(Trace)}
\end{array}
$$

Figure 14.3: Syntax of the Isla trace language (ITL).

Isla trace language. The Sail ISA definition language is designed to be as simple as possible while still supporting readable definitions of full-scale ISAs, but it is still relatively complex, with a rich type structure (including lightweight dependent types for bitvector lengths) and complex control flow (first-order functions, pattern matching, and loops). In contrast, the Isla trace language, with syntax in Figure 14.3 (as adapted for Islaris, and typeset in the mathematical form we use later), is simple: traces t are trees of events j[3] —register and memory accesses, augmented by declarations and definitions of SMT constants, and assertions, assumptions, and a $\mathsf{Cases}()$ construct for branching (explained in §14.4). We have already seen most of the trace language in Figure 14.2. For example, $\mathsf{ReadReg}(\mathsf{R0}, v)$ corresponds to `(read-reg |R0| nil v)`, and $\mathsf{DefineConst}(x, e)$ to `(define-const x e)`. Events rely on SMT-lib expressions e, values v containing bitvectors b and Booleans, register names r, and value types τ.

[3] Note that these ITL events are not related to DimSum's notion of events described in Part IV.

14.2 Our Contribution: Islaris

After seeing how Isla can generate specialized traces for single instructions, we now describe how we use that in modular verification for machine code. §14.3 describes the Islaris separation logic for reasoning about Isla traces; §14.4 shows how Islaris handles branching; §14.5 discusses how complete functions are verified, with a simple `memcpy` example; §14.6 explains how Islaris can reason equally well about systems code, *e.g.*, installing and calling an Armv8-A exception vector table; and §14.7 demonstrates that Islaris is not specific to Armv8-A but can also be used for RISC-V.

14.3 Islaris Separation Logic

$$\textsc{hoare-declare-const}\ \frac{\forall v \in \tau.\ \{P\}\ t[v/x]}{\{P\}\ \mathsf{DeclareConst}(x,\tau) :: t} \qquad \textsc{hoare-define-const}\ \frac{e \downarrow v \qquad \{P\}\ t[v/x]}{\{P\}\ \mathsf{DefineConst}(x,e) :: t} \qquad \textsc{hoare-read-reg}\ \frac{\{r \mapsto_R v' * v = v' * P\}\ t}{\{r \mapsto_R v' * P\}\ \mathsf{ReadReg}(r,v) :: t}$$

$$\textsc{hoare-assume-reg}\ \frac{v = v' \qquad \{r \mapsto_R v' * P\}\ t}{\{r \mapsto_R v' * P\}\ \mathsf{AssumeReg}(r,v) :: t} \qquad \textsc{hoare-write-reg}\ \frac{\{r \mapsto_R v * P\}\ t}{\{r \mapsto_R v' * P\}\ \mathsf{WriteReg}(r,v) :: t} \qquad \textsc{hoare-cases}\ \frac{\forall t \in \bar{t}.\ \{P\}\ t}{\{P\}\ \mathsf{Cases}(\bar{t})}$$

$$\textsc{hoare-assert}\ \frac{e \downarrow v \qquad \{P * v = \mathsf{true}\}\ t}{\{P\}\ \mathsf{Assert}(e) :: t} \qquad \textsc{hoare-assume}\ \frac{e \downarrow \mathsf{true} \qquad \{P\}\ t}{\{P\}\ \mathsf{Assume}(e) :: t} \qquad \textsc{hoare-instr}\ \frac{\{\mathsf{PC} \mapsto_R a * \mathsf{instr}(a,t) * P\}\ t}{\{\mathsf{PC} \mapsto_R a * \mathsf{instr}(a,t) * P\}\ []}$$

$$\textsc{hoare-instr-pre}\ \frac{P \mathrel{-\!\!*} Q}{\{\mathsf{PC} \mapsto_R a * a\ @@\ Q * P\}\ []} \qquad \textsc{instr-pre-intro}\ \frac{\{\mathsf{PC} \mapsto_R a * Q * P\}\ t}{\mathsf{instr}(a,t) * P \vdash a\ @@\ Q}$$

Figure 14.4: Key rules of the Islaris separation logic.

The core of Islaris is the Islaris separation logic for reasoning about Isla traces. We present the logic using a Hoare double $\{P\}\ t$, which asserts that the Isla trace t is safe assuming the precondition P (technically, Islaris proves more than safety; see §16.2). Hoare doubles are commonly used in Hoare logics for assembly languages,[4] as the postconditions of Hoare triples are difficult to interpret with assembly's unstructured indirect jumps.

We now explain how we verify the addition to the `SP_EL2` register on lines 6-10 of Figure 14.2—the following implication, where t_{SP} comprises those four Isla trace events:

$$\{\mathsf{SP\_EL2} \mapsto_R (b + 64)\}\ t \ \Rightarrow\ \{\mathsf{SP\_EL2} \mapsto_R b\}\ t_{SP} +\!\!+ t$$

Intuitively, assuming that $\mathsf{SP\_EL2}$ initially contains the 64-bit bitvector b, we have to show that after those four trace events, $\mathsf{SP\_EL2}$ contains $b+64$, where $(+)$ is 64-bit bitvector addition (observe how the precondition on the left of the implication acts like a postcondition). Note that, similar to Myreen and Gordon,[5] the Islaris separation logic uses a points-to predicate $r \mapsto_R v$ for asserting that register r contains the value v. This is useful for dealing with the large number of registers in the full Armv8-A model, as irrelevant registers can easily be framed away.

[4] Chlipala, "Mostly-Automated Verification of Low-Level programs in Computational Separation Logic", 2011 [Chl11]; Jensen et al., "High-Level Separation Logic for Low-Level Code", 2013 [JBK13].

[5] Myreen and Gordon, "Hoare Logic for Realistically Modelled Machine Code", 2007 [MG07].

To prove this implication, we first verify the read of the SP_EL2 register in two steps. First, the declaration of the v38 variable on line 6 is handled by hoare-declare-const (Figure 14.4), which non-deterministically chooses a bitvector value v to substitute for v38. This rule uses $v \in \tau$ to assert that the value v has type τ (here, that v is a 64-bit bitvector). Then, hoare-read-reg uses $\mathsf{SP\_EL2} \mapsto_R b$ to determine that v must be equal to b, *i.e.*, it provides $v = b$ as an assumption for the following proof.

In contrast, in hoare-assume-reg, $v = v'$ is an obligation. This use of "assume" might seem counter-intuitive, but it makes sense from the perspective of Isla: AssumeReg is an *assumption* used by Isla's symbolic execution. The same applies to the names of Assert and Assume discussed later.

The rest of the verification is straightforward: on line 8, `define-const` is handled by hoare-define-const which computes $b + 64$ and, after some simplification, substitutes it for v61. Finally, the write of this value to SP_EL2 is verified using hoare-write-reg.

Islaris proof automation. Applying these proof steps by hand quickly becomes quite tedious, especially for more complex instructions with many events. Islaris thus provides proof automation that automatically completes the verification described above. We describe the automation in §16.3.

14.4 Intra-instruction Branching

The Sail semantics for a single instruction typically involves many Sail-language control-flow choices, *e.g.*, to select among the Arm stack-pointer registers as mentioned in §14.1. In many cases, these are resolved by the assumed constraints, and the instruction's behavior can be represented by a linear trace. But what if this is not the case? The canonical examples are conditional-jump instructions such as `beq -16`, jumping 16 bytes backwards if the zero flag is set, whose semantics include a Sail-level branch determined by the flag register (which is usually written by a preceding `cmp` instruction).

The Isla trace of `beq -16` is shown in Figure 14.5 (simplified for presentation to remove assumptions about nine different system registers). It reads the zero flag (PSTATE.Z) on line 3 and computes the branching condition in v37 on line 4 (*i.e.*, whether PSTATE.Z is set). The `cases` on line 5 expresses the control-flow choice by giving two subtraces. The subtraces begin with *assertions* about their respective branch conditions. The first asserts on line 7 that v37 is true (*i.e.*, the zero flag is set) and subtracts 16 from _PC (expressed as addition of `0xfffffffffffffff0` in 64-bit arithmetic). The second subtrace asserts on line 14 that v37 is false (*i.e.*, the zero flag is not set), and sets _PC to the address of the next instruction.

During verification, the `cases` construct is handled by hoare-cases, which requires verifying the subtraces independently. In this rule, both branches use the full separation logic precondition P, since the actual execution will follow only one branch. The `assert`s within the two branches

```
(trace
  (declare-const v27 (_ BitVec 1))
  (read-reg |PSTATE| ((_ field |Z|)) (_ struct (|Z| v27)))
  (define-const v37 (= v27 #b1))
  (cases
    (trace
      (assert v37)
      (declare-const v38 (_ BitVec 64))
      (read-reg |_PC| nil v38)
      (define-const v39 (bvadd v38 #xfffffffffffffff0))
      (define-const v52 v39)
      (write-reg |_PC| nil v52))
    (trace
      (assert (not v37))
      (declare-const v38 (_ BitVec 64))
      (read-reg |_PC| nil v38)
      (define-const v39 (bvadd v38 #x0000000000000004))
      (write-reg |_PC| nil v39))))
```

Figure 14.5: Isla trace of `beq -16` (simplified).

```
void memcpy(unsigned char *d,
            unsigned char *s,
            size_t n) {
  for (size_t i = 0; i < n;
    i++) {
    d[i] = s[i];
  }
}
```

```
memcpy:cbz  x2, .L1      ; if(x2 == 0) goto .L1;
       mov  x3, 0        ; x3 = 0;
.L3:   ldrb w4, [x1,x3]  ; w4 = *(x1 + x3);
       strb w4, [x0,x3]  ; *(x0 + x3) = w4;
       add  x3, x3, 1    ; x3 = x3 + 1;
       cmp  x2, x3       ; (with next line)
       bne  .L3          ; if(x2 != x3) goto .L3;
.L1:   ret               ; return
```

```
memcpy: beqz a2, .L2
.L1:    lb   a3, 0(a1)
        sb   a3, 0(a0)
        addi a2, a2, -1
        addi a0, a0, 1
        addi a1, a1, 1
        bnez a2, .L1
.L2:    ret
```

Figure 14.6: C implementation of a `memcpy`-like function (first column) together with Arm assembly (second column, compiled with GCC 11.2 `-O2`) and RISC-V assembly (third column, compiled with Clang 13.0.0 `-O2`). We actually verify the machine-code versions of this assembly. For readability, the Arm assembly is annotated with a simplified pseudocode version of its semantics.

are verified using HOARE-ASSERT. This rule provides the respective branching condition as an assumption within each branch (similar to HOARE-READ-REG). Overall, HOARE-CASES combined with HOARE-ASSERT works like the standard rule for an if-then-else in other program logics. The rest of the trace is verified using the rules explained in §14.3.

All conditional execution is expressed using such `cases`, with unconstrained non-determinism over subtraces, followed by `assert`s providing additional assumptions implied by the choice of the case.

14.5 Verification of a Complete C Function: `memcpy`

So far, we have discussed how Islaris reasons about single instructions. Next, we turn to code containing multiple instructions. We illustrate this on the naive C `memcpy` implementation in Figure 14.6, compiled to Arm using GCC.

Our goal is to show that the `memcpy` implementation satisfies the specification in Figure 14.7. Lines 1, 2 of the specification encode the precondition on the registers used by `memcpy`. Following the Armv8-A ABI C calling convention, `x0`, `x1`, and `x2` contain the arguments `d`, `s`, and `n`; `x3` and `x4` are scratch registers; and `x30` contains the return address r. Line 3 states that `memcpy` also requires ownership of standard system registers and the flags registers (like `PSTATE.Z`). This is encoded using the `reg_col(...)` predicate, which is shorthand for a collection of register points-to assertions (described further in §16.1).

$$
\begin{aligned}
&\mathsf{memcpy\_spec} \triangleq \exists s\, d\, n\, r\, B_s\, B_d.\\
&\quad \mathsf{x0} \mapsto_R d * \mathsf{x1} \mapsto_R s * \mathsf{x2} \mapsto_R n * && (1)\\
&\quad \mathsf{x3}, \mathsf{w4} \mapsto_R \_ * \mathsf{x30} \mapsto_R r * && (2)\\
&\quad \mathsf{reg\_col}(\mathsf{sys\_regs}) * \mathsf{reg\_col}(\mathsf{CNVZ\_regs}) * && (3)\\
&\quad s \mapsto^*_M B_s * d \mapsto^*_M B_d * n = |B_s| * n = |B_d| * && (4)\\
&\quad r \mathbin{@@} (&& (5)\\
&\qquad s \mapsto^*_M B_s * d \mapsto^*_M B_s * && (6)\\
&\qquad \mathsf{x0}, \mathsf{x1}, \mathsf{x2}, \mathsf{x3}, \mathsf{w4}, \mathsf{x30} \mapsto_R \_ * && (7)\\
&\qquad \mathsf{reg\_col}(\mathsf{sys\_regs}) * \mathsf{reg\_col}(\mathsf{CNVZ\_regs})) && (8)
\end{aligned}
$$

Figure 14.7: Specification of `memcpy` (Arm assembly version).

Finally, Line 4 asserts that the pointers s and d point to memory containing the lists of bytes B_s and B_d of length n, using the points-to predicate for arrays ($\mapsto^*_M$) (see also §16.1). The rest of the specification, starting on line 5, describes the postcondition: `memcpy` ensures that after it is done, the bytes B_s stored in s have been copied to d (Line 6), and it returns ownership of the registers mentioned in the precondition (Lines 7, 8). The r @@ P assertion used to state the postcondition is described below.

Inter-instruction reasoning. Let us now take a step back to see how Islaris bridges the verification between multiple instructions. Consider the rules for $\{P\}\,[]$, *i.e.*, for the empty trace reached after having fully executed an instruction. There are two ways to proceed.

First, if the Isla trace of the next instruction is known, verification directly continues with this trace. This is encoded in HOARE-INSTR: if the PC register contains the address a at the end of an instruction, and one knows that instruction memory at a contains an instruction with Isla trace t (encoded via $\mathsf{instr}(a, t)$), the verification continues with t.

Second, if the code starting at the next instruction has been verified w.r.t. a precondition Q, it is enough to prove Q. This is encoded in HOARE-INSTR-PRE using a @@ Q, which asserts that the instruction at address a has been verified assuming precondition Q (the assertion a @@ Q is inspired by Chlipala[6]). The assertion can be established from $\mathsf{instr}(a, t)$ by proving a Hoare double for t as in INSTR-PRE-INTRO. This assertion is used in Figure 14.7, where the postcondition of `memcpy` is represented as the "Q" of this assertion, *i.e.*, as the precondition of `memcpy`'s continuation. The verification of `ret` on line 8 uses HOARE-INSTR-PRE, and thus establishes the postcondition.

[6] Chlipala, "Mostly-Automated Verification of Low-Level programs in Computational Separation Logic", 2011 [Chl11].

Verification of `memcpy`. The main task of the verification of the `memcpy` function is to find a loop invariant I for the code between `.L3` and `.L1`. Here, we use the invariant that the first m bytes, where m is the value of x3, have already been copied from s to d, and the remaining bytes of d are unchanged. With this invariant I, we establish .L3 @@ I. The proof can

```
1 .org 0x80000
2 _start:             ; *** initialisation at EL2 ***
3   mov x0, 0xa0000
4   msr vbar_el2, x0 ; Install exception vector
5   mov x0, 0x80000000
6   msr hcr_el2, x0  ; Hypervisor config: aarch64 at EL1
7   mov x0, 0x3c4
8   msr spsr_el2, x0 ; EL1 config (use SP_EL0, no interupts)
9   mov x0, 0x90000
10  msr elr_el2, x0  ; Write EL1 start address to ELR_EL2
11  eret             ; Simulate an "exception return"
12 .org 0x90000
13 enter_el1:         ; *** calling the vector from EL1 ***
14  mov x0, xzr      ; Zero out x0.
15  hvc 0            ; Perform a hypervisor call
16  b .              ; Hang forever in a loop
17 .org 0xa0000
18 el2_exception_vector:;*** the exception vector table ***
19  ...
20  ; Synchronous - Lower EL with AArch64
21  mov x0, 42       ; Put 42 in x0
22  eret             ; Return from exception
```

Figure 14.8: Install and use an exception vector.

assume that this assertion holds for later iterations of the loop, thanks to step-indexing in the underlying Iris logic.

The proof is almost completely automated by the Islaris proof automation. The proof automation handles all separation logic reasoning for the 169 events of the Isla traces in 9 seconds, and most generated side conditions are automatically discharged via a solver for bitvectors provided by Islaris. The only manual steps are hints related to array indices that are accessed, and pure reasoning about lists to prove that one more byte is copied from s to d after each iteration of the loop.

14.6 Installing and Using an Exception Vector

The above `memcpy` is expressed in C, and the binary we verify uses only user-mode instructions, but because Islaris handles the full ISA, we can verify code that involves sequential aspects of the systems architecture, and system-mode instructions, in the same way, and just as easily and authoritatively. To illustrate this, we hand-wrote an Arm assembly program (Figure 14.8) that sets up an exception vector table to handle hypervisor calls at exception level 2 (EL2), sets up the system state to transition to exception level 1 (EL1), and then performs a hypervisor call (`hvc`) at EL1, which is handled at EL2 before returning to EL1 with `eret`. The exception handler for the hypervisor call is itself very simple: it sets the value of register `x0` to 42. This assembles, links, and runs correctly on a Raspberry Pi 3B+, and on QEMU.

The specification we prove for this code states that, upon reaching line 16, register `x0` contains the expected value 42. The interesting part of this verification is how Islaris handles the (changing) system configuration. The system configuration in the Sail models is largely held in registers.

For Armv8-A, these include around 500 system registers from the Arm ASL, alongside the normal general-purpose registers; the ASL/Sail definition refers to these in many places. An example is the `hcr_el2` register which controls many aspects of the Armv8-A virtualization. Here, we care about bit 31 of `hcr_el2` (set on line 6) which switches the EL1 exception level from 32-bit mode (AArch32) to 64-bit mode (AArch64). Reasoning about `hcr_el2` is like reasoning about any other register: the Isla trace of `msr hcr_el2, x0` contains a `(write-reg |HCR_EL2| v0)` event which is verified using HOARE-WRITE-REG, turning $\mathsf{HCR\_EL2} \mapsto_R \_$ into $\mathsf{HCR\_EL2} \mapsto_R 0x80000000$ (which is the word all of whose bits are 0, except the 31st bit, which is 1). The values of `hcr_el2` and other system registers are passed to Isla to simplify the traces of the instructions running at EL1 (lines 14-16) and the $\mathsf{HCR\_EL2} \mapsto_R 0x80000000$ assertion is used to discharge the corresponding `assume-reg` inserted by Isla.

14.7 RISC-V

We focused so far on Armv8-A, but it is important to note that almost everything presented here, including the tooling, is independent of the underlying architecture. To use Islaris as a verification tool for RISC-V code instead of Armv8-A code, one just needs to give the RISC-V Sail model instead of the Armv8-A Sail model to Isla, with a suitable assumption on the initial machine configuration. To demonstrate this, we compiled the `memcpy` C function from Figure 14.6 for RISC-V using the mainstream Clang compiler, and verified the resulting code (third column in Figure 14.6) using Islaris.

Although these two architectures differ greatly (*e.g.*, in their definitions of memory accesses), we can use the same assertions and rules described earlier, as the Isla traces are expressed in the same language. The specification of `memcpy` is thus very similar between the two architectures, differing only in the calling convention, system registers, valid ranges of memory addresses, and the required alignment of the return address (the last two omitted for presentation). Crucially, the specifications use the same assertion language, and the Islaris proof automation works equally well for both architectures.

14.8 Verification Workflow

Having seen how various kinds of programs can be verified using Islaris, we recap the verification workflow when using Islaris.

The first step of Islaris-based verification is to run Isla with the right constraints to generate the instruction traces. For most instructions the default constraints suffice to generate sensible traces but more complex instructions (e.g. `eret`) require specialized constraints (e.g. on specific bits of `hcr_el2`). These constraints are usually determined by knowledge of the architecture, knowledge of the intended context and behavior of the code, and interactive exploration using Isla. These constraints are enforced by the previously explained `assume` and `assume-reg` events.

The next step is to write a specification for the code and use the proof automation to discharge the separation logic reasoning. These steps are often intertwined as one often interactively modifies the specification (e.g. adding register points-to assertions) and re-runs the proof automation until it successfully discharges the separation logic reasoning. For large examples one can use intermediate specifications for chunks of code to make this process faster.

After the separation logic reasoning is discharged, the last step is to solve the pure side conditions generated by the verification. These are usually discharged by a combination of automatic solvers and manual reasoning, depending on the exact nature of the side conditions.

Chapter 15

Isla Trace Language

The Isla trace language (ITL) was originally developed solely for SMT-based symbolic execution.[1] This chapter describes our operational semantics for ITL that enables reasoning about Isla traces in Coq.

[1] Armstrong et al., "Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models", 2021 [Arm+21].

STEP-READ-REG-EQ
$$\frac{\Sigma[r] = v}{\langle \mathsf{ReadReg}(r, v) :: t, \Sigma\rangle \to \langle t, \Sigma\rangle}$$

STEP-READ-REG-NEQ
$$\frac{\Sigma[r] \neq v}{\langle \mathsf{ReadReg}(r, v) :: t, \Sigma\rangle \to \top}$$

STEP-WRITE-REG
$$\langle \mathsf{WriteReg}(r, v) :: t, \Sigma\rangle \to \langle t, \Sigma[r \mapsto v]\rangle$$

STEP-READ-MEM-EQ
$$\frac{|b| = n \qquad \Sigma[a..a{+}n] = \mathsf{enc}(b)}{\langle \mathsf{ReadMem}(b, a, n) :: t, \Sigma\rangle \to \langle t, \Sigma\rangle}$$

STEP-READ-MEM-EVENT
$$\frac{|b| = n \qquad \Sigma[a..a{+}n] = \bot \qquad \kappa = \mathsf{R}(a, b)}{\langle \mathsf{ReadMem}(b, a, n) :: t, \Sigma\rangle \xrightarrow{\kappa} \langle t, \Sigma\rangle}$$

STEP-READ-MEM-NEQ
$$\frac{|b| = n \qquad \Sigma[a..a{+}n] \neq \bot \qquad \Sigma[a..a{+}n] \neq \mathsf{enc}(b)}{\langle \mathsf{ReadMem}(b, a, n) :: t, \Sigma\rangle \to \top}$$

STEP-WRITE-MEM
$$\frac{|b| = n \qquad \Sigma[a..a{+}n] \neq \bot}{\langle \mathsf{WriteMem}(a, b, n) :: t, \Sigma\rangle \to \langle t, \Sigma[a..a{+}n \mapsto \mathsf{enc}(b)]\rangle}$$

STEP-WRITE-MEM-EVENT
$$\frac{|b| = n \qquad \Sigma[a..a{+}n] = \bot \qquad \kappa = \mathsf{W}(a, b)}{\langle \mathsf{WriteMem}(a, b, n) :: t, \Sigma\rangle \xrightarrow{\kappa} \langle t, \Sigma\rangle}$$

STEP-DECLARE-CONST
$$\frac{v \in \tau}{\langle \mathsf{DeclareConst}(x, \tau) :: t, \Sigma\rangle \to \langle t[v/x], \Sigma\rangle}$$

STEP-DEFINE-CONST
$$\frac{e \downarrow v}{\langle \mathsf{DefineConst}(x, e) :: t, \Sigma\rangle \to \langle t[v/x], \Sigma\rangle}$$

STEP-ASSERT-TRUE
$$\frac{e \downarrow \mathsf{true}}{\langle \mathsf{Assert}(e) :: t, \Sigma\rangle \to \langle t, \Sigma\rangle}$$

STEP-ASSERT-FALSE
$$\frac{e \downarrow \mathsf{false}}{\langle \mathsf{Assert}(e) :: t, \Sigma\rangle \to \top}$$

STEP-ASSUME-TRUE
$$\frac{e \downarrow \mathsf{true}}{\langle \mathsf{Assume}(e) :: t, \Sigma\rangle \to \langle t, \Sigma\rangle}$$

STEP-ASSUME-REG-TRUE
$$\frac{R[r] = v}{\langle \mathsf{AssumeReg}(r, v) :: t, \Sigma\rangle \to \langle t, \Sigma\rangle}$$

STEP-CASES
$$\frac{1 \leq i \leq n}{\langle \mathsf{Cases}(t_1, \ldots, t_n), \Sigma\rangle \to \langle t_i, \Sigma\rangle}$$

STEP-NIL
$$\frac{\Sigma[\mathsf{PC}] = a \qquad \Sigma[a] = t}{\langle [\,], \Sigma\rangle \to \langle t, \Sigma\rangle}$$

STEP-NIL-END
$$\frac{\Sigma[\mathsf{PC}] = a \qquad \Sigma[a] = \bot \qquad \kappa = \mathsf{E}(a)}{\langle [\,], \Sigma\rangle \xrightarrow{\kappa} \top}$$

STEP-FAIL
$$\frac{\text{No other rule reduces } \langle t, \Sigma\rangle}{\langle t, \Sigma\rangle \to \bot}$$

Figure 15.1: Operational semantics of the Isla trace language.

Traces, whose syntax is given in Figure 14.3, are reduced from left to right using the rules of Figure 15.1. The operational semantics is a labeled transition system over machine configurations σ. A machine configuration can either be a pair $\langle t, \Sigma\rangle$ of a trace t and a machine state, or a final configuration $\bot$ or $\top$ (denoting failure and successful termination). The single-step relation ($\xrightarrow{\kappa}$) is annotated with an (optional) label κ representing externally visible events, which are then accumulated by the

multistep relation ($\xrightarrow{\kappa s}{}^*$) in κs.

$$\kappa \ ::= \ \mathsf{R}(a, v_d) \mid \mathsf{W}(a, v_d) \mid \mathsf{E}(a) \qquad \text{(Label)}$$

Most reduction rules inspect and/or modify the machine state Σ, which is a triple (R, I, M) of finite partial maps.

$$R : \mathsf{Reg} \rightharpoonup \mathsf{Val} \qquad I : \mathsf{Addr} \rightharpoonup \mathsf{Trace} \qquad M : \mathsf{Addr} \rightharpoonup \mathsf{Byte}$$

The register map R associates registers with their value (*e.g.*, a bitvector), the instruction map I associates addresses (*i.e.*, 64-bit bitvectors) to Isla traces (*i.e.*, the trace for the instruction stored at the address), and the memory map M associates addresses to bytes (*i.e.*, 8-bit bitvectors). Assuming $\Sigma = (R, I, M)$, we write $\Sigma[r]$ for $R[r]$ and $\Sigma[r \mapsto v]$ for $(R[r \mapsto v], I, M)$, and similarly for I and M.

Non-determinism. The operational semantics of ITL are non-standard, because ITL is based on SMT constraints, not designed as a programming language. One therefore first introduces new (symbolic) variables via `declare-const`, which are then restricted by later constructs like `read-reg` or `assert`, as seen *e.g.*, in Figure 14.2 (in a more standard programming language, the read would return a value). To model this, the operational semantics of ITL makes heavy use of non-determinism:[2] the operational semantics of $\mathsf{DeclareConst}(x, \tau) :: t$ (given by step-declare-const) non-deterministically picks a value v of type τ and substitute it for x in t. This non-determinism is then restricted by events later in the trace. For example, the operational semantics of $\mathsf{ReadReg}(r, v)$ compares v with the value stored in r, and only allows further execution if the two values coincide (step-read-reg-eq). Otherwise, execution terminates in the state $\top$ (step-read-reg-neq), and thus these executions do not have to be considered further during verification. Overall, this leads to the proof rule hoare-read-reg in Figure 14.4. Note that the use of $\top$ instead of $\bot$ is crucial here, as otherwise it would be trivial to reach the failure state $\bot$ by picking a wrong value in step-declare-const.

[2] All non-determinism in Islaris is demonic, for a more detailed comparison of angelic and demonic non-determinism, see Part IV.

Non-determinism is also used for branching, as explained in §14.4. Traces of instructions with branching (*e.g.*, conditional jumps) typically contain a $\mathsf{Cases}(t_1, t_2)$ that splits the trace into multiple subtraces. The operational semantics non-deterministically picks one of these subtraces (step-cases), but this non-determinism is then restricted by Assert events on each subtrace. An $\mathsf{Assert}(e)$ ensures that one only has to consider this subtrace if e evaluates to true (step-assert-true, using a standard big-step semantics of SMT expressions $e \downarrow v$). Otherwise, execution terminates with $\top$, and this subtrace can be ignored (step-assert-false). So, intuitively, an Assert can be seen as an assertion *proven* by Isla during symbolic execution and *assumed* by verification.

The dual of these assertions are assumptions *used* by Isla to simplify the trace. These are encoded using Assume and AssumeReg, which behave like Assert and ReadReg, except that they terminate in the failure state $\bot$, instead of $\top$ (step-fail). One therefore has to prove during verification that these assumptions used by Isla hold (since the verification ensures that $\bot$ is not reachable).

Memory events. The ITL memory events ReadMem and WriteMem are similar to the corresponding register events, except that they read and write (little-endian) bitvectors from and to memory ($\mathsf{enc}(b)$ denotes the little-endian encoding of bitvector b and $|b|$ the number of bytes in this encoding). Reads and writes for unmapped memory (step-read-mem-event and step-write-mem-event) are treated as externally visible events, modeling interaction with devices via memory-mapped IO. This will be important for the adequacy of the Islaris separation logic (§16.2).

Instruction fetch. At the end of the trace of an instruction (configuration of the form $\langle [], \Sigma \rangle$), rule (step-nil) retrieves the address of the next instruction from the PC register, and loads the trace of the next instruction from the instruction map. If there is no such trace, the operational semantics terminates with the visible event $\mathsf{E}(a)$ (step-nil-end). (The name of the PC register is the only part of the operational semantics that is specific to the underlying Sail model.)

Chapter 16

Islaris Separation Logic

This chapter presents the Islaris separation logic: the interesting assertions and rules not already in §14 (§16.1 and Figure 16.1), the adequacy theorem (§16.2), and proof automation (§16.3).

16.1 Assertions and Rules

Register collections. We have already seen the $r \mapsto_R v$ assertion, asserting that the register r contains the value v, with its corresponding rules, in Figure 14.4 (§14.3). The Islaris separation logic additionally provides the $\mathsf{reg\_col}(C)$ assertion that collects a set of $r \mapsto_R v$ into a single assertion via a big separating conjunction. This is useful to deal with large numbers of registers. For example, $\mathsf{reg\_col}(\mathsf{sys\_regs})$ asserts the values of commonly used systems registers like `PSTATE.SP`. One can remove and add elements from $\mathsf{reg\_col}(C)$ via EQ-REG-COL-REG, and with this rule it is straightforward to derive rules for the register operations (*e.g.*, HOARE-READ-REG-COL).

Memory. The main assertion about memory is $a \mapsto_M b$, which asserts that the memory at address a stores the (little-endian encoded) bitvector b. The rule HOARE-READ-MEM for reading memory behaves similarly to the corresponding rule for registers, except that one has to check that the number of bytes of b, $|b|$, corresponds to the size of the read. The rule for writing works accordingly, and is omitted for brevity. Islaris also provides the $a \mapsto^*_M B$ assertion to handle arrays of bitvectors B, since arrays are common in low-level code. The rules for this assertion (*e.g.*, HOARE-READ-MEM-ARRAY) can be easily derived from the rules for $a \mapsto_M b$.

16.2 Adequacy of the Islaris Separation Logic

Islaris's *adequacy* theorem describes the guarantee that a successful verification provides. There are two parts to this guarantee. First, Islaris proves that the program never reaches the $\bot$ state, and thus that all assumptions used by Isla hold. Second, Islaris proves a (user-defined) safety property about the externally visible behavior of the program (*i.e.*, reads and writes to unmapped memory and termination as described in §15). For this, Islaris provides the $\mathsf{spec}(s)$ assertion stating that the externally visible behavior of the program satisfies the specification s given as a set of label sequences. This assertion is used in the following rule for

$$\textsc{eq-reg-col-reg}\quad \frac{(r,v) \in C}{\mathsf{reg\_col}(C) \dashv\vdash \mathsf{reg\_col}(C \setminus (r,v)) * r \mapsto_R v} \qquad \textsc{hoare-read-reg-col}\quad \frac{(r,v') \in C \qquad \{\mathsf{reg\_col}(C) * v = v' * P\}\, t}{\{\mathsf{reg\_col}(C) * P\}\, \mathsf{ReadReg}(r,v) :: t}$$

$$\textsc{hoare-read-mem}\quad \frac{|b'| = |b| = n \qquad \{a \mapsto_M b * b = b' * P\}\, t}{\{a \mapsto_M b * P\}\, \mathsf{ReadMem}(b', a, n) :: t} \qquad \textsc{hoare-read-mem-array}\quad \frac{0 \le i < |B| \qquad |b'| = |B_i| = n \qquad \{a \mapsto^*_M B * B_i = b' * P\}\, t}{\{a \mapsto^*_M B * P\}\, \mathsf{ReadMem}(b', a + i * n, n) :: t}$$

Figure 16.1: Selected rules of the Islaris separation logic.

reading from unmapped memory (there is a similar rule for writing):

$$\textsc{hoare-read-mem-mmio}\quad \frac{|b| = n \qquad [\mathsf{R}(a,b)] \in s \qquad \left\{a \mapsto^{IO}_M n * \mathsf{spec}(\{\kappa s \mid \mathsf{R}(a,b) :: \kappa s \in s\}) * P\right\} t}{\left\{a \mapsto^{IO}_M n * \mathsf{spec}(s) * P\right\} \mathsf{ReadMem}(b, a, n) :: t}$$

When reading a value v from unmapped memory at address a (witnessed by the assertion $a \mapsto^{IO}_M n$), one has to prove that the event $\mathsf{R}(a,v)$ is allowed by the spec s and the rest of the execution can only produce events κs where $\mathsf{R}(a,b) :: \kappa s \in s$.

We can now state adequacy for Islaris:

Theorem 2 (Adequacy) *For all initial states $\Sigma = (R, I, M)$, the following rule is sound:*

$$\frac{\left\{\mathsf{reg\_col}(R) * \mathsf{spec}(s) * \mathop{\Large *}_{I[a]=t} \mathsf{instr}(a,t) * \mathop{\Large *}_{M[a]=b} a \mapsto_M b * \mathop{\Large *}_{M[a]=\bot} a \mapsto^{IO}_M 1\right\}\, [] \qquad \langle [], \Sigma\rangle \xrightarrow{\kappa s}{}^{*} \sigma}{\sigma \neq \bot \wedge \kappa s \in s}$$

This captures the above intuition: for all initial states Σ, if one can prove a Hoare double assuming all the ownership from the initial state and $\mathsf{spec}(s)$, executions from this initial state never reach $\bot$, and the produced events satisfy s.

16.3 Islaris Proof Automation

While the above rules allow the verification of machine-code programs, using them directly would be quite tedious, since Isla expands every instruction to several ITL events. Thus, a crucial part of Islaris is proof automation, which is challenging because it should simultaneously be comprehensive (covering as much reasoning as possible), and efficient. As seen in Part II, RefinedC[1] addresses this problem by using Lithium. However, there is a fundamental challenge when applying Lithium to Islaris: RefinedC avoids backtracking by using rich types and the structure of source C programs to guide Lithium's proof search. However, Islaris has no types and, additionally, most of the source program structure has been lost. Up front, it is unclear how to avoid expensive backtracking during proof search.

[1] Sammler et al., "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types", 2021 [Sam+21b].

Our key insight is that—with some effort—backtracking can also be avoided using the *separation logic context*, which is available in Islaris. We illustrate this using $\mathsf{ReadReg}(r,v)$. Let us start by considering the following

naive representations of hoare-read-reg and hoare-read-reg-col in Lithium:

li-read-reg-naive

1: $\mathsf{wp}\,\mathsf{ReadReg}(r, v) :: t$ **:-**
2: $\quad \exists v'.\ \mathbf{exhale}\ r \mapsto_R v'\mathbf{;}\ \mathbf{inhale}\ v = v' * r \mapsto_R v'\mathbf{;}\ \mathsf{wp}\, t$

li-read-reg-col-naive

1: $\mathsf{wp}\,\mathsf{ReadReg}(r, v) :: t$ **:-**
2: $\quad \exists C\, v'.\ \mathbf{exhale}\ \mathsf{reg\_col}(C) * (r, v') \in C\mathbf{;}$
3: $\quad \mathbf{inhale}\ v = v' * \mathsf{reg\_col}(C)\mathbf{;}\ \mathsf{wp}\, t)$

These rules are stated using Iris's weakest precondition, $\mathsf{wp}\, t$ (Hoare doubles are defined as $\{P\}\, t \triangleq P \mathrel{-\!\!*} \mathsf{wp}\, t$). They apply when the automation needs to verify a trace starting with $\mathsf{ReadReg}(r, v)$. Rule li-read-reg-naive instructs Lithium to find an assertion $r \mapsto_R v'$ in the context, add the assumptions $v = v'$ and $r \mapsto_R v'$ to the context, and finally continue with proving $\mathsf{wp}\, t$. Rule li-read-reg-col-naive is similar except that it requires finding $\mathsf{reg\_col}(C)$ and proving $(r, v') \in C$.

There are two problems when doing proof search with these two rules. (1) It is not clear which rule to apply when verifying a trace starting with ReadReg. One solution is to try each rule in turn, but this requires backtracking and makes proof search inefficient. (2) A similar issue arises with rule li-read-reg-col-naive alone in cases where the context contains multiple register collections: finding the appropriate one also requires backtracking.

Our solution to these problems is to define a new find function $r \mapsto_R -$[2] that searches for r in the separation logic context, which we then use to replace the two rules above with a single rule that does not require backtracking:

li-read-reg

1: $\mathsf{wp}\,\mathsf{ReadReg}(r, v) :: t$ **:-**
2: $\quad x \leftarrow \mathbf{find}\ r \mapsto_R -\mathbf{;}$
3: $\quad \mathbf{match}\ x\ \mathbf{with}$
4: $\quad \mid v' \Rightarrow\ \mathbf{inhale}\ v = v' * r \mapsto_R v'\mathbf{;}\ \mathsf{wp}\, t$
5: $\quad \mid C \Rightarrow\ \exists v'.\ \mathbf{exhale}\ (r, v') \in C\mathbf{;}\ \ \mathbf{inhale}\ v = v' * \mathsf{reg\_col}(C)\mathbf{;}\ \mathsf{wp}\, t)$

If $r \mapsto_R -$ finds $r \mapsto_R v'$, then the rule above goes into the first branch (corresponding to li-read-reg-naive).[3] If $r \mapsto_R -$ finds $\mathsf{reg\_col}(C)$ with $(r, \_) \in C$,[4] the rule goes into the second branch (corresponding to li-read-reg-col-naive).

In effect, we have solved the problems above by shifting the role of backtracking over nondeterministic rules to a deterministic instruction $r \mapsto_R -$ which looks through the separation logic context efficiently.

A similar find function is used to decide between the rules for memory points-to predicates (hoare-read-mem, hoare-read-mem-array, and hoare-read-mem-mmio). It searches the context for an $a' \mapsto_M b$, $a' \mapsto_M^* B$, or $a' \mapsto_M^{IO} n$ assertion that contains the address a. Checking this containment requires querying a bitvector solver, as a is usually a complex bitvector expression computed by the Sail model.

[2] See §3.5 for an introduction to find functions in Lithium.

[3] The **match** statement is a meta-level match statement on the return value of $r \mapsto_R -$ that Lithium automatically reduces.

[4] To achieve this, we use a feature of Lithium not described in Part I: When providing a rule for a find function, Lithium allows specifying a custom tactic that determines if the pattern matches an assertion in the context. We use this to check that only $\mathsf{reg\_col}(C)$ assertions with $(r, \_) \in C$ are returned by the find function.

Chapter 17

Translation Validation for RISC-V

To explore whether one can remove Isla and the SMT solver from the Islaris TCB, we built infrastructure to prove (in Coq) correctness of the Isla-generated traces with respect to the Coq definitions generated by Sail from the Sail RISC-V model. This is a form of translation validation, rather than an up-front correctness proof of Isla: given an Isla-generated trace, the infrastructure can be used to prove that the trace is refined by the Sail-generated Coq model. This proof can then be composed with Theorem 2 to obtain a theorem that only mentions the Sail-generated Coq model and the user-written specification, without Isla or the Islaris separation logic. We have also investigated this approach for Armv8-A but found it infeasible, since the size of the Armv8-A model means it cannot be manipulated efficiently in Coq.

We first define an operational semantics for the free monad used by the Sail-generated Coq model, with constructors corresponding to the ITL events in Figure 14.3. The state of this semantics is similar to that of ITL except that the current instruction is an element m of the monad, instead of an ITL trace, and the instructions I_{Coq} are represented as bitvector opcodes not Isla traces. We then define a notion of refinement $\sigma_{\text{Coq}} \sqsubseteq \sigma_{\text{ITL}}$.[1] Crucially, when proving such refinements one can use the assumptions given by Assume and AssumeReg. Finally, we prove this refinement by establishing a simulation $m \sim t$ between the instructions:[2]

[1] This notion of refinement is different from DimSum's refinement described in Part IV.

[2] Done initiates the fetch of the next instruction, similar to []

Theorem 3 (Isla validation)

$$\frac{\forall a.\, I_{\text{Coq}}[a] \sim I_{\text{ITL}}[a]}{\langle \mathit{Done}, (R, I_{\text{Coq}}, M)\rangle \sqsubseteq \langle [\,], (R, I_{\text{ITL}}, M)\rangle}$$

To evaluate this infrastructure, we have proven $m \sim t$ for all instructions that appear in the RISC-V `memcpy` binary. The proofs are mostly automated using custom tactics, but require a few manual steps to match the branches of the Coq model to the subtraces of the Isla trace, and to check some facts that were automatically proven by the SMT solver. We also used this infrastructure to obtain a closed statement about a simple program that only mentions the Coq model and the user-defined specification by composing Theorem 2 and Theorem 3. Overall, this shows that the operational semantics described in §15 is sensible (especially its use of non-determinism and Assert vs. Assume), and increases confidence in our use of Isla and the underlying SMT solver, showing that this example does

not trigger any bugs in those. We did find a bit-flip bug in a primitive used by the Sail-generated Coq (not previously thoroughly exercised).

Chapter 18

Evaluation

We demonstrate that Islaris supports practical verification of a range of system software idioms. Our examples are not large in instruction count, but direct proofs above the Arm and RISC-V ISA models would require reasoning about many thousands of lines of those specifications, and they involve many system registers. They include part of a real-world exception handler that installs a new exception vector, and that is parametric on a relocation address offset; faulting from misaligned accesses; memory-mapped IO; and production C compiler output with inline assembly and function pointers. The `hvc` and `memcpy` examples are described in §14.

Relocation-parametric real-world code: pKVM exception handler. This is part of an exception handler taken from real-world code, namely the pKVM hypervisor under development by Google. The handler branches to one of two sub-handlers, depending on the cause of the exception and the value of a hypercall parameter. We assume one of these to be correct, as it calls into the large pKVM C codebase, but verify the hypercalls handled by the other, two of which replace the exception vector table—in total interacting with 49 different system registers. The verification establishes that each hypercall returns to the correct address at the correct exception level with appropriately updated system state.

This example exercises Islaris's ability to handle parametric traces. The hypervisor code is loaded into memory at an address offset determined at runtime, so a branch from the handler into the rest of the hypervisor needs to be adapted to that offset. This is done during initialization by patching four Armv8-A instructions, that each load a 16-bit immediate, to use the appropriate parts of the correct value. We thus have to verify a family of programs, one for each possible offset value. To achieve this, we use Isla's support for partially symbolic opcodes to generate traces for these instructions that are parametric in their immediate values. We can then verify for all offsets that the patched code will branch to the correct address.

The example also requires reasoning about an instruction under somewhat relaxed constraints. The two hypercalls that update the exception vector table (`HVC_SOFT_RESTART` and `HVC_RESET_VECTORS`) both conclude with the same block of code, ending in an `eret` instruction to return from the exception. The `eret` instruction uses the `SPSR` register to determine the values of various registers to be restored at the termination of an exception handler. However, the `HVC_SOFT_RESTART` hypercall updates

SPSR so that eret returns to exception level 2 (rather than the exception level of the caller)—this is necessary during the initialization of the hypervisor. Unfortunately this means neither the original nor the updated value of SPSR can be used to simplify the traces for eret. Instead, we give a more complex constraint, capturing both possible values. This results in a set of traces simple enough that we can prove in Coq which traces are relevant for each fixed value of SPSR. This allows us to recover fully simplified reasoning.

Unaligned access faults. To show how one can reason accurately about faults, we verified a misaligned store w.r.t. an Armv8-A configuration in which this raises an exception. We prove it jumps to the correct exception handler, saves the PC and PSTATE registers, masks interrupts, and updates the exception syndrome and fault address registers.

Interaction with MMIO: UART. To evaluate Islaris's capabilities to verify interaction with memory-mapped IO, we have verified the binary for the following C function, writing a character to a memory-mapped UART.

```
1 void uart1_putc(char c) {
2   while(!(*LSR & LSR_TX_EMPTY)) { asm volatile("nop"); }
3   *IO = (u32) c; }
```

The code polls whether the UART is ready to receive an input and then writes `c` to a special `IO` memory location; it runs on a Raspberry Pi 3B+ and in QEMU. We verified the specification:

$$\mathsf{srec}(R.\exists b.\, \mathsf{scons}(\mathsf{R}(\mathsf{LSR}, b), b[5]\,?\,\mathsf{scons}(\mathsf{W}(\mathsf{IO}, c), s) : R)))$$

This specification uses a loop (encoded via the least fixpoint combinator srec) to read b from the memory-mapped location LSR (via $\mathsf{scons}(\kappa, s)$ which prepends κ to the elements in s). If the fifth bit of b (corresponding to `LSR_TX_EMPTY`) is set, the UART is ready to receive input and the character c is written to the memory-mapped IO register, and the specification continues with s. Otherwise, it tries again.

C inline assembly: rbit. C code using inline assembly is often challenging for C verification tools, but not for Islaris, which applies to the compiled machine code. We show this by verifying a (compiled) C function that reverses the bits of its argument via an inline rbit instruction. The combination of C and assembly is handled automatically, with manual proof needed only to relate the complex bitvector term produced by Isla to the function's intuitive specification.

Higher-order reasoning: Binary search. C supports a limited form of higher-order functions, via function pointers. To show how Islaris handles this, we verified a binary search implementation that is parametric over the comparison function (based on the binary search case study discussed in §10). The implementation is written in C and compiled with clang `-O2`. In the verification, the function pointer is encoded via the a @@

P assertion and a formalization of the Arm AArch64 ABI C calling convention. Since this encoding only uses standard Islaris constructs, Islaris handles reasoning about the function pointer automatically.

RISC-V: Binary search and memcpy. To demonstrate that Islaris is not specific to a single ISA, we compiled and verified the binary for RISC-V, in addition to Armv8-A, for our two platform-independent case studies: the `memcpy` of §14, and the binary search. As already described in §14.7, the Islaris separation logic and most of the tooling is shared between the two ISAs. Only the (system) registers, calling convention, and some side conditions had to be adapted.

Test	ISA	Size (lines)				Time (s)	
		asm	ITL	Spec	Proof	Isla	Coq
memcpy	Arm	8	169	20	55	6	9/2/8/-
	RV	8	134	19	54	1	10/4/7/-
hvc	Arm	13	436	93	5	10	28/5/25/-
pKVM	Arm	47	1070	159	232	37	67/16/62/16
unaligned	Arm	1	104	89	29	2	10/12/24/-
UART	Arm	14	207	33	42	10	9/3/6/-
rbit	Arm	2	26	18	27	3	4/73/54/-
bin.search	Arm	32	741	25	146	25	54/16/37/19
	RV	48	801	25	108	5	63/22/37/21

Figure 18.1: Example sizes and times.

Proof sizes and times. The main goal of Islaris is to make it possible and practical to verify machine code above these authoritative models, which was not previously possible. Practicality requires a reasonable level of performance. Figure 18.1 gives the proof sizes and the Isla and Coq proof times for our examples.[1] Proof size is the number of manually-written lines, including any loop invariants. The Coq time is subdivided by '/'s into the Lithium-based proof automation (second step in §14.8), custom tactics to solve side conditions (third step in §14.8), and the Qed check of the generated proof term (this check happens after the programmer finishes the interactive proof). The larger case studies use intermediate specifications for some instructions to let these be verified in parallel; these are the last times given. Times were collected with a populated lia cache on an 8-core Intel Core i7 8th Gen laptop with 24 GB of RAM. Isla and the instruction specification proofs are parallelized. Overall, this shows that Islaris is already a practical tool for verifying challenging case studies against the full Armv8-A and RISC-V models, but further performance improvements are possible (especially when using many registers and in the bitvector automation).

[1] The numbers in Figure 18.1 (and in the rest of this part) reflect the original evaluation in Sammler et al., "Islaris: Verification of Machine Code Against Authoritative ISA Semantics", 2022 [Sam+22].

Chapter 19

Related Work

There have been many approaches to verification of assembly and machine code, using a wide variety of underlying models. Here, we compare to the most relevant related work.

L3 and decompilation into logic. Some closely related work uses L3,[1] which is a well-developed ISA specification language broadly similar to Sail, but with a simpler type system. L3 comes with handwritten models of ISA fragments of several architectures (ARMv4–7, ARMv8, MIPS, x86, and RISC-V) that can be extracted to HOL4 and Isabelle/HOL. Recent work[2] validated the L3 ARMv8 model against the HOL version of the ARMv8 Sail model. The main reasoning support in HOL4 is provided by per-architecture handwritten *step* libraries, which provide an equational view of individual instructions; CakeML[3] builds directly on these libraries and Myreen and Gordon[4] build a separation logic using them. This logic is significantly simpler than the Iris-based Islaris separation logic; in particular, it does not support higher-order specifications for code pointers. It is then integrated into the decompilation into logic approach,[5] which produces HOL functions that are equivalent to the machine code. This process has the advantage that it does not rely on an external SMT-solver, but the L3 models of Armv8 and RISC-V have substantially less coverage than the Sail models used here, and it is unclear whether the approach would scale to these larger models. Campbell and Stark[6] automate construction of step libraries using symbolic execution, similar to our use of Isla, but for test generation rather than verification.

ACL2 X86isa model. The ACL2 X86isa model[7] gives a detailed and well-validated description of a large fragment of the x86 architecture, including both user- and system-level instructions. The model comes with a large proof library for verifying programs via direct reasoning about the model and its state. However, unlike Islaris, X86isa does not provide a high-level separation logic. As a consequence, the proofs become quite large—*e.g.*, Goel et al.[8] report thousands of lines for a simple example. In contrast, Islaris proofs for similar-scale examples are usually one to two orders of magnitude smaller thanks to its proof automation. One reason for this difference is that X86isa requires explicit disjointness reasoning about memory regions that are automatically handled via separation logic in Islaris.

[1] Fox, "Directions in ISA Specification", 2012 [Fox12]; Fox, "Improved Tool Support for Machine-Code Decompilation in HOL4", 2015 [Fox15].

[2] Kanabar et al., "Taming an Authoritative Armv8 ISA Specification: L3 Validation and CakeML Compiler Verification", 2022 [KFM22].

[3] Fox et al., "Verified Compilation of CakeML to Multiple Machine-Code Targets", 2017 [Fox+17].

[4] Myreen and Gordon, "Hoare Logic for Realistically Modelled Machine Code", 2007 [MG07].

[5] Myreen et al., "Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic", 2008 [MGS08]; Myreen et al., "Decompilation into Logic - Improved", 2012 [MGS12]; Myreen, "Formal verification of machine-code programs", 2009 [Myr09].

[6] Campbell and Stark, "Extracting Behaviour from an Executable Instruction Set Model", 2016 [CS16].

[7] Goel and Jr., "Automated Code Proofs on a Formal Model of the X86", 2013 [GJ13]; Goel et al., "Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls", 2014 [Goe+14]; Goel et al., "Engineering a Formal, Executable x86 ISA Simulator for Software Verification", 2017 [GJK17].

[8] Goel et al., "Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls", 2014 [Goe+14].

Higher-order separation logic for assembly. Jensen et al.[9] provide a separation logic for a fragment of x86 assembly[10] in Coq. Its key feature is a higher-order frame connective that gives nice reasoning principles for jumps to unknown code. We achieve similar reasoning principles via the $\mathsf{wp}\, t$ connective described in §16.3 that is based on the standard Iris weakest precondition.

Bedrock[11] provides a separation logic for a custom intermediate language in Coq with a focus on proof automation. Bedrock inspired the a @@ P connective for handling code pointers. Bedrock's annotation overhead for verifying a memcpy function[12] is comparable to Islaris's for the similar memcpy function described in §14.5, with roughly comparable performance, even though Bedrock targets a much simpler intermediate language rather than full ISA semantics (∼45s for Bedrock vs. ∼30s for Islaris on the same machine, but with an older version of Coq for Bedrock).

Both Jensen et al. and Bedrock use models that are simple enough that they can be handled directly in Coq without SMT-based simplification, and both are specific to concrete languages, while Islaris works for multiple ISAs specified in Sail.

Large-scale systems verification efforts. There have been several successful efforts to verify large-scale systems w.r.t. assembly code, but based on low-level semantics that are considerably less authoritative and complete compared with the models used by Islaris. The PROSPER project[13] and seL4[14] manually extend the L3 models described above with the systems features they need. The verified concurrent kernel CertiKOS and hypervisor SeKVM[15] use CompCert's assembly semantics and add various models of some systems aspects. The assembly verification of the VerisoftXT project (that verified parts of the Hyper-V hypervisor[16]) uses Vx86[17] to translate x86 assembly code including some virtualization extensions to C code that can then be verified using VCC.[18] Syeda and Klein[19] build a program logic for address translation for Armv7-A.

Erbsen et al.[20] provide an integrated verification of an embedded system across hardware and software that includes direct verification of application code against the MIT RISC-V formalization (which is roughly comparable to the Sail-based RISC-V formalization[21]). Since RISC-V is comparatively small, it is not surprising that direct proofs against this model are possible, but it is unclear whether this approach would scale to significantly more complex models like Armv8-A. Also, all the work by Erbsen et al. is specific to RISC-V while Islaris applies generically to Armv8-A and RISC-V.

Push-button verification of assembly code. Serval[22] achieved impressive push-button verification w.r.t. small handwritten models of x86 and RISC-V, using SMT-based symbolic execution. However, Serval does not support modular Hoare-style reasoning as provided by Islaris, and only works for programs with bounded loops.

[9] Jensen et al., "High-Level Separation Logic for Low-Level Code", 2013 [JBK13].

[10] Kennedy et al., "Coq: The world's best macro assembler?", 2013 [Ken+13].

[11] Chlipala, "Mostly-Automated Verification of Low-Level programs in Computational Separation Logic", 2011 [Chl11]; Chlipala, "The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier", 2013 [Chl13]; Malecha et al., "Compositional Computational Reflection", 2014 [MCB14].

[12] Bedrock team, *Verification of memcpy*, 2015 [Bed15b].

[13] Baumann et al., "A High Assurance Virtualization Platform for ARMv8", 2016 [Bau+16]; Guanciale et al., "Provably secure memory isolation for Linux on ARM", 2016 [Gua+16].

[14] Klein et al., "seL4: Formal Verification of an OS Kernel", 2009 [Kle+09].

[15] Gu et al., "Building Certified Concurrent OS Kernels", 2019 [Gu+19]; Li et al., "A Secure and Formally Verified Linux KVM Hypervisor", 2021 [Li+21].

[16] Leinenbach and Santen, "Verifying the Microsoft Hyper-V Hypervisor with VCC", 2009 [LS09].

[17] Maus et al., "Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving", 2008 [MMS08].

[18] Cohen et al., "VCC: A Practical System for Verifying Concurrent C", 2009 [Coh+09].

[19] Syeda and Klein, "Formal Reasoning Under Cached Address Translation", 2020 [SK20].

[20] Erbsen et al., "Integration Verification across Software and Hardware for a Simple Embedded System", 2021 [Erb+21].

[21] RISC-V team, *ISA Formal Spec Public Review*, 2019 [RIS19].

[22] Nelson et al., "Scaling symbolic evaluation for automated verification of systems code with Serval", 2019 [Nel+19].

Part IV

DimSum

Chapter 20

Introduction

To focus and simplify the problem of program verification, it is common to assume that the programs one is verifying are written in a single, well-defined language. However, many (if not most) real-world programs are assembled from components written in *multiple* languages. For example, programs in languages as diverse as Go, OCaml, Python, Rust, and Swift depend on standard or legacy libraries written in C; operating systems commonly implement interrupt handling in assembly code; low-level drivers link with architecture-specific assembly code. It thus remains a grand challenge to build formal methods that can handle such realistic multi-language programs.

What makes this so difficult is that, to verify a multi-language program, it is often not sufficient to verify the program's components separately—we have to additionally reason about the interactions between them. In particular, at the boundaries, we have to account for the friction that arises from the *language differences*. For example, in a high-level language calling other code is often done through a function call construct with argument names, whereas assembly-like languages typically use jumps and designated argument registers. The languages could also differ in their representation of values (*e.g.*, hierarchical vs. flat), their language features (*e.g.*, structured vs. unstructured control flow), and their memory models (*e.g.*, an abstract memory model where pointers are offsets into abstract blocks vs. a concrete memory model where pointers are concrete integers).

Much prior work on multi-language verification has focused on the specific important case of compiler verification, and in particular so-called *compositional compiler verification*.[1] The broad goal of compositional compiler verification is to specify and verify compilers in terms of how they transform individual *libraries* in a program, so that different libraries may be correctly linked together even if they are produced by different verified compilers for potentially different languages. (This is in contrast to the original CompCert,[2] for example, which was verified only as a compiler for whole programs.) Building on the ideas of Matthews and Findler,[3] Ahmed and collaborators[4] have subsequently recognized compositional compiler verification as an instance of the much broader problem of *multi-language semantics*: what is the right way to even *define* the behavior and interoperation of multi-language programs to best support verified linking of code from different languages and compilers?

[1] Neis et al., "Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language", 2015 [Nei+15]; Perconti and Ahmed, "Verifying an Open Compiler Using Multi-language Semantics", 2014 [PA14]; Stewart et al., "Compositional CompCert", 2015 [Ste+15]; Song et al., "CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification", 2020 [Son+20]; Koenig and Shao, "CompCertO: Compiling Certified Open C Components", 2021 [KS21].

[2] Leroy, "Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant", 2006 [Ler06].

[3] Matthews and Findler, "Operational Semantics for Multi-Language Programs", 2007 [MF07].

[4] Ahmed and Blume, "An Equivalence-Preserving CPS Translation via Multi-Language Semantics", 2011 [AB11]; Perconti and Ahmed, "Verifying an Open Compiler Using Multi-language Semantics", 2014 [PA14]; Mates et al., "Under Control: Compositionally Correct Closure Conversion with Mutable State", 2019 [MPA19]; Patterson et al., "FunTAL: Reasonably Mixing a Functional Language with Assembly", 2017 [Pat+17]; Patterson et al., "Semantic Soundness for Language Interoperability", 2022 [Pat+22].

In this part of the dissertation, we propose a new approach to multi-language semantics and verification, which we realize in a new Coq-based framework we call **DimSum**. Our approach is based on a simple observation: if we consider the aforementioned work in the context of *multi-language semantics*, then certain aspects of the semantics are *fixed up front*, thus restricting the flexibility with which components from different languages can be composed together. In contrast, DimSum is what we call *decentralized*: the semantics of a library can be specified (and the library verified) without regard to the other libraries in the program—without even needing to know in what languages or under what memory models the other libraries are written.

20.1 Principles of Decentralization

To give a clearer sense of the motivation behind DimSum, let us now articulate four key principles that we aim to satisfy and explain the ways in which prior approaches do or do not satisfy them.

Principle #1: No fixed source language. Among the first to explore compositional compiler verification were Hur and collaborators;[5] they developed a line of work that culminated in Pilsner,[6] a compositionally verified compiler from an ML-like source language to a low-level assembly target language, which showcased the ability to soundly link the verified compilations of source-language modules with tricky, handwritten assembly modules. Despite the sophistication of the Pilsner verification, a key limitation of the approach used by this line of work was identified by Perconti and Ahmed:[7] Pilsner (and the other compilers in its lineage) only permit compiled libraries to be linked with assembly libraries for which there is *some semantically equivalent source module*. This limitation effectively rules out a significant use case for multi-language linking, since one of the main reasons to link compiled libraries against handwritten assembly libraries is when the latter provide some functionality that is *not* expressible in the source language of one's compiler.

Ahmed *et al.*'s line of work on multi-language semantics was at least partly motivated by the goal of lifting this restriction of Hur *et al.*'s work, a goal which Patterson and Ahmed[8] later termed "source-independent linking". With DimSum, we aim to fulfill this goal as well: we do not fix any one language as "the source"; rather, we explicitly allow for the possibility of linking high-level code with low-level (*e.g.*, assembly) libraries that have no high-level semantic equivalent.

Principle #2: No fixed set of languages. Ahmed *et al.*'s aforementioned research programme on multi-language semantics takes the approach of combining all interoperating languages into one big "syntactic multi-language" and providing type-directed *wrappers* to convert values of one language to values of the other languages. One advantage of this approach is that it supports interoperation between libraries in very different languages—libraries which, unlike in Pilsner, are not expressible in any common source language. Another advantage is that compositional

[5] Benton and Hur, "Biorthogonality, Step-indexing and Compiler Correctness", 2009 [BH09]; Benton and Hur, *Realizability and Compositional Compiler Correctness for a Polymorphic Language*, 2010 [BH10]; Hur and Dreyer, "A Kripke Logical Relation Between ML and Assembly", 2011 [HD11]; Hur et al., "The Marriage of Bisimulations and Kripke Logical Relations", 2012 [Hur+12].

[6] Neis et al., "Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language", 2015 [Nei+15].

[7] Perconti and Ahmed, "Verifying an Open Compiler Using Multi-language Semantics", 2014 [PA14].

[8] Patterson and Ahmed, "The Next 700 Compiler Correctness Theorems (Functional Pearl)", 2019 [PA19].

compiler correctness can then be formalized in terms of *contextual equivalence* (a very standard and well-understood criterion) in the syntactic multi-language.

A disadvantage of the syntactic multi-language approach, however, is that it requires one to fix the set of interoperating languages up front. As a result, it means that proofs about libraries in any one language must take into account all the other languages comprising the syntactic multi-language, and such proofs may break if new languages are added to the mix in the future.

With DimSum, we aim to support what we call *language-local* reasoning: we should not fix the set of interoperating languages up front, and we should be able to verify a library in one language without having to worry *a priori* about the other languages with which that library could be linked.

Principle #3: No fixed memory model. Since the development of CompCert, a wide range of projects have explored compositionally verified extensions of CompCert, including Compositional CompCert,[9] CompCertX,[10] SepCompCert,[11] CompCertM,[12] and CompCertO.[13] Except for SepCompCert, these projects follow Principles #1 and #2 above. However, unlike Pilsner and Ahmed *et al.*'s work, these CompCert extensions assume all interoperating languages to adhere to a particular memory model, namely the CompCert memory model.[14] As noted by Patterson and Ahmed,[15] this places a significant restriction on the set of languages that can realistically participate in a multi-language program.

With DimSum, we aim to support linking of libraries written in languages with *different* memory models, yet still allow such linking to be reasoned about in a language-local way (as per Principle #2). We will see a concrete instance of this problem in §21, where we link a language with an abstract memory model not unlike CompCert's (*i.e.*, pointers are abstract block identifiers with offsets) to an assembly language with a concrete memory model (*i.e.*, pointers are integer addresses).

Principle #4: No fixed notion of linking. A key aspect of multi-language semantics is formalizing interlanguage *linking*. Individual languages typically come equipped with their own pre-existing notions of *syntactic* linking $L \cup L'$, and then on top of that, multi-language semantics frameworks often define their own notions of *semantic* linking $L \oplus L'$ in order to characterize interoperation between different languages. However, in all the work we are aware of, the definition of semantic linking is fixed up front.

In DimSum, we aim to avoid fixing any "official" notion of semantic linking up front; instead, we permit users of the framework to develop new, library-specific notions of linking that support higher-level reasoning principles. To illustrate what this looks like, let us consider a concrete example, depicted in Figure 20.1: we take a high-level language with recursive functions called Rec and augment it with a coroutine library written in an assembly-like language called **Asm**.[16] More specifically, in the example, the two Rec-libraries stream and main are "linked" with each other through a coroutine **Asm**-library called **yield**. The stream function

[9] Stewart et al., "Compositional CompCert", 2015 [Ste+15].

[10] Gu et al., "Deep Specifications and Certified Abstraction Layers", 2015 [Gu+15]; Wang et al., "An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code", 2019 [WWS19].

[11] Kang et al., "Lightweight Verification of Separate Compilation", 2016 [Kan+16].

[12] Song et al., "CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification", 2020 [Son+20].

[13] Koenig and Shao, "CompCertO: Compiling Certified Open C Components", 2021 [KS21].

[14] Except for CompCertO, this assumption is crucial for the techniques. In the case of CompCertO, the assumption may not be crucial, but the approach has not been applied to a different memory model than the CompCert one. See §25 for details.

[15] Patterson and Ahmed, "The Next 700 Compiler Correctness Theorems (Functional Pearl)", 2019 [PA19].

[16] Inspired by Patrignani, "Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets", 2020 [Pat20], we depict Rec in red, sans-serif and **Asm** in **blue, bold**.

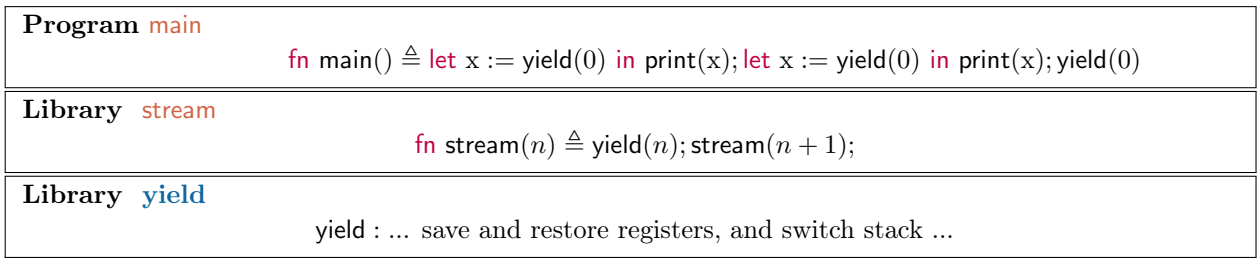


Figure 20.1: Example using coroutines.

generates an infinite stream of integers $0, 1, 2, \ldots$ that is consumed by the main function (*i.e.*, the main function prints the first two elements and then returns the third). For the **Asm**-library **yield**, the exact implementation is not relevant. The only relevant aspect of **yield** is that it sequentially passes the control back-and-forth between main and stream whenever yield is called in either.[17]

Most approaches to multi-language semantics can reason about this program in one way or another. For example, what they could do is consider the **Asm**-program $\downarrow \mathsf{main} \cup_{\mathsf{a}} \downarrow \mathsf{stream} \cup_{\mathsf{a}} \mathbf{yield}$ where $\downarrow \mathsf{R}$ denotes compilation and then show that it indeed prints 0, then 1, and then returns 2. What no existing approach can do—and here is where decentralization comes in—is locally extend the notion of semantic linking in one language (*e.g.*, Rec) due to the presence of a library written in another language (*e.g.*, the **yield**-library written in **Asm**). That is, at the level of Rec, all that we care about is that **yield** provides a *new form of semantic linking* "$\mathsf{R}_1 \oplus_{\mathtt{coro}} \mathsf{R}_2$", where function calls to yield on one side are perceived as function returns of yield on the other (*e.g.*, the call of $\mathsf{yield}(n)$ in stream is the return of $\mathsf{yield}(0)$ in main). This new, custom form of semantic linking "$\mathsf{R}_1 \oplus_{\mathtt{coro}} \mathsf{R}_2$" considerably simplifies reasoning about the interactions of R_1 and R_2, because we do not have to consider the **Asm**-implementation of **yield** itself. That is, whereas reasoning about **yield** drops down to the **Asm** level and involves reasoning about saving and restoring the stack pointer and certain other machine registers, reasoning about $\mathsf{R}_1 \oplus_{\mathtt{coro}} \mathsf{R}_2$ stays at the level of Rec.

[17] We only consider a statically known set of coroutines, so there is no function for spawning a coroutine. The first yield(0) in main starts the function stream with argument 0.

20.2 DimSum

In this part of the dissertation, we present DimSum, a Coq-based framework for multi-language semantics and verification that adheres to the four principles of decentralization laid out in §20.1. At the heart of DimSum lies our novel *decentralized multi-language semantics*, which forms the basis of all our reasoning. As a starting point for the semantics, we adopt the same viewpoint as the work surrounding CompCert,[18] namely that the semantics of a library L written in language L is *a labeled transition system*, which we call *a module*. What makes the DimSum approach decentralized is how we reason about these modules. We take a page out of the work on process algebra and think of the modules as *communicating processes*. More specifically, we associate each language L with a set of events E_{L}, we give semantics to a library L as a module $[\![L]\!]_{\mathsf{L}} \in \mathsf{Module}(E_{\mathsf{L}})$, and then we model the interactions of modules (*e.g.*, jumps) as synchronization

[18] Stewart et al., "Compositional CompCert", 2015 [Ste+15]; Song et al., "CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification", 2020 [Son+20]; Koenig and Shao, "CompCertO: Compiling Certified Open C Components", 2021 [KS21].

on events (*e.g.*, outgoing jumps are synchronized with incoming jumps). Following the style of process algebra, we build up larger modules from smaller ones using compositional combinators. For example, we define a suite of language-specific linking operators $M \oplus_{\mathsf{L}} M'$ that synchronize modules based on their events, and a collection of wrappers $\lceil M \rceil_{\mathsf{L} \rightleftharpoons \mathsf{L}'}$ that embed modules from one language L into another language L'.

The resulting approach to multi-language semantics is decentralized in the sense that when we reason about a particular collection of modules, we only care about their events and the languages to which these events belong. For example, in the rest of this part of the dissertation, we consider modules in the high-level language Rec, the low-level assembly language **Asm**, and a mathematical specification language Spec. When we reason about the interactions of two Rec-modules $\mathsf{M}_1 \oplus_{\mathtt{r}} \mathsf{M}_2$ (*e.g.*, to prove that they refine a specification written in Spec), then we only need to know about the calling convention of Rec and its memory model. In particular, we do not need to take into account the existence of the language **Asm** or its memory model in any shape or form. In contrast, when we reason about an **Asm**-module $\mathbf{M_1}$ interacting with a Rec-module M_2, *i.e.*, $\mathbf{M_1} \oplus_{\mathtt{a}} \lceil \mathsf{M}_2 \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$, we need to consider the different calling conventions and memory models of Rec and **Asm**. As a result, in DimSum, we can "mix and match" components written in different languages using a collection of language-specific combinators (*e.g.*, $\mathbf{M_1} \oplus_{\mathtt{a}} \mathbf{M_2}$, $\mathsf{M}_1 \oplus_{\mathtt{r}} \mathsf{M}_2$, $\mathsf{M}_1 \oplus_{\mathtt{coro}} \mathsf{M}_2$, and $\lceil \mathsf{M} \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$).

To make the simple idea of "multi-language program components as communicating processes" scale to reasonably complex languages such as Rec and **Asm**, we bring several ideas from the literature to bear, albeit casting them in a new light:

1. **Open-world events.** The work on fully abstract traces[19] introduced the idea of including all the visible parts of the program state in the events. Previously, the idea was used to prove contextual refinement via trace refinement where the traces consist of these detailed open-world events. In DimSum, we use similar open-world events to express the interactions of modules: modules share state (*e.g.*, the program memory), which has to be exchanged when two modules interact.

2. **Wrappers.** The work on multi-language semantics[20] introduced the idea of expressing language translations via wrappers. Previously, the idea was used to construct a syntactic multi-language using syntactic wrappers that embed expressions of other languages. In DimSum, we use wrappers at the level of the *semantics*: we define translations such as $\lceil \mathsf{M} \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$ that operate on modules (*i.e.*, LTSs) and translate the events between two languages (*e.g.*, Rec and **Asm**).

3. **Kripke relations.** The literature on compiler verification[21] employed the idea of *Kripke relations*—relations that maintain evolving internal state in order to track, *e.g.*, relationships between growing heaps—to reason about program executions. Previously, the idea was used to build expressive simulation relations for establishing compiler correctness. In DimSum, the role of expressive simulations is largely filled by our wrappers (see the previous idea) and, hence, we use Kripke relations to

[19] Jeffrey and Rathke, "Java Jr: Fully Abstract Trace Semantics for a Core Java Language", 2005 [JR05]; Laird, "A Fully Abstract Trace Semantics for General References", 2007 [Lai07].

[20] Matthews and Findler, "Operational Semantics for Multi-Language Programs", 2007 [MF07]; Ahmed and Blume, "An Equivalence-Preserving CPS Translation via Multi-Language Semantics", 2011 [AB11].

[21] Leroy and Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations", 2008 [LB08]; Hur and Dreyer, "A Kripke Logical Relation Between ML and Assembly", 2011 [HD11]; Perconti and Ahmed, "Verifying an Open Compiler Using Multi-language Semantics", 2014 [PA14]; Koenig and Shao, "CompCertO: Compiling Certified Open C Components", 2021 [KS21].

define them. Furthermore, to ease their formalization (in particular, to avoid explicit reasoning about possible worlds), we encode our Kripke relations in the separation logic Iris.[22]

4. **Rely-guarantee reasoning using angelic non-determinism.** The recent work on Conditional Contextual Refinement (CCR)[23] explored the idea of expressing rely-guarantee reasoning between a program component and its environment using angelic and demonic non-determinism. Previously, the idea was used to add user-provided preconditions and postconditions to contextual refinements. In DimSum, we apply the idea in our wrappers (*e.g.*, $\lceil \mathsf{M} \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$) to define language-specific protocols between a module and its environment.

[22] Jung et al., "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning", 2015 [Jun+15]; Jung et al., "Higher-Order Ghost State", 2016 [Jun+16]; Krebbers et al., "The Essence of Higher-Order Concurrent Separation Logic", 2017 [Kre+17]; Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b]; Jung, "Understanding and Evolving the Rust Programming Language", 2020 [Jun20].

[23] Song et al., "Conditional Contextual Refinement", 2023 [Son+23].

Contributions. In summary, the main contribution of this part of the dissertation is DimSum, a Coq-based framework for decentralized multi-language semantics and verification. The framework introduces the notion of a module, refinement between modules, and a library of language-agnostic combinators for linking and translating modules (§22). We then apply the framework to concrete instantiations:

- An instantiation of DimSum with (1) a high-level imperative language Rec with structured values, function calls, and an abstract memory model; (2) an assembly language **Asm** based on registers, unstructured jumps and a concrete memory model; and (3) a mathematical specification language Spec, together with linking operators (§23) and wrappers (§24).
- Two **Asm** libraries that extend Rec with new kinds of functionality: A library for pointer comparison (§21) and the coroutine library described earlier (§23.3).
- A compositional multi-pass compiler from Rec to **Asm** (§24).

Scope. We present a first step towards exploring a decentralized approach for multi-language verification. As such, we focus on the setting of a C-like language Rec and an assembly language **Asm**. This is similar to the compositional variants of CompCert, except that the two languages differ in their memory model and program components can interact with unstructured jumps at the **Asm** level (and, of course, that Rec and **Asm** are significantly simpler than the realistic languages used by CompCert). It would be interesting to consider languages with other features like closures, garbage collection, types, or concurrency in future work.

Additionally, we focus our attention on safety properties and do not prove liveness properties (similar to Sprenger et al.,[24] who use process algebra ideas for the verification of distributed systems). This restriction simplified the development of DimSum's model (in particular, the notion of refinement). We believe it should be possible to extend DimSum to support liveness reasoning, but we leave this to future work.

[24] Sprenger et al., "Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification", 2020 [Spr+20].

Chapter 21

Key Ideas

To illustrate the key ideas of DimSum, let us consider a motivating example. We want to verify the following program using libraries depicted in Figure 21.1:

$$\begin{aligned}\mathsf{fn}\ \mathsf{main}() \triangleq\ & \mathsf{local}\ x[3]; x[0] \leftarrow 1; x[1] \leftarrow 2; && \backslash\backslash\ x \mapsto [1,2,0]\\ & \mathsf{memmove}(x+1, x+0, 2); && \backslash\backslash\ x \mapsto [1,1,2]\\ & \mathsf{print}(x[1]); \mathsf{print}(x[2])\end{aligned}$$

The program first initializes the local array x, then moves the contents of x by one to the right using memmove, and finally prints the last two elements of x. It is primarily written in Rec, our high-level language with recursive functions. The parts that are not written in Rec, print and locle, are written in **Asm**, our low-level assembly language, because—as we will soon see—they cannot be implemented in Rec.

Library memmove

$\mathsf{fn}\ \mathsf{memmove}(d, s, n) \triangleq \mathsf{if}\ \mathsf{locle}(d, s)\ \mathsf{then}\ \mathsf{memcpy}(d, s, n, 1)\ \mathsf{else}\ \mathsf{memcpy}(d+n-1, s+n-1, n, -1)$

$\mathsf{fn}\ \mathsf{memcpy}(d, s, n, o) \triangleq \mathsf{if}\ 0 < n\ \mathsf{then}\ d \leftarrow !s; \mathsf{memcpy}(d+o, s+o, n-1, o)$

Library **locle**

locle : sle **x0**, **x0**, **x1**; ret

Library **print**

print : mov **x8**, **PRINT**; syscall; ret

Figure 21.1: Libraries written in Rec and **Asm**.

The function memmove is inspired by the corresponding function in the C standard library. It takes in a source pointer s, a destination pointer d, and the number of elements n that should be copied over. It then checks whether the source lies to the left or to the right of the destination in memory using the **Asm**-library **locle**. Depending on the outcome, memmove then copies the memory either front-to-back or back-to-front to the destination. The function memmove varies the copy direction to ensure that it does not step on its own toes: even if the two pointers overlap, memmove will never overwrite data that was supposed to be copied later.

The functions print and locle are implemented in **Asm**. The function locle simply compares its two arguments with less-or-equal and returns the result. The arguments of locle are in the first two registers **x0** and **x1**, because the calling convention of **Asm** is that the arguments are

in **x0**-**x8**. The return value is stored in **x0**. The function print leaves the argument **x0** unchanged, stores the flag for printing in **x8**, and then triggers a *syscall* (*i.e.*, a call to the operating system).

What makes this example interesting is that the functions print and locle have to be implemented in **Asm** because they cannot be implemented in the high-level language Rec. For print, the reason is that it makes a syscall, which—similar to C—is not something Rec can do. For locle, the reason is that it compares two pointers. Comparing pointers in **Asm** is easy because they are "just" integers. In contrast, Rec cannot compare pointers natively because it uses an abstract, block-based memory model (inspired by CompCert[1]). That is, conceptually, memory in Rec is a collection of unordered blocks, and a pointer consists of a *block identifier* and an *offset into the block*. Since the blocks are unordered, comparing pointers from different blocks (as memmove does when called with pointers into different blocks) does not make sense from the perspective of Rec.

[1] Leroy and Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations", 2008 [LB08].

Verification goal. Our end goal for this example will be to show that the entire program refines a top-level specification, which says that the program prints 1 and then 2. Let us make this goal more precise. The program consists of several Rec and **Asm**-libraries. To obtain a whole program, we thus have to *compile* the Rec-libraries to **Asm**-libraries and then *link* all the **Asm**-libraries together. We end up with the following program:

$$\textbf{onetwo} \triangleq \downarrow \mathsf{main} \cup_a \downarrow \mathsf{memmove} \cup_a \textbf{locle} \cup_a \textbf{print}$$

Here, main denotes a singleton library containing the main function from above, $\mathbf{A_1} \cup_a \mathbf{A_2}$ denotes syntactic linking in **Asm**, and $\downarrow R$ denotes compilation from of a Rec-library R to an **Asm**-library.[2]

[2] The compiler will be explained in §24, but its exact definition is not relevant for this example.

For the **Asm**-program **onetwo**, we then want to show that it refines a specification $\textbf{onetwo}_{\mathsf{spec}}$:

$$\textbf{onetwo} \preceq_{a,s} \textbf{onetwo}_{\mathsf{spec}}$$

In DimSum, refinement is defined as a notion of simulation, roughly stating that each step of **onetwo** can be matched by zero or more steps of $\textbf{onetwo}_{\mathsf{spec}}$ producing the same externally visible behavior (for details see §22.1). The specification $\textbf{onetwo}_{\mathsf{spec}}$ is written in our specification language Spec, which we will discuss later in this chapter. Roughly speaking, the specification says that the program prints 1 and then 2.

21.1 Event-Based Semantics

To explain how we define and prove $\textbf{onetwo} \preceq_{a,s} \textbf{onetwo}_{\mathsf{spec}}$, we have to turn to the core building block of DimSum: *modules*. Modules are how DimSum assigns meaning to every program component (*e.g.*, memmove, **locle**, and $\textbf{onetwo}_{\mathsf{spec}}$). The entire approach is centered around modules: we define interpretations of syntactic libraries into semantic modules, we define refinement as a simulation on modules, we define wrappers between modules of different languages, and we define semantic linking operators as combinators on modules. We will make the notion of a module precise

$$\begin{aligned}
\mathbf{Events} \ni \mathbf{e} &::= \mathbf{Jump!}(\mathbf{r},\mathbf{m}) \mid \mathbf{Jump?}(\mathbf{r},\mathbf{m}) \\
&\mid \mathbf{Syscall!}(\mathbf{v_1},\mathbf{v_2},\mathbf{m}) \mid \mathbf{SyscallRet?}(\mathbf{v},\mathbf{m}) \\
\mathbf{Memory} \ni \mathbf{m} &\triangleq \mathbb{Z} \xrightarrow{\text{fin}} \mathbf{Val} \cup \{\#\} \\
\mathbf{Registers} \ni \mathbf{r} &\triangleq \mathbf{RegisterName} \to \mathbf{Val} \\
\mathbf{Val} \ni \mathbf{v} &\triangleq \mathbb{Z} \\
\mathbf{RegisterName} \ni \mathbf{x} &\triangleq \{\mathbf{x0},\ldots,\mathbf{x30},\mathbf{sp},\mathbf{pc}\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Events} \ni \mathsf{e} &::= \mathsf{Call!}(\mathsf{f},\overline{v},\mathsf{m}) \mid \mathsf{Call?}(\mathsf{f},\overline{v},\mathsf{m}) \\
&\mid \mathsf{Return!}(\overline{v},\mathsf{m}) \mid \mathsf{Return?}(\overline{v},\mathsf{m}) \\
\mathsf{Memory} \ni \mathsf{m} &\triangleq \mathsf{Loc} \xrightarrow{\text{fin}} \mathsf{Val} \\
\mathsf{Loc} \ni \ell &::= \{\mathsf{blockid} : \mathsf{Id}, \mathsf{offset} : \mathbb{Z}\} \\
\mathsf{Val} \ni \mathsf{v} &::= z : \mathbb{Z} \mid b : \mathbb{B} \mid \ell : \mathsf{Loc} \\
\mathsf{FnName} \ni \mathsf{f} &\triangleq \mathsf{String}
\end{aligned}$$

Figure 21.2: **Asm** and Rec events.

in §22. For now, it suffices to know that a module $M \in \mathsf{Module}(E)$ is a labeled transition system emitting events from a language-specific set of events E.

Event-based communication. The set of events E of each module $M \in \mathsf{Module}(E)$ varies from language to language. For a given language, these events formalize how modules interact with their environment. That is, following the school of process algebra, we model the interaction of program components as event-based communication (*i.e.*, synchronization on events). For example, the events of Rec are function calls, and the events of **Asm** are jumps and syscalls. To scale this simple idea to stateful languages like Rec and **Asm**, we borrow an idea from the work on fully abstract traces:[3] the events carry *a detailed description of the program state*. As we will see, this enables expressing linking between modules as event synchronization.

[3] Jeffrey and Rathke, "Java Jr: Fully Abstract Trace Semantics for a Core Java Language", 2005 [JR05]; Laird, "A Fully Abstract Trace Semantics for General References", 2007 [Lai07].

The events of Rec and **Asm** are shown in Figure 21.2. Rec-modules (*i.e.*, modules with Rec-events) can emit function calls with $\mathsf{Call!}(\mathsf{f},\overline{v},\mathsf{m})$ and accept incoming function calls with $\mathsf{Call?}(\mathsf{f},\overline{v},\mathsf{m})$. In general, we distinguish between *outgoing events* (!) and *incoming events* (?). In both cases, the events include the function name f, the arguments $\overline{v}$, *and the entire memory* m. Rec-modules can return from calls with $\mathsf{Return!}(\mathsf{v},\mathsf{m})$ and accept returns from functions they called with $\mathsf{Return?}(\mathsf{v},\mathsf{m})$. In **Asm**, modules communicate using jumps: $\mathbf{Jump!}(\mathbf{r},\mathbf{m})$ and $\mathbf{Jump?}(\mathbf{r},\mathbf{m})$. These events contain the registers **r** (including the target address of the jump $\mathbf{r}(\mathbf{pc})$) and the program memory **m**. Additionally, **Asm**-modules can initiate syscalls with $\mathbf{Syscall!}(\mathbf{v_1},\mathbf{v_2},\mathbf{m})$ where $\mathbf{v_1}$ is the syscall identifier (*e.g.*, $\mathbf{PRINT} = 8$), $\mathbf{v_2}$ is the argument of the syscall, and **m** the memory when performing the syscall. They can then receive control again from the operating system through $\mathbf{SyscallRet?}(\mathbf{v},\mathbf{m})$ with return value **v** and resulting memory **m**. (We assume a syscall calling convention where all registers except the return register **x0** are restored.)

By comparing the events of the two languages, we can quite succinctly see their differences—the differences that we have to deal with in the proof of $\mathbf{onetwo} \preceq_{\mathsf{a}\mathsf{s}} \mathbf{onetwo}_{\mathsf{spec}}$. First, the two languages use a different function call structure. Calls in Rec are always bracketed with first a call event and then a return event, while **Asm**-modules only emit and accept jumps, without distinguishing calls from returns. The second important distinction is that **Asm** can do syscalls, whereas programs written in Rec cannot. This means that for any **Asm** program doing a

syscall, there is no corresponding Rec program. The third distinction is that Rec uses a structured model of values: they can be integers, locations, or Booleans. In **Asm**, values can only be integers, so pointers and Booleans are represented as integers. And finally, Rec and **Asm** use very different memory models. In Rec the memory model is block based, whereas in **Asm**, the memory is simply a map from addresses, *i.e.*, integers, to integers.[4]

With the events of Rec and **Asm** at hand, let us turn to the semantics of their syntactic libraries. For each language, we define a *module semantics* $[\![-]\!]_-$ that maps syntactic libraries (*i.e.*, R and **A**) into semantic modules (*i.e.*, $[\![\mathsf{R}]\!]_{\mathsf{r}} \in \mathsf{Module}(\mathsf{Events})$ and $[\![\mathbf{A}]\!]_{\mathsf{a}} \in \mathsf{Module}(\mathbf{Events})$) based on the operational semantics of the language. The exact definitions of the module semantics will not be relevant for the rest of this chapter, so we postpone them to §23.

[4] Technically, a memory address can also be part of a "guard page" denoted by #. An access to a guard page immediately and safely terminates the program. Each stack is followed by a guard page that is used to detect stack overflow, as described *e.g.*, by Tanenbaum and Bos, *Modern Operating Systems*, 2014 [TB14], Section 11.5.

High-level specifications. Before we can start the verification of the refinement $\mathbf{onetwo} \preceq_{\mathsf{a}\mathsf{s}} \mathbf{onetwo}_{\mathsf{spec}}$, we first have to *define the specification* $\mathbf{onetwo}_{\mathsf{spec}}$. For this, we use the specification language Spec. We will formally define Spec in §22.2. For now, it suffices to know that Spec is a language with *co-inductively defined* syntax and the following constructs:

$$\mathsf{Spec}(E) \ni p ::=_{\mathsf{coind}} \mathsf{any} \mid \mathsf{vis}(e); p \mid \mathsf{assume}(\phi); p \mid \exists x : T; p(x) \mid \cdots \qquad (e \in E, \phi \in \mathit{Prop})$$

The construct any means the implementation can do anything, *i.e.*, its behavior is not specified further. The construct $\mathsf{vis}(e); p$ means the implementation emits the visible event $e \in E$ and afterwards behaves like p. Spec is parametric over the set of events E that the programs emits. The construct $\mathsf{assume}(\phi); p$ means the implementation must behave like p *if* the proposition ϕ is true; otherwise, it may have any behavior. Finally, the construct $\exists x : T; p$ means the implementation must non-deterministically choose some $x : T$ and then behave like $p(x)$. As we will see in §21.2, the fact that programs are defined co-inductively in Spec allows us to express unbounded loops in the specifications.

With Spec at hand, we can turn to the specification of our example program **onetwo**. Since **onetwo** is an **Asm**-library, its specification is stated using **Asm**-events such as jumps and syscalls:

$$\begin{aligned}\mathbf{onetwo}_{\mathsf{spec}} \triangleq\ & \exists \mathbf{r}, \mathbf{m_0}; \mathsf{vis}(\mathbf{Jump?}(\mathbf{r}, \mathbf{m_0})); \mathsf{assume}(\mathbf{r}(\mathbf{pc}) = a_{\mathsf{main}} \wedge \mathbf{has\_stack}(\mathbf{r}(\mathbf{sp}), \mathbf{m_0})); \\ & \exists \mathbf{m_1}; \mathsf{vis}(\mathbf{Syscall!}(\mathbf{PRINT}, \mathbf{1}, \mathbf{m_1})); \exists \mathbf{m_2}; \mathsf{vis}(\mathbf{SyscallRet?}(*, \mathbf{m_2})); \mathsf{assume}(\mathbf{m_2} = \mathbf{m_1}); \\ & \mathsf{vis}(\mathbf{Syscall!}(\mathbf{PRINT}, \mathbf{2}, *)); \mathsf{vis}(\mathbf{SyscallRet?}(*, *)); \mathsf{any}\end{aligned}$$

First, the program can accept any jump to it from the environment with $\mathbf{Jump?}(\mathbf{r}, \mathbf{m_0})$. The following assume encodes that, during the verification of an implementation against this specification, one need only consider the choices of $\mathbf{r}$ and $\mathbf{m_0}$ where the environment decides to jump to the *start of the main function* (*i.e.*, the program counter **pc** points to the first instruction in main after compilation), and where the stack pointer **sp** points to a valid stack in memory $\mathbf{m_0}$ (because compiled Rec-libraries assume a stack). In this case, the implementation will perform a sequence of events: it will print 1 by emitting a syscall and wait for the operating

system to return;[5] then, assuming that the print syscall did not change the memory, it will print 2, and again wait for the operating system to return. After this point, the specification (for simplicity) uses any so as not to constrain the program's behavior further.

[5] The return value of the syscall is irrelevant, so we omit it using $*$. The $*$ notation is interpreted via non-deterministic choice, *i.e.*, $\mathsf{vis}(\mathbf{SyscallRet}?(*,*)); p$ is defined as $\exists \mathbf{v}; \exists \mathbf{m}; \mathsf{vis}(\mathbf{SyscallRet}?(\mathbf{v},\mathbf{m})); p$.

Module refinement. Finally, we can see how the refinement that we want to prove ($\mathbf{onetwo}\ {}_{\mathtt{a}}\preceq_{\mathtt{s}} \mathbf{onetwo}_{\mathtt{spec}}$) is defined:

$$\mathbf{onetwo}\ {}_{\mathtt{a}}\preceq_{\mathtt{s}} \mathbf{onetwo}_{\mathtt{spec}} \triangleq [\![\mathbf{onetwo}]\!]_{\mathtt{a}} \preceq [\![\mathbf{onetwo}_{\mathtt{spec}}]\!]_{\mathtt{s}}$$

In DimSum, we do not center our reasoning around refinements relating the *syntax* of two programs (*i.e.*, libraries), but around refinements relating the *semantics* of two programs (*i.e.*, modules). Here, $[\![\cdot]\!]_{\mathtt{a}}$ is the module semantics of **Asm** mentioned earlier, $[\![\cdot]\!]_{\mathtt{s}}$ is the module semantics of $\mathsf{Spec}(\mathbf{Events})$ (defined in §22.2), and $M_1 \preceq M_2$ is *the language-agnostic simulation relation* of DimSum (defined in §22.1), which we use as the notion of refinement. Let us now see how to prove this refinement.

21.2 The Proof Strategy

$$\begin{aligned}
[\![\mathbf{onetwo}]\!]_{\mathtt{a}} &= [\![\downarrow \mathsf{main} \cup_{\mathtt{a}} \downarrow \mathsf{memmove} \cup_{\mathtt{a}} \mathbf{locle} \cup_{\mathtt{a}} \mathbf{print}]\!]_{\mathtt{a}} && (1)\\
&\preceq [\![\downarrow \mathsf{main}]\!]_{\mathtt{a}} \oplus_{\mathtt{a}} [\![\downarrow \mathsf{memmove}]\!]_{\mathtt{a}} \oplus_{\mathtt{a}} [\![\mathbf{locle}]\!]_{\mathtt{a}} \oplus_{\mathtt{a}} [\![\mathbf{print}]\!]_{\mathtt{a}} && (2)\\
&\preceq \lceil [\![\mathsf{main}]\!]_{\mathtt{r}} \rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}} \oplus_{\mathtt{a}} \lceil [\![\mathsf{memmove}]\!]_{\mathtt{r}} \rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}} \oplus_{\mathtt{a}} [\![\mathbf{locle}]\!]_{\mathtt{a}} \oplus_{\mathtt{a}} [\![\mathbf{print}]\!]_{\mathtt{a}} && (3)\\
&\preceq \lceil [\![\mathsf{main}]\!]_{\mathtt{r}} \rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}} \oplus_{\mathtt{a}} \lceil [\![\mathsf{memmove}]\!]_{\mathtt{r}} \rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}} \oplus_{\mathtt{a}} \lceil [\![\mathsf{locle}_{\mathtt{spec}}]\!]_{\mathtt{s}} \rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}} \oplus_{\mathtt{a}} [\![\mathbf{print}_{\mathtt{spec}}]\!]_{\mathtt{s}} && (4)\\
&\preceq \lceil [\![\mathsf{main}]\!]_{\mathtt{r}} \oplus_{\mathtt{r}} [\![\mathsf{memmove}]\!]_{\mathtt{r}} \oplus_{\mathtt{r}} [\![\mathsf{locle}_{\mathtt{spec}}]\!]_{\mathtt{s}} \rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}} \oplus_{\mathtt{a}} [\![\mathbf{print}_{\mathtt{spec}}]\!]_{\mathtt{s}} && (5)\\
&\preceq \lceil [\![\mathsf{main} \cup_{\mathtt{r}} \mathsf{memmove}]\!]_{\mathtt{r}} \oplus_{\mathtt{r}} [\![\mathsf{locle}_{\mathtt{spec}}]\!]_{\mathtt{s}} \rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}} \oplus_{\mathtt{a}} [\![\mathbf{print}_{\mathtt{spec}}]\!]_{\mathtt{s}} && (6)\\
&\preceq \lceil [\![\mathsf{main}_{\mathtt{spec}}]\!]_{\mathtt{s}} \rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}} \oplus_{\mathtt{a}} [\![\mathbf{print}_{\mathtt{spec}}]\!]_{\mathtt{s}} && (7)\\
&\preceq [\![\mathbf{onetwo}_{\mathtt{spec}}]\!]_{\mathtt{s}} && (8)
\end{aligned}$$

Figure 21.3: Proof outline.

The proof of $[\![\mathbf{onetwo}]\!]_{\mathtt{a}} \preceq [\![\mathbf{onetwo}_{\mathtt{spec}}]\!]_{\mathtt{s}}$ consists of a sequence of refinements, as depicted in Figure 21.3. We can concatenate the sequence into our desired goal, because the refinement relation $M_1 \preceq M_2$ is transitive (and reflexive, see Lemma 4). The basic proof strategy—and here is where the decentralization of DimSum comes in—will be to decompose the program into several independent parts, gradually abstract those parts, and then assemble the entire program again. We will discuss these steps below. Along the way, we will point out whether the proof step is specific to the example or a generic reasoning principle for the involved languages (see Figure 21.4).

Linking [(1) to (2), generic]. As a first step, we decompose the program **onetwo** into a collection of **Asm**-modules. We do so by replacing the *syntactic linking operator* $\mathbf{A_1} \cup_{\mathtt{a}} \mathbf{A_2}$ of **Asm** with the *semantic linking operator* $\mathbf{M_1}\ {}^{\mathbf{d_1}}\oplus_{\mathtt{a}}^{\mathbf{d_2}}\ \mathbf{M_2}$ of **Asm** (using ASM-LINK-SYN). The syntactic operator $\mathbf{A_1} \cup_{\mathtt{a}} \mathbf{A_2}$ takes two **Asm**-*libraries* $\mathbf{A_1}$ and $\mathbf{A_2}$ and combines their program code. In contrast, the semantic linking operator $\mathbf{M_1}\ {}^{\mathbf{d_1}}\oplus_{\mathtt{a}}^{\mathbf{d_2}}\ \mathbf{M_2}$ takes two **Asm**-*modules* $\mathbf{M_1}$ and $\mathbf{M_2}$ with associated instruction addresses $\mathbf{d_1}$ and $\mathbf{d_2}$ and then synchronizes them via their jump events.[6]

[6] We omit $\mathbf{d_1}$ and $\mathbf{d_2}$ where they clutter the discussion. We write $|\mathbf{A}|$ for the instruction addresses of $\mathbf{A}$.

ASM-LINK-SYN

$$\frac{|\mathbf{A_1}| \cap |\mathbf{A_2}| = \emptyset}{[\![\mathbf{A_1} \cup_{\mathsf{a}} \mathbf{A_2}]\!]_{\mathsf{a}} \equiv [\![\mathbf{A_1}]\!]_{\mathsf{a}} \,{}^{|\mathbf{A_1}|}\!\oplus_{\mathsf{a}}^{|\mathbf{A_2}|}\, [\![\mathbf{A_2}]\!]_{\mathsf{a}}}$$

REC-LINK-SYN

$$\frac{|\mathsf{R}_1| \cap |\mathsf{R}_2| = \emptyset}{[\![\mathsf{R}_1 \cup_{\mathsf{r}} \mathsf{R}_2]\!]_{\mathsf{r}} \equiv [\![\mathsf{R}_1]\!]_{\mathsf{r}} \,{}^{|\mathsf{R}_1|}\!\oplus_{\mathsf{r}}^{|\mathsf{R}_2|}\, [\![\mathsf{R}_2]\!]_{\mathsf{r}}}$$

ASM-LINK-HORIZONTAL

$$\frac{\mathbf{M_1} \preceq \mathbf{M_1'} \qquad \mathbf{M_2} \preceq \mathbf{M_2'}}{\mathbf{M_1} \,{}^{\mathbf{d_1}}\!\oplus_{\mathsf{a}}^{\mathbf{d_2}}\, \mathbf{M_2} \preceq \mathbf{M_1'} \,{}^{\mathbf{d_1}}\!\oplus_{\mathsf{a}}^{\mathbf{d_2}}\, \mathbf{M_2'}}$$

REC-LINK-HORIZONTAL

$$\frac{\mathsf{M}_1 \preceq \mathsf{M}_1' \qquad \mathsf{M}_2 \preceq \mathsf{M}_2'}{\mathsf{M}_1 \,{}^{\mathsf{d}_1}\!\oplus_{\mathsf{r}}^{\mathsf{d}_2}\, \mathsf{M}_2 \preceq \mathsf{M}_1' \,{}^{\mathsf{d}_1}\!\oplus_{\mathsf{r}}^{\mathsf{d}_2}\, \mathsf{M}_2'}$$

REC-WRAPPER-COMPAT

$$\frac{\mathsf{M} \preceq \mathsf{M}'}{\lceil \mathsf{M} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \preceq \lceil \mathsf{M}' \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}}$$

COMPILER-CORRECT

$$\frac{\downarrow \mathsf{R} \ \textit{defined}}{[\![\downarrow \mathsf{R}]\!]_{\mathsf{a}} \preceq \lceil [\![\mathsf{R}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}}$$

REC-TO-ASM-LINK

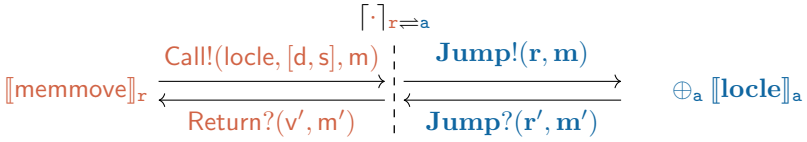
$$\lceil \mathsf{M}_1 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \oplus_{\mathsf{a}} \lceil \mathsf{M}_2 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \preceq \lceil \mathsf{M}_1 \oplus_{\mathsf{r}} \mathsf{M}_2 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$$

Figure 21.4: Proof rules of DimSum (with $M_1 \equiv M_2$ for $(M_1 \preceq M_2 \wedge M_2 \preceq M_1)$).

Let us take a closer look at the synchronization. Suppose we are linking two **Asm**-modules $\mathbf{M_1}$ and $\mathbf{M_2}$, and $\mathbf{M_1}$ is currently executing. If it wants to execute a jump, then it will emit the event $\mathbf{Jump!(r, m)}$ where the value of the program counter $\mathbf{r(pc)}$ indicates the destination. If the destination is in the instructions of $\mathbf{M_2}$, *i.e.*, in $\mathbf{d_2}$, then $\mathbf{M_2}$ gets to accept the jump event by emitting the dual event $\mathbf{Jump?(r, m)}$. In this case, the two components have synchronized, exchanging the values of the registers $\mathbf{r}$ and the memory $\mathbf{m}$. To the outside, the synchronization will be hidden: the combined module $\mathbf{M_1} \,{}^{\mathbf{d_1}}\!\oplus_{\mathsf{a}}^{\mathbf{d_2}}\, \mathbf{M_2}$ will do a silent τ-step. If the module $\mathbf{M_1}$ decides to jump *outside* $\mathbf{M_2}$, *i.e.*, outside $\mathbf{d_2}$, then $\mathbf{M_1} \,{}^{\mathbf{d_1}}\!\oplus_{\mathsf{a}}^{\mathbf{d_2}}\, \mathbf{M_2}$ will simply forward the jump to the environment.

There is one additional, subtle property of the linking operator that is used in going from (1) to (2): *horizontal compositionality* of $\mathbf{M_1} \oplus_{\mathsf{a}} \mathbf{M_2}$ (ASM-LINK-HORIZONTAL). Horizontal compositionality in DimSum means compatibility with the refinement. We will see several semantic linking operators in DimSum (*i.e.*, $\mathbf{M_1} \oplus_{\mathsf{a}} \mathbf{M_2}$, $\mathsf{M}_1 \oplus_{\mathsf{r}} \mathsf{M}_2$, and $\mathsf{M}_1 \oplus_{\mathsf{coro}} \mathsf{M}_2$,) and they will all be horizontally compositional. In fact, all these linking operators will be derived from a single, language-generic linking operator that is horizontally compositional (see §22.3). But no need to get ahead of ourselves.

Module translation [(2) to (3), generic]. In the next step, we reap the benefits of the semantic linking operator: we can link modules that are not *syntactically* **Asm**-libraries, but *semantically* are **Asm**-modules. More precisely, in this step we take the **Asm**-modules $[\![\downarrow \mathsf{main}]\!]_{\mathsf{a}}$ and $[\![\downarrow \mathsf{memmove}]\!]_{\mathsf{a}}$ obtained through compilation of the Rec-libraries main and memmove, and then we turn them into Rec-modules $[\![\mathsf{main}]\!]_{\mathsf{r}}$ and $[\![\mathsf{memmove}]\!]_{\mathsf{r}}$ inside a *semantic wrapper* $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$. The semantic wrapper $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ is an embedding of Rec modules into **Asm** (*i.e.*, if M is a Rec module, then $\lceil \mathsf{M} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ is an **Asm** module), and as such translates between Rec-events and **Asm**-events on the fly:



Conceptually, this wrapper is similar to a wrapper in a multi-language semantics of Matthews and Findler:[7] it embeds constructs from one language into another. The key distinction of the wrappers in DimSum is that they are *semantic*: they operate on modules (*i.e.*, transition systems) instead of syntactic constructs. As a result, their task is to translate interactions (*i.e.*, events) between the two languages. Take the interaction of memmove and **locle** depicted above. In this case, the Rec module issues a call to the function locle with arguments d and s and the memory m. The wrapper $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$ then constructs the corresponding **Asm** jump event, including the correct representation of the registers **r** and of the memory **m**. When **locle** eventually jumps back, the wrapper translates the jump to a corresponding function return. (We will discuss these translations in more detail in §21.3.)

[7] Matthews and Findler, "Operational Semantics for Multi-Language Programs", 2007 [MF07].

The wrapper $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$ has two important properties. The first (see COMPILER-CORRECT) is that the compiled program refines the source program wrapped by $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$—*i.e.*, our compiler is *correct* up to the translation of the wrapper. More specifically, a (syntactically) compiled Rec library $\downarrow \mathsf{R}$ behaves like the semantically translated source module $\lceil[\![\mathsf{R}]\!]_{\mathsf{r}}\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$. The second (see REC-WRAPPER-COMPAT) is that the wrapper is compatible with refinement—this property will be used by the following steps.

Abstracting implementations [(3) to (4), example-specific]. In the next step, we replace the assembly libraries **print** and **locle** with high-level specifications written in Spec. We do so to abstract over the **Asm** implementation details of both libraries, since we only care about their *interaction behavior* with other modules (*e.g.*, their jumps, which values they compute, which syscalls they trigger). Formally, we prove:

PRINT-CORRECT $[\![\mathbf{print}]\!]_{\mathsf{a}} \preceq [\![\mathbf{print}_{\mathsf{spec}}]\!]_{\mathsf{s}}$ LOCLE-CORRECT $[\![\mathbf{locle}]\!]_{\mathsf{a}} \preceq \lceil[\![\mathsf{locle}_{\mathsf{spec}}]\!]_{\mathsf{s}}\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$

The specifications for **print** and **locle** are depicted in Figure 21.5.

$$
\begin{aligned}
\mathbf{print}_{\mathsf{spec}} \triangleq_{\mathsf{coind}}\ & \exists \mathbf{r}, \mathbf{m}; \mathsf{vis}(\mathbf{Jump?}(\mathbf{r}, \mathbf{m})); \mathsf{assume}(\mathbf{r}(\mathbf{pc}) = a_{\mathsf{print}});\\
& \mathsf{vis}(\mathbf{Syscall!}(\mathbf{PRINT}, \mathbf{r}(\mathbf{x0}), \mathbf{m})); \exists \mathbf{v}, \mathbf{m}'; \mathsf{vis}(\mathbf{SyscallRet?}(\mathbf{v}, \mathbf{m}'));\\
& \mathsf{vis}(\mathbf{Jump!}(\mathbf{r}[\mathbf{pc} \mapsto \mathbf{r}(\mathbf{x30})][\mathbf{x0} \mapsto \mathbf{v}][\mathbf{x8} \mapsto *], \mathbf{m}')); \mathbf{print}_{\mathsf{spec}}\\
\mathsf{locle}_{\mathsf{spec}} \triangleq_{\mathsf{coind}}\ & \exists \mathsf{f}, \overline{v}, \mathsf{m}; \mathsf{vis}(\mathsf{Call?}(\mathsf{f}, \overline{v}, \mathsf{m})); \mathsf{assume}(\mathsf{f} = \mathsf{locle}); \mathsf{assume}(\overline{v} \text{ is } [\ell_1, \ell_2]);\\
& \mathsf{if}\ \ell_1.\mathsf{blockid} = \ell_2.\mathsf{blockid}\ \mathsf{then}\ \mathsf{vis}(\mathsf{Return!}(\ell_1.\mathsf{offset} \leq \ell_2.\mathsf{offset}, \mathsf{m}));\ \mathsf{locle}_{\mathsf{spec}}\\
& \mathsf{else}\ \exists \mathsf{b}; \mathsf{vis}(\mathsf{Return!}(\mathsf{b}, \mathsf{m}));\ \mathsf{locle}_{\mathsf{spec}}
\end{aligned}
$$

Figure 21.5: Specifications for **print** and **locle**.

Before we consider the details of these specifications, we should discuss one fundamental difference that stands out: $\mathbf{print}_{\mathsf{spec}}$ is an **Asm**-level specification, while $\mathsf{locle}_{\mathsf{spec}}$ is a Rec-level specification. We can give a Rec specification to **locle**, because it has the *interaction behavior* of a Rec function. More precisely, while we cannot implement **locle** in Rec directly, we can still give it a Rec-level specification in Spec, because **locle** obeys the calling convention of Rec and triggers no syscalls. In contrast, the same cannot be said for **print**, because **print** does a syscall, which is beyond the interaction behavior of Rec.

Let us now turn to the details of both specifications. The specification $\mathbf{print}_{\mathsf{spec}}$ accepts jumps to the start of the print code address. Then it triggers a print syscall of the contents of **x0** and accepts the return value **v**, which is subsequently returned (by storing it in **x0**). The return address is then fetched from register **x30** and becomes the next program counter **pc**. Afterwards, the specification starts from the beginning again (*i.e.*, with $\mathbf{print}_{\mathsf{spec}}$). The last step is important to *reuse the module* for subsequent executions of **print**. It is made possible, because our programs in Spec are *co-inductive*, so $\mathbf{print}_{\mathsf{spec}}$ can refer to itself in its own definition.

The specification $\mathsf{locle}_{\mathsf{spec}}$ accepts an incoming function call to locle where the arguments are two locations ℓ_1 and ℓ_2. If the locations point to *the same block in memory* (*i.e.*, their block identifiers are the same), then $\mathsf{locle}_{\mathsf{spec}}$ compares their offsets and returns the result. Afterwards, the specification loops. If the locations point to *different blocks in memory*, then $\mathsf{locle}_{\mathsf{spec}}$ non-deterministically chooses a Boolean b and returns it. The non-deterministic choice here abstracts over the implementation detail of how exactly the Rec locations are mapped to the concrete **Asm** memory. This non-deterministic choice does not cause problems when verifying memmove since s and d cannot overlap if they point to different blocks and thus the result of locle is irrelevant.[8]

[8] The Coq development additionally verifies a stronger version of $\mathsf{locle}_{\mathsf{spec}}$ that gives a consistent ordering of locations across multiple calls.

Leaving assembly behind [(4) to (6), generic]. In the next two steps, we exploit the fact that $[\![\mathsf{main}]\!]_{\mathtt{r}}$, $[\![\mathsf{memmove}]\!]_{\mathtt{r}}$, and $[\![\mathsf{locle}_{\mathsf{spec}}]\!]_{\mathtt{s}}$ obey the Rec interaction behavior: we lift them out of **Asm** to reason about them at the level of Rec in the next step. To do so, we introduce two Rec-linking operators: syntactic linking ($\mathsf{R}_1 \cup_{\mathtt{r}} \mathsf{R}_2$) and semantic linking ($\mathsf{M}_1 \,{}^{\mathsf{d}_1}\!\oplus^{\mathsf{d}_2}_{\mathtt{r}}\, \mathsf{M}_2$), analogous to **Asm**.[9] We use the linking operators to combine the three Rec-modules into the module $[\![\mathsf{main} \cup_{\mathtt{r}} \mathsf{memmove}]\!]_{\mathtt{r}} \oplus_{\mathtt{r}} [\![\mathsf{locle}_{\mathsf{spec}}]\!]_{\mathtt{s}}$, leveraging that syntactic and semantic linking coincide for Rec-libraries (see REC-LINK-SYN), that $\mathsf{M}_1 \oplus_{\mathtt{r}} \mathsf{M}_2$ is horizontally compositional (see REC-LINK-HORIZONTAL), and that the wrapper $\lceil\cdot\rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}}$ is compatible with linking (see REC-TO-ASM-LINK).

[9] Here, d_1 and d_2 refer to the function names of M_1 and M_2. We often omit them to avoid clutter.

In a typical verification task, we want to leave the level of assembly as much as possible. The reason is that it is simpler to reason about programs at the level of Rec than it is to reason about them at the level of **Asm**. In particular, when we reason about programs at the level of Rec, we do not have to think about the surrounding wrapper $\lceil\cdot\rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}}$.

High-level reasoning [(6) to (7), example-specific]. In the next step, we can reap the benefits from reasoning at the level of Rec. More specifically, we can ignore that $[\![\mathsf{main}]\!]_{\mathtt{r}}$, $[\![\mathsf{memmove}]\!]_{\mathtt{r}}$, and $[\![\mathsf{locle}_{\mathsf{spec}}]\!]_{\mathtt{s}}$ are inside **Asm** (using the wrapper $\lceil\cdot\rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}}$) and instead reason about *their interactions* at the level of Rec. We can abstract over their implementation details and show:

$$\textsc{main-correct}\ [\![\mathsf{main} \cup_{\mathtt{r}} \mathsf{memmove}]\!]_{\mathtt{r}} \oplus_{\mathtt{r}} [\![\mathsf{locle}_{\mathsf{spec}}]\!]_{\mathtt{s}} \preceq [\![\mathsf{main}_{\mathsf{spec}}]\!]_{\mathtt{s}}$$

Here, $\mathsf{main}_{\mathsf{spec}}$ is a Spec specification for the three modules with Rec events:

$$\begin{aligned}\mathsf{main}_{\mathsf{spec}} \triangleq\ & \exists \mathsf{f}, \overline{v}; \mathsf{vis}(\mathsf{Call?}(\mathsf{f}, \overline{v}, *)); \mathsf{assume}(\mathsf{f} = \mathsf{main}); \mathsf{assume}(\overline{v} = []); \\ & \exists \mathsf{m}_1; \mathsf{vis}(\mathsf{Call!}(\mathsf{print}, [1], \mathsf{m}_1)); \exists \mathsf{m}_2; \mathsf{vis}(\mathsf{Return?}(*, \mathsf{m}_2)); \mathsf{assume}(\mathsf{m}_2 = \mathsf{m}_1); \\ & \exists \mathsf{m}_3; \mathsf{vis}(\mathsf{Call!}(\mathsf{print}, [2], \mathsf{m}_3)); \exists \mathsf{m}_4; \mathsf{vis}(\mathsf{Return?}(*, \mathsf{m}_4)); \mathsf{assume}(\mathsf{m}_4 = \mathsf{m}_3); \mathsf{any}\end{aligned}$$

The combined module will accept any incoming call to the main function. Subsequently, it will call print with argument 1 and some memory m_1, and expect print to return with the same memory.[10] Subsequently, the specification will call print with argument 2, accept the corresponding return, and afterwards its behavior can be arbitrary.[11]

[10] Returning with a different memory $\mathsf{m}_2 \neq \mathsf{m}_1$ will be accepted, but in this case the assume fails, and the specification does provide not any additional guarantees about the behavior of the program.

[11] Technically, this specification also needs to accept incoming calls after the call to print and behave arbitrarily in this case, but we omit this here for simplicity.

Reasoning with specifications [(7) to (8), example-specific]. In a final step, we turn back to the $\mathbf{print}_{\mathsf{spec}}$ module. Recall that the module relies on interaction fundamentally not available at the level of Rec—syscalls—which is why we reason about it at the level of **Asm**. Fittingly, we also have to reason about $\lceil [\![\mathsf{main}_{\mathsf{spec}}]\!]_{\mathtt{s}} \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}} \oplus_{\mathtt{a}} [\![\mathbf{print}_{\mathsf{spec}}]\!]_{\mathtt{s}}$ at the level of **Asm**. Typically, reasoning about programs at the level of **Asm** can be a daunting task, since there are many low-level details to consider. However, since we have already condensed the other modules into a single, specification $\mathsf{main}_{\mathsf{spec}}$, the last step is relatively straightforward:

$$\lceil [\![\mathsf{main}_{\mathsf{spec}}]\!]_{\mathtt{s}} \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}} \oplus_{\mathtt{a}} [\![\mathbf{print}_{\mathsf{spec}}]\!]_{\mathtt{s}} \preceq [\![\mathbf{onetwo}_{\mathsf{spec}}]\!]_{\mathtt{s}}$$

In the proof, we essentially only have to make sure that the calls in main and the jumps in **print** line up. The translation of the events is taken care of by the wrapper $\lceil \cdot \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$, which we discuss next.

21.3 Semantic Language Wrappers

$$\mathsf{Call!}(\mathsf{f}, \overline{v}, \mathsf{m}) \rightharpoonup_w \mathbf{Jump!}(\mathbf{r}, \mathbf{m}) \triangleq \mathbf{r}(\mathbf{pc}) = a_{\mathsf{f}} \wedge \overline{v} \sim_w \mathbf{r}(\mathbf{x0} \ldots \mathbf{x8}) \wedge |\overline{v}| \leq 9 \wedge \mathsf{m} \sim_w \mathbf{m}$$

$$\mathsf{Return?}(\mathsf{v}, \mathsf{m}) \leftharpoonup_{\mathbf{r}', w} \mathbf{Jump?}(\mathbf{r}, \mathbf{m}) \triangleq \begin{array}{c} \mathbf{r}(\mathbf{pc}) = \mathbf{r}'(\mathbf{x30}) \wedge \mathsf{v} \sim_w \mathbf{r}(\mathbf{x0}) \wedge \mathsf{m} \sim_w \mathbf{m} \wedge \\ \mathbf{r}(\mathbf{x19} \ldots \mathbf{x29}, \mathbf{sp}) = \mathbf{r}'(\mathbf{x19} \ldots \mathbf{x29}, \mathbf{sp}) \end{array}$$

Figure 21.6: Select cases of the calling convention between Rec and **Asm**.

One of the most important building blocks of the proof in §21.2 is the wrapper $\lceil \cdot \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$, which converts events from Rec to **Asm** *and back*. In this section, we take a closer look at how the wrapper works. Recall the event exchange between memmove and **locle** (in §21.2). In this exchange, the wrapper has to translate between (1) the calling conventions of both languages (*e.g.*, calls and returns in Rec are jumps in **Asm**), (2) the values of both languages (*e.g.*, structured values v in Rec are integers **v** in **Asm**), and (3) the memory models of both languages (*e.g.*, the block-based memory m in Rec is a flat memory **m** in **Asm**). As we will discuss below, the two key ingredients to getting this translation right are *Kripke relations* (explained using the direction Rec-to-**Asm**) and *angelic non-determinism* (explained using the direction **Asm**-to-Rec). In this section, we describe a simplified account of the wrapper $\lceil \cdot \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$. Its actual

definition is derived from the language generic combinators presented in §22.3.

Kripke relations. Let us start with the direction of translating Rec events into **Asm** events (*e.g.*, translating Call!(locle, [d, s], m) into **Jump!(r, m)**). In principle, this direction is relatively straightforward, because we go from a high-level language with more structure to a low-level language with less structure (*e.g.*, we map structured values v to integers **v**). The main challenge in this direction is that the wrapper has to maintain a mapping from Rec-level block identifiers to **Asm**-level addresses, which remains consistent *across function calls*. That is, if we translate the location ℓ to the address **v** once, then we have to ensure that we pick **v** again for subsequent calls exposing ℓ, because assembly libraries typically expect the location ℓ to not move in between function calls.

To maintain a consistent mapping across function calls, the wrapper $\lceil\cdot\rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}}$ keeps around a block-identifier-to-address map w.[12] In the simplified account of $\lceil\cdot\rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}}$ that we discuss here, one can think of the map w as one component of the *internal state of the wrapper*. For example, in the case of translating outgoing calls to jumps, the wrapper transitions as follows:

[12] In the full definition of the wrapper, this piece of state w is maintained using a separation logic relation, as discussed in §22.3.

$$\frac{\sigma \xrightarrow{\mathsf{Call!}(\mathsf{f},\overline{v},\mathsf{m})}_{M} \sigma' \qquad \mathsf{Call!}(\mathsf{f},\overline{v},\mathsf{m}) \rightharpoonup_{w'} \mathbf{Jump!}(\mathbf{r},\mathbf{m}) \qquad w \subseteq w'}{(\mathsf{rec}, w, \sigma) \xrightarrow{\mathbf{Jump!}(\mathbf{r},\mathbf{m})}_{\lceil M\rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}}} (\mathbf{asm}(\mathbf{r}), w', \sigma')} \text{CALL-ASM}$$

Here, the state of the wrapper contains information about who is currently executing (*e.g.*, rec or **asm(r)**), the address mapping w, and the state of the wrapped module σ.[13] As the wrapper executes, the mapping w gradually grows along with the memories, written $w \subseteq w'$. In the context of Kripke relations, the state w is typically called a *world* and the relation $w \subseteq w'$ is *world extension*.

[13] The reason why we record the registers **r** in **asm(r)** will become apparent below.

The relation $\mathsf{Call!}(\mathsf{f},\overline{v},\mathsf{m}) \rightharpoonup_w \mathbf{Jump!}(\mathbf{r},\mathbf{m})$ in CALL-ASM, defined in Figure 21.6, encodes a part of the calling convention of Rec and **Asm**. To define it, we first relate values between both languages:

$$z \sim_w z \qquad b \sim_w (\mathsf{if}\ b\ \mathsf{then}\ 1\ \mathsf{else}\ 0) \qquad \ell \sim_w w(\ell.\mathsf{blockid}) + \ell.\mathsf{offset}$$

In the case of locations ℓ, we look up the base address for the block in the mapping w. The relation can then be lifted to memories, written $\mathsf{m} \sim_w \mathbf{m}$. To translate a call from Rec to **Asm** with $\mathsf{Call!}(\mathsf{f},\overline{v},\mathsf{m}) \rightharpoonup_w \mathbf{Jump!}(\mathbf{r},\mathbf{m})$ we have to translate the components as follows: The program counter must point to the start address of the function f, the argument values must be stored in the registers **x0** to **x8**, there may be at most nine arguments,[14] and the memories must be related. While this definition does incorporate quite a number of technical details about the calling conventions of both languages, there is no way around it: when we call an **Asm** program, we have to make sure that its expectations are met, which includes satisfying the calling convention.

[14] The calling convention of **Asm** restricts functions to nine registers. We rule out Rec functions with more than nine arguments in the compiler and restrict the number of function arguments in the wrapper.

Angelic non-determinism. Let us now turn to the reverse direction (**Asm**-to-Rec). This direction is more challenging because we need to

"guess" the additional structure of the representation at the level of Rec. For example, consider translating $\mathbf{Jump?}(\mathbf{r}, \mathbf{m})$ to $\mathsf{Return?}(\mathsf{v}, \mathsf{m})$ (*e.g.*, when **locle** returns from **Asm**). In this translation, the return value is stored in **x0** as *an integer* and we need to pick "the right" Rec return value v. The issue is that there can be multiple candidates, but not all will work. For instance, if **locle** returns **0** (*i.e.*, the first location is not less-or-equal to the second) and this **0** is subsequently translated to a location ℓ_0 instead of the Boolean false, then memmove will have undefined behavior. Unfortunately, the wrapper cannot choose the right v by itself, because locally, it knows possible candidates (*e.g.*, false, 0, and ℓ_0), but it does not know which one will work "down the road". (Also, Rec is untyped, so there is no type system to help with this choice.) To help the wrapper out, we delegate the choice to a well-meaning angel: we use *angelic non-determinism*.[15]

[15] Floyd, "Nondeterministic Algorithms", 1967 [Flo67]; Back, "Changing Data Representation in the Refinement Calculus", 1989 [Bac89].

Before we discuss angelic non-determinism, let us first explain the calling convention for this direction (see Figure 21.6). The relation for this case, $(\rightharpoonup_{\mathbf{r}',w})$, takes an additional piece of state: the register state $\mathbf{r}'$ that we record in $\mathbf{asm}(\mathbf{r}')$ when calling **Asm**-code (see CALL-ASM). The relation requires the program counter to point to the original return address (in **x30**), the return values and memories to be related, and the callee-saved registers $\mathbf{x19}, \ldots, \mathbf{x29}, \mathbf{sp}$ to be restored.

Let us turn to *angelic non-determinism* and how it helps us here. To return from **Asm**, we have to define the analogue of CALL-ASM but for $\mathsf{Return?}(\mathsf{v}, \mathsf{m}) \rightharpoonup_{\mathbf{r}',w} \mathbf{Jump?}(\mathbf{r}, \mathbf{m})$. However, if we follow the structure of CALL-ASM, then event translation would become a *proof obligation for the wrapper* including choosing v and m. That is, applying the hypothetical rule would lead to the obligation:

$$\text{"}\exists \mathsf{v}, \mathsf{m}.\ (\mathsf{Return?}(\mathsf{v}, \mathsf{m}) \rightharpoonup_{\mathbf{r}',w} \mathbf{Jump?}(\mathbf{r}, \mathbf{m})) \wedge \ldots\text{"}$$

However, what we want here is that the event translation becomes *an assumption of the wrapper* including "the right choices" for v and m. In other words, we want something like:

$$\text{"}\forall \mathsf{v}, \mathsf{m}.\ (\mathsf{Return?}(\mathsf{v}, \mathsf{m}) \rightharpoonup_{\mathbf{r}',w} \mathbf{Jump?}(\mathbf{r}, \mathbf{m})) \Rightarrow \ldots\text{"}$$

Unfortunately, we cannot literally define an analogue of CALL-ASM using this precondition, because then there could only be a single successor state σ' for all possible memories m and values v. There are, however, typically multiple candidates σ' depending on the choices of m and v. Since the wrapper does not know how to choose m and v itself, the only sensible option is to continue in *all possible states* σ' under the assumption of $\mathsf{Return?}(\mathsf{v}, \mathsf{m}) \rightharpoonup_{\mathbf{r}',w} \mathbf{Jump?}(\mathbf{r}, \mathbf{m})$. That is exactly what *angelic non-determinism* allows us to do. (We make formal how in the next chapter, §22.1.) It will then be the job of the angel to pick the right m and v, and thereby choose one of the states σ'.

Of course, we cannot keep delegating the responsibility to make "the right choices" to the angel forever. Eventually, we, the user of DimSum, have to slip into the role of the angel and provide "the right choices". In this case, we do so in proving REC-TO-ASM-LINK (*i.e.*, $\lceil \mathsf{M}_1 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \oplus_{\mathsf{a}} \lceil \mathsf{M}_2 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \preceq \lceil \mathsf{M}_1 \oplus_{\mathsf{r}} \mathsf{M}_2 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$). Consider the case where M_2 returns to M_1. In terms of

events, this means we come from Rec, go through **Asm**, and then return to Rec again. This path allows us as the user to observe the right choice: the memory m and value v will be determined by M_2 and we, as the angel, can then forward them to M_1. Inspired by CCR,[16] this use of angelic non-determinism allows us to express rely-guarantee protocols between modules and their environment.

[16] Song et al., "Conditional Contextual Refinement", 2023 [Son+23].

Chapter 22

Modules and Refinement

In this chapter, we discuss the formal definition of modules and simulation (in §22.1), the meaning of non-deterministic choices (in §22.2), and the library of compositional combinators (in §22.3).

22.1 Modules and Refinement in the Abstract

$$\frac{\sigma \in \Sigma}{\sigma \xrightarrow{\mathsf{nil}}{}^{*}_{M}\ \Sigma} \qquad \frac{\exists \Sigma'.\ \sigma \xrightarrow{\alpha_1}_M \Sigma' \wedge \forall \sigma' \in \Sigma'.\ \sigma' \xrightarrow{\overline{e_2}}{}^{*}_{M}\ \Sigma}{\sigma \xrightarrow{\alpha_1 ::_? \overline{e_2}}{}^{*}_{M}\ \Sigma}$$

Figure 22.1: Multistep execution $\sigma \xrightarrow{\overline{e}}{}^{*}\ \Sigma$ with $\alpha_1 ::_? \overline{e_2} \triangleq \mathsf{if}\ \alpha_1 = e_1\ \mathsf{then}\ e_1 :: \overline{e_2}\ \mathsf{else}\ \overline{e_2}$.

A module $M \in \mathsf{Module}(E)$ is a labeled transition system with events drawn from the set E and demonic and angelic non-determinism. Formally, a module $M = (S, \to, \sigma^0)$ consists of a set of states S, an initial state σ^0, and a transition relation $\to \in \mathcal{P}(S \times (E \uplus \{\tau\}) \times \mathcal{P}(S))$. The labels type $\alpha \in E \uplus \{\tau\}$ indicates that each transition can either emit a visible event $e \in E$ or be silent, denoted by τ. Notably, the transitions of a module $\sigma \xrightarrow{\alpha} \Sigma$ go from a single state σ to a *set of states* Σ. The use of a set is inspired by alternating automata[1] and binary multi-relations[2] as a means to incorporate both demonic and angelic non-determinism. Demonic non-determinism works "as usual": a single state σ can transition to multiple sets Σ (*i.e.*, $\sigma \xrightarrow{\alpha} \Sigma$ and $\sigma \xrightarrow{\alpha'} \Sigma'$ where $\Sigma \neq \Sigma'$). Angelic non-determinism works differently: after the transition $\sigma \xrightarrow{\alpha} \Sigma$, the module is in *every state* $\sigma' \in \Sigma$. This intuition becomes precise when we consider multistep executions of a module, depicted in Figure 22.1: we pick some successor set Σ and then proceed for every possible $\sigma' \in \Sigma$.

[1] Chandra et al., "Alternation", 1981 [CKS81]; Vardi, "Alternating Automata and Program Verification", 1995 [Var95].

[2] Rewitzky, "Binary Multirelations", 2003 [Rew03].

Simulation. For modules $M_1, M_2 \in \mathsf{Module}(E)$, we define refinement as the simulation ($\preceq_{\mathrm{co}}$):

$$M_1 \preceq M_2 \triangleq (M_1, \sigma^0_{M_1}) \preceq_{\mathrm{co}} (M_2, \sigma^0_{M_2})$$
$$(M_1, \sigma_1) \preceq_{\mathrm{co}} (M_2, \sigma_2) \triangleq_{\mathsf{coind}} \forall e, \Sigma_1.\ \sigma_1 \xrightarrow{\alpha}_{M_1} \Sigma_1 \Rightarrow \exists \Sigma_2.\ \sigma_2 \xrightarrow{\alpha}{}^{*}_{M_2} \Sigma_2 \wedge \forall \sigma_2' \in \Sigma_2.\ \exists \sigma_1' \in \Sigma_1.\ (M_1, \sigma_1') \preceq_{\mathrm{co}} (M_2, \sigma_2')$$

Here, ($\preceq_{\mathrm{co}}$) is a coinductive simulation: For every step of the implementation $\sigma_1 \xrightarrow{\alpha}_{M_1} \Sigma_1$ with label α, the simulation demands a corresponding multistep of the specification $\sigma_2 \xrightarrow{\alpha}{}^{*}_{M_2} \Sigma_2$[3]. This part of the definition is

[3] $\sigma \xrightarrow{\alpha}{}^{*}_{M} \Sigma$ is defined as $\sigma \xrightarrow{\alpha ::_? \mathsf{nil}}{}^{*}_{M} \Sigma$

standard for a simulation with demonic non-determinism. Then the sides flip, and for every possible *successor state of the specification* $\sigma_2' \in \Sigma_2$, the simulation demands a corresponding state $\sigma_1' \in \Sigma_1$ such that σ_1' and σ_2' are in the simulation again. This second part is only present in simulations with angelic non-determinism.[4]

[4] The definition of this simulation is inspired by Alur et al., "Alternating Refinement Relations", 1998 [Alu+98] and Fritz and Wilke, "Simulation relations for alternating Büchi automata", 2005 [FW05].

The simulation is a preorder:

Lemma 4 *$M_1 \preceq M_2$ is reflexive and transitive.*

Simulation vs. trace refinement. The reader might wonder why we center our reasoning around a simulation instead of another form of refinement (*e.g.*, a contextual refinement or a trace refinement). In DimSum, we work with a simulation, because simulations are sensitive to branching (*i.e.*, the order of visible events and non-deterministic choices) and branching sensitivity is crucial to implement linking as event-synchronization, as we will see in §22.2. Fortunately, the simulation $M_1 \preceq M_2$ strikes exactly the right balance: it is large enough to contain our desired examples (*e.g.*, the compiler passes in §24 and the coroutine linking operator in §23.3), it is compositional enough to be compatible with the operators of DimSum (see §22.3), and it is small enough to imply a traditional whole-program trace refinement. More specifically, we define whole-program trace refinement as $M_1 \sqsubseteq_{\mathcal{T}} M_2 \triangleq \mathcal{T}(M_1) \subseteq \mathcal{T}(M_2)$ where $\mathcal{T}(M) \triangleq \left\{\overline{e} \,\middle|\, \sigma^0 \xrightarrow{\overline{e}}{}^* S_M\right\}$ and, as to be expected, we obtain:

Theorem 5 *If $M_1 \preceq M_2$, then $M_1 \sqsubseteq_{\mathcal{T}} M_2$.*

22.2 Angelic and Demonic Non-Determinism

As we have discussed in §21.3 and §22.1, modules in DimSum have two kinds of non-determinism: *demonic* and *angelic non-determinism*. To understand when we want to use one vs. the other and how they affect proofs of the simulation $M_1 \preceq M_2$, we discuss them in the context of a concrete example: the specification language Spec. As mentioned in §21.1, we have so far only discussed a fragment of Spec. Formally, the full language is defined coinductively as follows:

$$\mathsf{Spec}(E) \ni p ::=_{\mathsf{coind}} \mathsf{vis}(e); p \mid \exists x : T; p(x) \mid \forall x : T; p(x) \qquad (e \in E)$$

As before, $\mathsf{vis}(e); p$ emits a visible event $e \in E$. The program $\exists x : T; p(x)$ uses demonic non-determinism—think "$\exists$" reminiscent of the devil's trident Ψ—to choose some $x : T$ and then proceed as p. The program $\forall x : T; p(x)$ uses angelic non-determinism—think "$\forall$" for an inverted A for angel—to assume a choice $x : T$ and then proceed as p. Besides the analogy, as we will see shortly, the symbols "$\forall$" and "$\exists$" also have a more literal reading as quantifiers in the context of the simulation.

Formally, the module semantics of a program $p \in \mathsf{Spec}(E)$ is a module $[\![p]\!]_{\mathsf{s}} \triangleq (\mathsf{Spec}(E), \to_{\mathsf{s}}, p)$ where programs execute according to the following transition system:

$$(\mathsf{vis}(e); p) \xrightarrow{e}_{\mathsf{s}} \{p\} \qquad (\exists x : T; p(x)) \xrightarrow{\tau}_{\mathsf{s}} \{p(y)\} \text{ (for } y \in T) \qquad (\forall x : T; p(x)) \xrightarrow{\tau}_{\mathsf{s}} \{p(y) \mid y \in T\}$$

$$\textsc{sim-vis}\ \frac{[\![p]\!]_{\mathsf{s}} \preceq [\![p']\!]_{\mathsf{s}}}{[\![\mathsf{vis}(e); p]\!]_{\mathsf{s}} \preceq [\![\mathsf{vis}(e); p']\!]_{\mathsf{s}}} \qquad \textsc{sim-ex-r}\ \frac{\exists y \in T.\, M \preceq [\![p(y)]\!]_{\mathsf{s}}}{M \preceq [\![\exists x : T; p(x)]\!]_{\mathsf{s}}}$$

$$\textsc{sim-ex-l}\ \frac{\forall y \in T.\, [\![p(y)]\!]_{\mathsf{s}} \preceq M}{[\![\exists x : T; p(x)]\!]_{\mathsf{s}} \preceq M} \qquad \textsc{sim-all-r}\ \frac{\forall y \in T.\, M \preceq [\![p(y)]\!]_{\mathsf{s}}}{M \preceq [\![\forall x : T; p(x)]\!]_{\mathsf{s}}}$$

$$\textsc{sim-all-l}\ \frac{\exists y \in T.\, [\![p(y)]\!]_{\mathsf{s}} \preceq M}{[\![\forall x : T; p(x)]\!]_{\mathsf{s}} \preceq M}$$

Figure 22.2: Derived quantifier elimination/introduction rules for Spec-programs.

We embed constructs of our meta theory (*e.g.*, $\mathsf{if}\ \phi\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$) into Spec[5] and, hence, we can derive the remaining constructs of Spec shown in §21.1 using the three primitives:

$$\mathsf{any} \triangleq \mathsf{ub} \triangleq_{\mathsf{coind}} \forall x : \emptyset; \mathsf{ub} \qquad \mathsf{assume}(\phi); p \triangleq \mathsf{if}\ \phi\ \mathsf{then}\ p\ \mathsf{else}\ \mathsf{ub}$$
$$\mathsf{nb} \triangleq_{\mathsf{coind}} \exists x : \emptyset; \mathsf{nb} \qquad \mathsf{assert}(\phi); p \triangleq \mathsf{if}\ \phi\ \mathsf{then}\ p\ \mathsf{else}\ \mathsf{nb}$$

The specification any means the program can have *any behavior* or, in other words, the behavior is not defined (*i.e.*, ub). Formally, we can represent this behavior as an angelic choice over the empty set, because $M \preceq [\![\forall x : \emptyset; p]\!]_{\mathsf{s}}$ for any M (*cf.* the definition of $(\preceq)$). The specification nb means the program has finished executing or, in other words, the program has *no behavior* anymore. Formally, we can represent termination as a demonic choice over the empty set: there is no "next" state that the program can step to. We can then derive the constructs $\mathsf{assume}(\phi); p$ and $\mathsf{assert}(\phi); p$.[6]

Non-deterministic choices as quantifiers. As mentioned above, the notation for demonic and angelic choice in Spec is no accident: the different forms of non-determinism have a reading as logical connectives in a simulation. The interactions of the two kinds of non-determinism with simulation are depicted in Figure 22.2. If we read simulation "$\preceq$" as a form of implication "$\Rightarrow$", then the proof rules for the two quantifiers correspond to the introduction and elimination rules for universal and existential quantification of first-order logic. For example, existential quantification $\exists x : T; p$ on the left side (sim-ex-l) means we need to consider all possible choices of x, whereas existential quantification on the right side (sim-ex-r) means we need to choose x. Furthermore, the quantifiers validate and invalidate all the usual quantifier commuting principles shown in Figure 22.3.

The reading of non-deterministic choices as quantifiers generalizes beyond Spec. When we prove a simulation $M_1 \preceq M_2$, then demonic non-determinism in M_1 means we need to consider all possible choices; in M_2 it means we need to provide a particular choice. For angelic non-determinism, the rules are flipped. In M_1, we need to provide a particular choice; in M_2, we need to consider all possible choices. For example, in §21.3, we have discussed angelic non-determinism in the wrapper $\lceil \mathsf{M} \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$. Recall that angelic non-determinism in this case meant that the wrapper can

[5] Spec is a shallow embedding in Coq and therefore inherits its rich collection of datatypes (*e.g.*, $\mathbb{N}$, $\mathsf{list}(T)$, *etc.*) and functions.

[6] The Coq development uses the (classically) equivalent definitions $\mathsf{assume}(\phi); p \triangleq \forall \_ : \phi; p$ and $\mathsf{assert}(\phi); p \triangleq \exists \_ : \phi; p$.

SIM-ALL-ALL-COMM
$[\![\forall x; \forall y; p(x,y)]\!]_{\mathsf{s}} \preceq [\![\forall y; \forall x; p(x,y)]\!]_{\mathsf{s}}$

SIM-EX-EX-COMM
$[\![\exists x; \exists y; p(x,y)]\!]_{\mathsf{s}} \preceq [\![\exists y; \exists x; p(x,y)]\!]_{\mathsf{s}}$

SIM-EX-ALL-COMM
$[\![\exists x; \forall y; p(x,y)]\!]_{\mathsf{s}} \preceq [\![\forall y; \exists x; p(x,y)]\!]_{\mathsf{s}}$

NO-SIM-ALL-EX-COMM
$[\![\forall y; \exists x; p(x,y)]\!]_{\mathsf{s}} \not\preceq [\![\exists x; \forall y; p(x,y)]\!]_{\mathsf{s}}$

SIM-EX-VIS-COMM
$[\![\exists x; \mathsf{vis}(e); p(x)]\!]_{\mathsf{s}} \preceq [\![\mathsf{vis}(e); \exists x; p(x)]\!]_{\mathsf{s}}$

NO-SIM-VIS-EX-COMM
$[\![\mathsf{vis}(e); \exists x; p(x)]\!]_{\mathsf{s}} \not\preceq [\![\exists x; \mathsf{vis}(e); p(x)]\!]_{\mathsf{s}}$

SIM-VIS-ALL-COMM
$[\![\mathsf{vis}(e); \forall x; p(x)]\!]_{\mathsf{s}} \preceq [\![\forall x; \mathsf{vis}(e); p(x)]\!]_{\mathsf{s}}$

NO-SIM-ALL-VIS-COMM
$[\![\forall x; \mathsf{vis}(e); p(x)]\!]_{\mathsf{s}} \not\preceq [\![\mathsf{vis}(e); \forall x; p]\!]_{\mathsf{s}}$

Figure 22.3: Admissible and inadmissible quantifier commuting principles for Spec-programs.

assume "the right choice" is provided to it by the angel. When we prove the simulation $\lceil \mathsf{M}_1 \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}} \oplus_{\mathtt{a}} \lceil \mathsf{M}_2 \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}} \preceq \lceil \mathsf{M}_1 \oplus_{\mathtt{r}} \mathsf{M}_2 \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$, then we have to slip into the role of the angel: for calls from M_1 to M_2, we obtain the "right choice" of, *e.g.*, the memory m through demonic non-determinism in M_1 (think "$\exists \mathsf{m}$") and then we pass it on through angelic non-determinism in M_2 (think "$\forall \mathsf{m}$").

Branching-sensitivity and linking. What we have not discussed so far is the interaction of visible events and non-deterministic choices. As it turns out, it is crucial that the simulation $M_1 \preceq M_2$ preserves the order of visible events and certain choices. More specifically, the rules SIM-EX-VIS-COMM and SIM-VIS-ALL-COMM of Figure 22.3 are admissible whereas NO-SIM-VIS-EX-COMM and NO-SIM-ALL-VIS-COMM are not. Intuitively, the reason is that *linking* can "inline" an entire module in the place of a visible event e, so whenever we commute a quantifier over e, we are effectively commuting it over all the choices made "on the other side" of e.

To illustrate this point, let us consider a concrete example: we will show that if one admits the commuting forbidden by NO-SIM-ALL-VIS-COMM, then the simulation is trivial in the sense that $[\![p_1]\!]_{\mathsf{s}} \preceq [\![p_2]\!]_{\mathsf{s}}$ for any specifications p_1 and p_2. For this example, we define $p_{\mathsf{L}} \triangleq \forall x : \emptyset; \mathsf{vis}(A!); \mathsf{any}$ and $p_{\mathsf{R}} \triangleq \mathsf{vis}(A?); p_2$ and consider what happens when we link them together with a suitably defined linking operation (*i.e.*, one matching $A!$ with $A?$). Using the forbidden commuting, we show:

$$[\![p_1]\!]_{\mathsf{s}} \preceq [\![p_{\mathsf{L}} \oplus p_{\mathsf{R}}]\!]_{\mathsf{s}} \preceq [\![p_2]\!]_{\mathsf{s}}$$

For the first part, $[\![p_1]\!]_{\mathsf{s}} \preceq [\![p_{\mathsf{L}} \oplus p_{\mathsf{R}}]\!]_{\mathsf{s}}$, it suffices to observe that $p_{\mathsf{L}} \oplus p_{\mathsf{R}}$ can be implemented by *any* program p, because it starts with an angelic choice over an empty set. That is, suppose the linked program $p_{\mathsf{L}} \oplus p_{\mathsf{R}}$ starts executing on the left side. Then we are given $x \in \emptyset$ in the proof of $[\![p_i]\!]_{\mathsf{s}} \preceq [\![p_{\mathsf{L}} \oplus p_{\mathsf{R}}]\!]_{\mathsf{s}}$ (by SIM-ALL-R) and we are done. For the second part, $[\![p_{\mathsf{L}} \oplus p_{\mathsf{R}}]\!]_{\mathsf{s}} \preceq [\![p_2]\!]_{\mathsf{s}}$ we use the commuting rule NO-SIM-ALL-VIS-COMM. With the commuting rule, it suffices to show $[\![(\mathsf{vis}(A!); \forall x : \emptyset; \mathsf{any}) \oplus (\mathsf{vis}(A?); p_2)]\!]_{\mathsf{s}} \preceq [\![p_2]\!]_{\mathsf{s}}$, which follows from executing the module: we start on the left, synchronize on A and continue execution on the right, and then continue with p_2.

In summary, angelic and demonic non-determinism allow modules to express universal and existential quantification, which enables the local encoding of assumptions about their environment and guarantees about their own behavior (as used by the $\lceil \mathsf{M} \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$ wrapper). To ensure that the semantics of operations like linking can be meaningfully expressed as compositions of modules, DimSum relies on a branch-sensitive simulation that carefully controls the commutation of visible events and non-deterministic choices.

22.3 Combinators

$$\textsc{product-compat}\ \frac{M_1 \preceq M_1' \qquad M_2 \preceq M_2'}{M_1 \times M_2 \preceq M_1' \times M_2'} \qquad \textsc{filter-compat}\ \frac{M_1 \preceq M_1'}{M_1 \setminus M_2 \preceq M_1' \setminus M_2} \qquad \textsc{link-compat}\ \frac{M_1 \preceq M_1' \qquad M_2 \preceq M_2'}{M_1 \oplus_X M_2 \preceq M_1' \oplus_X M_2'} \qquad \textsc{wrapper-compat}\ \frac{M \preceq M'}{\lceil M \rceil_X \preceq \lceil M' \rceil_X}$$

Figure 22.4: Compositional combinator reasoning principles.

One of the strong suits of DimSum is that it comes with a compositional set of language-agnostic combinators. Concretely, DimSum provides out-of-the-box a combinator for the product $M_1 \times M_2$ of two modules M_1 and M_2, one for filtering $M_1 \setminus M_2$, one for linking $M_1 \oplus_X M_2$, and one for stateful wrappers $\lceil M \rceil_X$. The combinators we have encountered so far—$\oplus_{\mathtt{r}}$, $\oplus_{\mathtt{a}}$, and $\lceil \cdot \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$—are all language-specific instantiations of these generic combinators (see §23 and §24). To allow for the compositional reasoning we aim for in DimSum, all of them need to be compatible with simulation, *i.e.*, they need to be monotone with respect to $\preceq$, as asserted, for example, by ASM-LINK-HORIZONTAL. The main benefit of expressing the language-specific combinators as instances of the generic combinators is that the desired compatibility properties—shown in Figure 22.4—can be proven once and for all for the generic combinators.[7] In the following, we will discuss the definition of the product combinator $M_1 \times M_2$ in detail and describe the functionality of the others.

[7] Technically, these compatibility properties only hold for modules that do not perform (non-trivial) angelic choices on steps with visible events. This requirement is trivial to satisfy by moving the angelic choice into a separate silent step.

Product. The product combinator

$$M_1 \times M_2 \triangleq (\{\mathsf{E}, \mathsf{L}, \mathsf{R}\} \times S_{M_1} \times S_{M_2}, \rightarrow_\times, (\mathsf{E}, \sigma^0_{M_1}, \sigma^0_{M_2}))$$

builds the product of two modules M_1 and M_2. It is inspired by parallel composition in process calculi such as CSP[8] and CCS,[9] but restricted to a particular form of scheduling (depicted in Figure 22.5): At any point in time, either the environment, the left module M_1, or the right module M_2 is executing. We store whose turn it currently is in a flag $d \in \mathsf{D} \triangleq \{\mathsf{E}, \mathsf{L}, \mathsf{R}\}$ as part of the state of the module (alongside the state of the two modules M_1 and M_2). If the executing party is one of M_1 or M_2 and executes a silent step (PRODUCT-STEP-L-SILENT and PRODUCT-STEP-R-SILENT), then it remains their turn. We only switch turns once the module emits a visible event (PRODUCT-STEP-L and PRODUCT-STEP-R). Whenever we switch, the next turn d is chosen using demonic non-determinism. If it is currently the environment's turn, then we non-deterministically choose d for the next step (PRODUCT-STEP-ENV).

[8] Hoare, "Communicating Sequential Processes", 1978 [Hoa78]; Roscoe, *Understanding Concurrent Systems*, 2010 [Ros10].

[9] Milner et al., "A Calculus of Mobile Processes, I/II", 1992 [MPW92]; Milner, *Communicating and Mobile Systems: the π-Calculus*, 1999 [Mil99].

$$\textsc{product-step-l}\quad \frac{\sigma_1 \xrightarrow{e} \Sigma}{(\mathsf{L}, \sigma_1, \sigma_2) \xrightarrow{\mathsf{left}(e,d)}_{\times} \{(d, \sigma_1', \sigma_2) \mid \sigma_1' \in \Sigma\}} \qquad \textsc{product-step-l-silent}\quad \frac{\sigma_1 \xrightarrow{\tau} \Sigma}{(\mathsf{L}, \sigma_1, \sigma_2) \xrightarrow{\tau}_{\times} \{(\mathsf{L}, \sigma_1', \sigma_2) \mid \sigma_1' \in \Sigma\}}$$

$$\textsc{product-step-r}\quad \frac{\sigma_2 \xrightarrow{e} \Sigma}{(\mathsf{R}, \sigma_1, \sigma_2) \xrightarrow{\mathsf{right}(e,d)}_{\times} \{(d, \sigma_1, \sigma_2') \mid \sigma_2' \in \Sigma\}} \qquad \textsc{product-step-r-silent}\quad \frac{\sigma_2 \xrightarrow{\tau} \Sigma}{(\mathsf{R}, \sigma_1, \sigma_2) \xrightarrow{\tau}_{\times} \{(\mathsf{R}, \sigma_1, \sigma_2') \mid \sigma_2' \in \Sigma\}}$$

$$\textsc{product-step-env}\quad (\mathsf{E}, \sigma_1, \sigma_2) \xrightarrow{\mathsf{env}(d)}_{\times} \{(d, \sigma_1, \sigma_2)\}$$

Figure 22.5: Definition of $\rightarrow_{\times}$.

The events of the module $e_{\times} \triangleq \mathsf{left}(e, d) \mid \mathsf{right}(e, d) \mid \mathsf{env}(d)$ expose the scheduling choice of the product combinator (*i.e.*, who is currently executing). We can exploit this information in other combinators to construct more deterministic schedules. For example, the linking combinator $M_1 \oplus_X M_2$ can filter out certain scheduling choices of $M_1 \times M_2$ and thereby enforce structured communication between M_1 and M_2 (as seen in §21.2).

Filter. For a module $M \in \mathsf{Module}(E_1)$, the filter combinator $M \setminus M' \in \mathsf{Module}(E_2)$ transforms the events of the left module. That is, intuitively, one can think of the filter M' as a relation "$e_1 \sim e_2 \subseteq E_1 \times E_2$" that is used to turn events $e_1 \in E_1$ into events $e_2 \in E_2$ and vice versa. In practice, expressing the filter M' in terms of *a relation* is too restrictive, because we sometimes want (1) to carry state in between the event translations, (2) map a single event to multiple events, and (3) use angelic and demonic non-determinism to control how the events are filtered. Thus, in DimSum, we go beyond a relation "$e_1 \sim e_2$" and instead use a *filter module* M'. This is similar to the notion of transducers in automata theory. The idea is that when M emits an event $e_1 \in E_1$, control is passed to the module M', which will then execute and emit one or more events to the environment. To be precise, the filter module M' communicates using the events:

$$e_{\setminus} ::= \mathsf{FromInner}(e_1 : E_1) \mid \mathsf{ToInner}(e_1 : \mathsf{option}(E_1)) \mid \mathsf{ToEnv}(e_2 : E_2) \mid \mathsf{FromEnv}(e_2 : E_2)$$

$\mathsf{FromInner}(e_1)$ means that M' is willing to accept e_1 from M, $\mathsf{ToInner}(e_1)$ means that M' wants to return control to the module M, optionally sending it the event e_1, $\mathsf{ToEnv}(e_2)$ means that M' wants to send e_2 to the environment, and $\mathsf{FromEnv}(e_2)$ means that M' is willing to accept e_2 from the environment. Throughout all of these interactions, the filter M' can maintain some internal state, since it itself is a module (*i.e.*, state transition system).[10] We will see an instance of a filter module below when we discuss the wrapper combinator.

[10] Readers familiar with process algebra can think of the filter combinator $M \setminus M'$ as the process $(M \parallel M') \backslash E_1$ where the module M' accepts the events from M and emits events $e_2 \in E_2$.

Linking. To express semantic linking in a *language-generic way*, the linking combinator $M_1 \oplus_X M_2$ works as follows. The modules M_1 and M_2 emit tagged events $e_{?!} \in E_{?!} \triangleq E \times \{?, !\}$ such as $\mathbf{Jump}?(\mathbf{r}, \mathbf{m})$

or **Jump**$^!(\mathbf{r}, \mathbf{m})$, where the tag $t \in \{?, !\}$ indicates whether the event is incoming (?) or outgoing (!). It is then the job of the linking operator $\oplus_X$ to flip the event $e_{?!}$ or replace the event $e_{?!}$ with a different event $e'_{?!}$ (*e.g.*, for the coroutine linking operator in §23.3 calls become returns). Technically we define $M_1 \oplus_X M_2 \triangleq (M_1 \times M_2) \setminus \mathsf{link}_X$: the non-deterministic scheduling of $M_1 \times M_2$ is filtered by link_X, which discards out all the interleavings that are "nonsensical". For example, if M_1 wants to "jump" to the environment, then link_X filters out the interleavings of $M_1 \times M_2$ where the next turn is L (for M_1) or R (for M_2).

The parameter $X = (S, \rightsquigarrow, s^0)$ determines how the events are linked. It consists of a set of linking-internal states S, an initial state $s^0 \in S$, and a relation $\rightsquigarrow \subseteq (\mathsf{D} \times S \times E) \times ((\mathsf{D} \times S \times E) \cup \{\lightning\})$ describing how events should be translated. Concretely, $(d, s, e) \rightsquigarrow (d', s', e')$ means the untagged event $e \in E$ coming from direction d should go to d' as the event e'. Behind the scenes, the linking operator then adds the right tag $t \in \{?, !\}$ to e', depending on whether e is part of an incoming or outgoing event. It is also possible that the event cannot be linked $(d, s, e) \rightsquigarrow \lightning$, in which case the linking $M_1 \oplus_X M_2$ has undefined behavior. The linking-internal states S are a form of private state that the linking operator can use to remember information across invocations (*e.g.*, a syscall triggered by the left module should return to the left module). We will discuss concrete instantiations of the linking relation $\rightsquigarrow$ in §23.

(Kripke) wrappers. For a module $M \in \mathsf{Module}(E_1)$, the wrapper $\lceil M \rceil_X \triangleq M \setminus \mathsf{wrap}_X \in \mathsf{Module}(E_2)$ translates events between languages with events E_1 and E_2 (*e.g.*, between Rec and **Asm** in the case of $\lceil \cdot \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$). The combinator is a special case of filtering where the filter wrap_X encodes a particular event translation. The parameter $X = (\mathcal{L}, \rightharpoonup, \leftharpoonup)$ contains a separation logic $\mathcal{L}$ (explained below) and a pair of Kripke relations ($\rightharpoonup$ and $\leftharpoonup$) where $e_1 \rightharpoonup e_2$ controls the translation E_1 to E_2 and $e_1 \leftharpoonup e_2$ controls the translation E_2 to E_1. In both directions, we use non-determinism in the filter wrap_X to pick "the right" corresponding events. For $e_2 \in E_2$ arriving from the environment, the filter *angelically* chooses an event e_1 such that $e_1 \leftharpoonup e_2$. For $e_1 \in E_1$ originating from the module M, the filter *demonically* chooses an event e_2 such that $e_1 \rightharpoonup e_2$.

The relations $\rightharpoonup$ and $\leftharpoonup$ are Kripke relations in the sense that they maintain state between events. Instead of explicitly indexing these relations with a "possible world" (as sketched in §21.3), we define them in separation logic. That is, their type is $\rightharpoonup, \leftharpoonup : E_1 \times E_2 \to \mathit{Prop}_{\mathcal{L}}$ where $\mathit{Prop}_{\mathcal{L}}$ denotes the type of propositions in the separation logic $\mathcal{L}$ (one component of X). The separation logic $\mathcal{L}$ determines which "resources" the relations $\rightharpoonup$ and $\leftharpoonup$ can refer to (*e.g.*, a Rec or **Asm** heap). We always use separation logics $\mathcal{L}$ that are instances of the Iris separation logic framework.[11] We will see a concrete choice of $\mathcal{L}$ and the relations $\rightharpoonup$ and $\leftharpoonup$ in §24.

To understand how the wrapper works, it is instructive to discuss the definition of the filter module wrap_X. It is given by $\mathsf{wrap}_X \triangleq [\![\mathsf{wrap}(\mathsf{True})]\!]_{\mathtt{s}}$ where wrap coordinates the exchange between the wrapped module M and its environment. The argument of wrap is a separation logic proposition

[11] In fact, readers familiar with Iris can think of the separation logic $\mathcal{L}$ as $\mathit{UPred}(R)$, the separation logic of uniform predicates over the resource algebra R, where each instance of the wrapper combinator chooses its own resource algebra R.

keeping track of the "resources" that the inner module owns privately (*i.e.*, that are not shared with the environment). The definition of wrap is given by:

$$\begin{aligned}&\mathsf{wrap}(P_1) \triangleq_{\mathsf{coind}} \\ &\quad \exists e_2; \mathsf{vis}(\mathsf{FromEnv}(e_2)); \forall e_1, P_2; \mathsf{assume}(\mathsf{sat}(P_1 * P_2 * e_1 \mathrel{-\!\!\prec} e_2)); \mathsf{vis}(\mathsf{ToInner}(e_1)); \\ &\quad \exists e_1'; \mathsf{vis}(\mathsf{FromInner}(e_1')); \exists e_2', P_1'; \mathsf{assert}(\mathsf{sat}(P_1' * P_2 * e_1' \mathrel{-\!\!\succ} e_2')); \mathsf{vis}(\mathsf{ToEnv}(e_2')); \mathsf{wrap}(P_1')\end{aligned}$$

Initially, the filter accepts any incoming event e_2 from the environment (with $\mathsf{FromEnv}(e_2)$). It then assumes it is given *angelically* the corresponding event e_1 for the inner module, which it sends to the module (with $\mathsf{ToInner}(e_1)$). Afterwards, the filter module accepts any response event e_1' from the inner module (with $\mathsf{FromInner}(e_1')$). It then *demonically* chooses the corresponding event e_2' to send to the environment (with $\mathsf{ToEnv}(e_2')$).

In the exchange between the wrapped module M and the environment, separation logic propositions are used to "divide up" the shared state (*e.g.*, the locations in the heap). The wrapped module exclusively owns some resources P_1 which the environment may not change, and it can update them to P_1' during the exchange. The environment exclusively owns some resources P_2 which the wrapped module must preserve while updating its own state.[12] Finally, during every exchange some parts of the state can be shared between the environment and the wrapped module using the separation logic relations $e_1 \mathrel{-\!\!\succ} e_2$ and $e_1 \mathrel{-\!\!\prec} e_2$ (see §24). The proposition $\mathsf{sat}(P)$ (read P is satisfiable) here means that there is some valid underlying resource (*e.g.*, a heap) for which P holds.

[12] The structure of this exchange follows Iris's frame-preserving update modality (see *e.g.*, Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b]): Initially, the module assumes some decomposition of the shared resources, and then it makes sure to only update the exclusively owned resources P_1 to resources P_1' that remain compatible with the environment resources.

Chapter 23

Instantiations of DimSum

23.1 The Language **Asm**

$$\begin{aligned}
\textbf{Library} \ni \mathbf{A} &\triangleq \mathbb{Z} \xrightarrow{\text{fin}} \textbf{Instr} \\
\textbf{Instr} \ni \mathbf{c} &::= \mathsf{ret} \mid \mathsf{mov}\ \mathbf{x},\ \mathbf{o} \mid \mathsf{add}\ \mathbf{x_1},\ \mathbf{x_2},\ \mathbf{o} \mid \mathsf{mul}\ \mathbf{x_1},\ \mathbf{x_2},\ \mathbf{o} \mid \mathsf{sle}\ \mathbf{x_1},\ \mathbf{x_2},\ \mathbf{o} \mid \mathsf{syscall} \\
&\quad \mid \mathsf{ldr}\ \mathbf{x_1}, [\mathbf{x_2} + i] \mid \mathsf{str}\ \mathbf{x_1}, [\mathbf{x_2} + i] \mid \mathsf{jmp}\ \mathbf{o} \mid \mathsf{beq}\ \mathbf{o_1},\ \mathbf{x},\ \mathbf{o_2} \mid \cdots \\
\textbf{Operand} \ni \mathbf{o} &::= \mathbf{x} : \textbf{RegisterName} \mid i : \mathbb{Z} \\
\textbf{Execution State} \ni \mathbf{I} &::= \textbf{Wait} \mid \textbf{Run}(\mathbf{r}, \mathbf{m}) \mid \textbf{WaitSyscall}(\mathbf{r}) \mid \textbf{Halted}
\end{aligned}$$

ASM-INCOMING

$$\frac{\mathbf{r}(\mathbf{pc}) = \mathbf{a} \qquad \mathbf{a} \in |\mathbf{A}|}{(\textbf{Wait}, \mathbf{A}) \xrightarrow{\textbf{Jump?}(\mathbf{r},\mathbf{m})}_{\mathsf{a}} \{(\textbf{Run}(\mathbf{r}, \mathbf{m}), \mathbf{A})\}}$$

ASM-JUMP-INTERNAL

$$\frac{\mathbf{r}(\mathbf{pc}) = \mathbf{a} \qquad \mathbf{A}(\mathbf{a}) = \mathsf{jmp}\ \mathbf{v} \qquad \mathbf{v} \in |\mathbf{A}|}{(\textbf{Run}(\mathbf{r}, \mathbf{m}), \mathbf{A}) \xrightarrow{\tau}_{\mathsf{a}} \{(\textbf{Run}(\mathbf{r}[\mathbf{pc} \mapsto \mathbf{v}], \mathbf{m}), \mathbf{A})\}}$$

ASM-JUMP-EXTERNAL

$$\frac{\mathbf{r}(\mathbf{pc}) = \mathbf{a} \qquad \mathbf{A}(\mathbf{a}) = \mathsf{jmp}\ \mathbf{v} \qquad \mathbf{v} \notin |\mathbf{A}|}{(\textbf{Run}(\mathbf{r}, \mathbf{m}), \mathbf{A}) \xrightarrow{\textbf{Jump!}(\mathbf{r}[\mathbf{pc} \mapsto \mathbf{v}], \mathbf{m})}_{\mathsf{a}} \{(\textbf{Wait}, \mathbf{A})\}}$$

Figure 23.1: Grammar and excerpt of the operational semantics of **Asm**.

The language **Asm** is an idealized assembly language with instructions for arithmetic, jumps, memory accesses, and syscalls (depicted in Figure 23.1).[1] The libraries **A** of **Asm** are finite maps from addresses to instructions. The set of their instruction addresses is defined as $|\mathbf{A}| = \mathsf{dom}\ \mathbf{A}$.

Module semantics. The semantics of an **Asm** library **A** is the module $[\![\mathbf{A}]\!]_{\mathsf{a}}$. We write $(\rightarrow_{\mathsf{a}})$ for the transition system (excerpt shown in Figure 23.1). The states of the module are of the form $\sigma = (\mathbf{I}, \mathbf{A})$ where **I** is the current *execution state*, and the initial state is $(\textbf{Wait}, \mathbf{A})$. Conceptually, four different execution states are possible during the execution of **A**: executing $(\textbf{Run}(\mathbf{r}, \mathbf{m}))$, waiting for incoming jumps (**Wait**), waiting for a syscall to return ($\textbf{WaitSyscall}(\mathbf{r})$, where **r** preserves the registers across the syscall), or finished executing (**Halted**). To explain the transitions, we discuss three cases. Initially, the module is waiting (**Wait**) and can accept any incoming jump with arbitrary memory and registers (see ASM-INCOMING). After accepting the jump ($\textbf{Run}(\mathbf{r}, \mathbf{m})$), the module executes the instructions of **A**, updating the current register assignment **r** and memory **m** (not shown in the figure). When the module reaches

[1] Following Islaris, the Coq development defines the instructions depicted in Figure 23.1 as compositions of micro-instructions.

$$\textsc{asm-link-jump}\quad \frac{(d' = \mathsf{L} \wedge \mathbf{r(pc)} \in \mathbf{d_1}) \vee (d' = \mathsf{R} \wedge \mathbf{r(pc)} \in \mathbf{d_2}) \vee (d' = \mathsf{E} \wedge \mathbf{r(pc)} \notin \mathbf{d_1} \cup \mathbf{d_2}) \qquad d \neq d'}{(d, \mathsf{None}, \mathbf{Jump(r, m)}) \rightsquigarrow_{\mathbf{d_1},\mathbf{d_2}} (d', \mathsf{None}, \mathbf{Jump(r, m)})}$$

$$\textsc{rec-link-call}\quad \frac{(d' = \mathsf{L} \wedge \mathsf{f} \in \mathsf{d}_1) \vee (d' = \mathsf{R} \wedge \mathsf{f} \in \mathsf{d}_2) \vee (d' = \mathsf{E} \wedge \mathsf{f} \notin \mathsf{d}_1 \cup \mathsf{d}_2) \qquad d \neq d'}{(d, \overline{d_s}, \mathsf{Call}(\mathsf{f}, \overline{v}, \mathsf{m})) \rightsquigarrow_{\mathsf{d}_1,\mathsf{d}_2} (d', d :: \overline{d_s}, \mathsf{Call}(\mathsf{f}, \overline{v}, \mathsf{m}))}$$

$$\textsc{rec-link-ret}\quad \frac{d \neq d'}{(d, d' :: \overline{d_s}, \mathsf{Return}(\mathsf{v}, \mathsf{m})) \rightsquigarrow_{\mathsf{d}_1,\mathsf{d}_2} (d', \overline{d_s}, \mathsf{Return}(\mathsf{v}, \mathsf{m}))}$$

$$\textsc{coro-link-yield}\quad \frac{(d = \mathsf{L} \wedge d' = \mathsf{R}) \vee (d = \mathsf{R} \wedge d' = \mathsf{L})}{(d, (d, \mathsf{None}), \mathsf{Call}(\mathsf{yield}, [\mathsf{v}], \mathsf{m})) \rightsquigarrow^{\mathsf{d}_1,\mathsf{d}_2}_{\mathsf{coro}} (d', (d', \mathsf{None}), \mathsf{Return}(\mathsf{v}, \mathsf{m}))}$$

Figure 23.2: Excerpt of the semantic linking relations of **Asm** ($\rightsquigarrow$), Rec ($\rightsquigarrow$), and Rec with coroutines ($\rightsquigarrow_{\mathsf{coro}}$).

a jump jmp **v** (or jumps to an instruction in another way), one of two things happens: Either the destination **v** is in the address range of **A** and we continue executing in the module (see ASM-JUMP-INTERNAL), or the destination **v** is outside the address range of **A**. In the latter case, the module emits **Jump!**$(\mathbf{r}[\mathbf{pc} \mapsto \mathbf{v}], \mathbf{m})$ and returns to the **Wait** state. Syscalls execute analogously to jumps (with **WaitSyscall**), and a library can finish executing (**Halted**).

Linking. Syntactically, linking of two **Asm** libraries (*i.e.*, $\mathbf{A_1} \cup_{\mathsf{a}} \mathbf{A_2}$) means merging the maps $\mathbf{A_1}$ and $\mathbf{A_2}$. In case of overlapping addresses, the conflict is resolved by using the instruction from $\mathbf{A_1}$ (the choice of $\mathbf{A_1}$ over $\mathbf{A_2}$ is arbitrary). Semantically, linking is more interesting. If we link two **Asm** modules (*i.e.*, $\mathbf{M_1} \,{}^{\mathbf{d_1}}\!\oplus^{\mathbf{d_2}}_{\mathsf{a}} \mathbf{M_2}$), then we have to synchronize based on the jump events. To define $\mathbf{M_1} \,{}^{\mathbf{d_1}}\!\oplus^{\mathbf{d_2}}_{\mathsf{a}} \mathbf{M_2}$, we use the combinator $M_1 \oplus_X M_2$ from §22.3. In the case of **Asm**, we pick $X = (\mathsf{option}(\mathsf{D}), \rightsquigarrow_{\mathbf{d_1},\mathbf{d_2}}, \mathsf{None})$. For a jump event (see ASM-LINK-JUMP in Figure 23.2), the linking operator resolves the destination d' based on the addresses in $\mathbf{d_1}$ and $\mathbf{d_2}$—jumps that are outside these addresses are passed on to the environment. Syscalls are propagated to the environment.

23.2 The Language Rec

The language Rec is a simple, high-level language with arithmetic operations, let bindings, memory operations, conditionals, and (potentially recursive) function calls (depicted in Figure 23.3). The libraries R of Rec are lists of function declarations. Each function declaration contains the name of the function f, the argument names $\overline{x}$, local variables $\overline{y}$ which are allocated in the memory, and a function body e. The set of function names $|\mathsf{R}|$ of a library R is defined as the names of the functions in the list R. The syntactic linking $\mathsf{R}_1 \cup_{\mathsf{r}} \mathsf{R}_2$ merges the function declarations of both libraries (again giving precedence to the left side in case of conflict). Similar to **Asm**, the semantic linking $\mathsf{M}_1 \,{}^{\mathsf{d}_1}\!\oplus^{\mathsf{d}_2}_{\mathsf{r}} \mathsf{M}_2$ is an instance of $(\oplus_X)$ where the linking relation is depicted in Figure 23.2. The most interesting

$$\begin{aligned}
\mathsf{Library} \ni R &::= (\mathsf{fn}\ f(\overline{x}) \triangleq \overline{\mathsf{local}\ y[n]; e}), R \mid \emptyset \\
\mathsf{Expr} \ni e &::= v \mid x \mid e_1 \oplus e_2 \mid \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid e_1(\overline{e_2}) \mid \mathbf{!}e \mid e_1 \leftarrow e_2 \\
\mathsf{BinOp} \ni \oplus &::= + \mid < \mid == \mid \leq
\end{aligned}$$

Figure 23.3: Grammar of Rec.

difference to **Asm** is that linking in Rec has to build up and then wind down a call-stack (through calls and returns), which is maintained as the internal state of $(\rightsquigarrow)$.

23.3 Coroutine Linking $\mathsf{M}_1 \oplus_{coro} \mathsf{M}_2$

One of the strong suits of DimSum is that it allows multiple semantic linking operators for the same language. We showcase this using the coroutine linking operator $\mathsf{M}_1 \oplus_{\mathsf{coro}} \mathsf{M}_2$ from the example in §20.1. Similar to $(\oplus_{\mathsf{a}})$ and $(\oplus_{\mathsf{r}})$, the operator $\mathsf{M}_1 \oplus_{\mathsf{coro}} \mathsf{M}_2$ is an instance of $M_1 \oplus_X M_2$. The most interesting case of its transition relation $\rightsquigarrow_{\mathsf{coro}}$, CORO-LINK-YIELD, is depicted in Figure 23.2. Here, we can see that the linking operator links calls to yield from M_1 (resp. M_2) with returns from yield in M_2 (resp. M_1). This translation of calls to returns captures the intuitive behavior of the coroutine library **yield** (in Figure 20.1) at the level of Rec—without mentioning the complex implementation of **yield** in **Asm**.

Verification of main *and* stream. We verify the example in Figure 20.1, namely that it prints 0, then 1, and then returns 2. The proof strategy for this verification is analogous to §21.2:

$$\begin{aligned}
\mathbf{coro} &\preceq [\![\mathbf{yield}]\!]_{\mathsf{a}} \oplus_{\mathsf{a}} \lceil [\![\mathsf{main}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \oplus_{\mathsf{a}} \lceil [\![\mathsf{stream}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \oplus_{\mathsf{a}} [\![\mathbf{print}_{\mathsf{spec}}]\!]_{\mathsf{s}} \\
&\preceq \lceil [\![\mathsf{main}]\!]_{\mathsf{r}} \oplus_{\mathsf{coro}} [\![\mathsf{stream}]\!]_{\mathsf{r}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \oplus_{\mathsf{a}} [\![\mathbf{print}_{\mathsf{spec}}]\!]_{\mathsf{s}} \\
&\preceq \lceil [\![\mathsf{main}_{\mathsf{spec}}]\!]_{\mathsf{s}} \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \oplus_{\mathsf{a}} [\![\mathbf{print}_{\mathsf{spec}}]\!]_{\mathsf{s}} \preceq [\![\mathbf{coro}_{\mathsf{spec}}]\!]_{\mathsf{s}}
\end{aligned}$$

Here **coro** denotes the compiled and syntactically linked **Asm**-program. The specifications $\mathbf{coro}_{\mathsf{spec}}$ and $\mathsf{main}_{\mathsf{spec}}$ are similar to the corresponding specifications in §21. The key steps of this proof are the second and third step which use the following rules:

CORO-LINK

$$[\![\mathbf{yield}]\!]_{\mathsf{a}} \oplus_{\mathsf{a}} \lceil \mathsf{M}_1 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \oplus_{\mathsf{a}} \lceil \mathsf{M}_2 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}} \preceq \lceil \mathsf{M}_1 \oplus_{\mathsf{coro}} \mathsf{M}_2 \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$$

MAIN-CORO

$$[\![\mathsf{main}]\!]_{\mathsf{r}} \oplus_{\mathsf{coro}} [\![\mathsf{stream}]\!]_{\mathsf{r}} \preceq [\![\mathsf{main}_{\mathsf{spec}}]\!]_{\mathsf{s}}$$

The lemma CORO-LINK is a generic lemma provided by the coroutine library that allows abstracting the **yield** library to the $\mathsf{M}_1 \oplus_{\mathsf{coro}} \mathsf{M}_2$. This lemma enables us to verify the composition of main and stream purely at the Rec-level (MAIN-CORO)—completely independently of **Asm**.

Chapter 24

Compiler

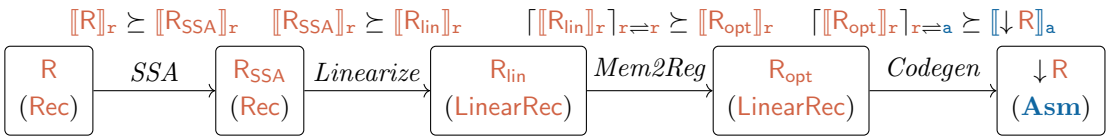


Figure 24.1: Structure of our Rec to **Asm** compiler.

This chapter describes our compiler $\downarrow \mathsf{R}$ from Rec to **Asm**, and how we verify it in DimSum. The compiler has four passes, depicted in Figure 24.1: The first pass, *SSA*, renames variables such that each variable is only assigned once, and the second pass, *Linearize*, converts the program into A-normal form. The A-normal form is expressed in an intermediate representation, LinearRec, which is a subset of Rec that only allows let-bindings and if-statements at the top-level and flattens all nested expressions. The third pass, *Mem2Reg*, is a non-trivial optimization pass that reduces memory consumption by turning local variables whose address is never observed into let-bindings (which can be compiled to registers subsequently). For example, it turns (the A-normal form of) $\mathsf{fn}\ \mathsf{f}(x) \triangleq \mathsf{local}\ y[1]; y \leftarrow x; !y + !y$ into $\mathsf{fn}\ \mathsf{f}(x) \triangleq \mathsf{let}\ y := 0\ \mathsf{in}\ \mathsf{let}\ y := x\ \mathsf{in}\ y + y$, because the *address* of y is never used. The final pass, *Codegen*, is a standard code-generation pass producing the **Asm** code: it takes care of register allocation (including spilling to the stack when necessary), allocating local variables on the stack, and adhering to the **Asm** calling convention.

Compiler correctness. Let us turn to the correctness of the compiler (COMPILER-CORRECT in §21.2):[1]

Theorem 6 (Compiler Correctness) *If* $\downarrow \mathsf{R}$ *is defined, then*

$$\llbracket \downarrow \mathsf{R} \rrbracket_{\mathtt{a}} \preceq \lceil \llbracket \mathsf{R} \rrbracket_{\mathtt{r}} \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$$

Intuitively, compiler correctness says that the compiled assembly code behaves like the original source program translated by the wrapper—*i.e.*, syntactic translation via the compiler refines semantic translation via the wrapper $\lceil \cdot \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$. As we have seen in §21.2, COMPILER-CORRECT is a useful result that allows one to replace reasoning about the compiled assembly code with reasoning about the source program. The $\lceil \cdot \rceil_{\mathtt{r} \rightleftharpoons \mathtt{a}}$ wrapper is defined using the $\lceil \cdot \rceil_X$ combinator from §22.3.

The compiler correctness result is proven by composing refinements for the individual passes (the refinements are shown in Figure 24.1 above each corresponding pass):

[1] To simplify the presentation, the rule COMPILER-CORRECT omits some technical details relevant for the translation between Rec-function names and the corresponding **Asm**-instruction addresses.

$$\lceil[\![R]\!]_r\rceil_{r\rightleftharpoons a} \succeq \lceil[\![R_{SSA}]\!]_r\rceil_{r\rightleftharpoons a} \succeq \lceil[\![R_{lin}]\!]_r\rceil_{r\rightleftharpoons a} \succeq \lceil\lceil[\![R_{lin}]\!]_r\rceil_{r\rightleftharpoons r}\rceil_{r\rightleftharpoons a} \succeq \lceil[\![R_{opt}]\!]_r\rceil_{r\rightleftharpoons a} \succeq [\![\downarrow R]\!]_a$$

The refinements for the *SSA* and *Linearize* passes are straightforward Rec refinements, and the *Codegen* pass uses the $\lceil\cdot\rceil_{r\rightleftharpoons a}$ wrapper to translate between Rec and **Asm**. The pass *Mem2Reg* is special, however, in that it introduces an additional wrapper $\lceil\cdot\rceil_{r\rightleftharpoons r}$. To understand what this wrapper does and why we have to introduce it, let us first take a step back and consider how DimSum determines which program transformations are considered semantics-preserving.

Semantics-preserving program transformations. There are two classes of program transformations that are semantics-preserving in DimSum.

The first class of semantics-preserving program transformations *does not* change observable parts of the events. This can be expressed by proving that the transformed program refines the original, *i.e.*, $[\![R_1]\!]_r \preceq [\![R_2]\!]_r$ where R_1 is the transformed program and R_2 the original. This refinement expresses that the transformed program R_1 must emit the same events as the original R_2, because the definition of $(\preceq)$ (in §22.1) enforces that the events of R_1 and R_2 match exactly. Thus, by selecting which information about the program state to include in its events, a language effectively determines this first class of semantics-preserving program transformations. For example, Rec includes values v and heaps m in its events and thus program transformations can change the syntactic structure of a program—since the program structure is not part of the events—but they cannot alter return values or the heap that is shared across function calls (unless they fall into the second class). In the compiler, the *SSA* and *Linearize* passes fall into this first class of transformations (see their correctness statements in Figure 24.1).

The second class of semantics-preserving program transformations *does* change observable parts of the events. For example, the *Mem2Reg* transformation falls into this category, because it alters memory in such a way that it becomes impossible to align the events of the transformed program R_{opt} with the original R_{lin}. To verify these transformations in DimSum, one can use additional wrappers to make sure the events do match up. For example, in the case of *Mem2Reg*, we *cannot prove* $[\![R_{opt}]\!]_r \preceq [\![R_{lin}]\!]_r$ because of the event mismatch, but we *can prove* $[\![R_{opt}]\!]_r \preceq \lceil[\![R_{lin}]\!]_r\rceil_{r\rightleftharpoons r}$ where the wrapper $\lceil\cdot\rceil_{r\rightleftharpoons r}$ transforms the events of $[\![R_{lin}]\!]_r$ such that they match up with $[\![R_{opt}]\!]_r$. Concretely, the $\lceil\cdot\rceil_{r\rightleftharpoons r}$ wrapper ensures that *private* memory locations (*i.e.*, local variables that have never been shared with the environment via function arguments or return values) are not part of the events, and thus *Mem2Reg* is allowed to optimize them away.

While wrappers such as $\lceil\cdot\rceil_{r\rightleftharpoons r}$ enable the verification of more program transformations, we also have to make sure that they do not allow *too many* transformations (*i.e.*, incorrect transformations). This constraint is handled implicitly by the compiler correctness statement compiler-correct—it does not mention any wrappers other than $\lceil\cdot\rceil_{r\rightleftharpoons a}$, and in order to use a new wrapper such as $\lceil\cdot\rceil_{r\rightleftharpoons r}$, we have to show that the additional transformations it enables are also allowed by the wrapper $\lceil\cdot\rceil_{r\rightleftharpoons a}$. We call this property "*vertical compositionality*" of the

two wrappers (rec-to-asm-vertical below) and it allows us to prove $\lceil\lceil[\![\mathsf{R}_{\mathsf{lin}}]\!]_{\mathtt{r}}\rceil_{\mathtt{r}\rightleftharpoons\mathtt{r}}\rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}} \preceq \lceil[\![\mathsf{R}_{\mathsf{lin}}]\!]_{\mathtt{r}}\rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}}$ in the refinement chain of compiler-correct.

Vertical compositionality. Vertical compositionality in DimSum means not only proving transitivity of the simulation relation ($\preceq$), but also that certain "intermediate wrappers" can be eliminated. For instance, the vertical compositionality result of $\lceil\cdot\rceil_{\mathtt{r}\rightleftharpoons\mathtt{r}}$ in this compiler correctness proof is given by:

$$\text{rec-to-asm-vertical}\ \lceil\lceil\mathsf{M}\rceil_{\mathtt{r}\rightleftharpoons\mathtt{r}}\rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}} \preceq \lceil\mathsf{M}\rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}}$$

This theorem, like most vertical compositionality theorems, is difficult to prove, since we need to show that certain Rec-level memory transformations do not change the behavior of M from the perspective of **Asm** in any meaningful way.

Note that in other approaches to multi-language semantics, vertical compositionality typically either requires composing simulation relations[2] or proving the transitivity of contextual refinement.[3] In contrast, in DimSum, the (Kripke) wrappers $\lceil M\rceil_X$ effectively assume the role of "simulation conventions" or "simulation invariants" in a (Kripke) simulation relation (*e.g.*, relations on memories and values), but they do so as *a module combinator*. As a result, proving vertical compositionality in DimSum is not necessarily simpler, but it is more localized: if we want vertical compositionality of two transformations (*i.e.*, two wrappers), then we prove a single simulation (*e.g.*, rec-to-asm-vertical), and no other parts of the framework are affected, since they are compatible with simulation.

[2] Neis et al., "Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language", 2015 [Nei+15]; Stewart et al., "Compositional CompCert", 2015 [Ste+15].

[3] Perconti and Ahmed, "Verifying an Open Compiler Using Multi-language Semantics", 2014 [PA14].

The $\lceil\cdot\rceil_{\mathtt{r}\rightleftharpoons\mathtt{r}}$ wrapper. Let us now turn to the definition of the wrapper $\lceil\cdot\rceil_{\mathtt{r}\rightleftharpoons\mathtt{r}}$, which is an instance of the generic combinator $\lceil M\rceil_X$ (from §22.3). As explained above, the purpose of $\lceil\cdot\rceil_{\mathtt{r}\rightleftharpoons\mathtt{r}}$ is to enable optimizing memory locations that are kept private and never shared with the environment (*e.g.*, via function arguments or return values). To this end, we instantiate $\lceil M\rceil_X$ with a separation logic $\mathcal{L}$ inspired by Gäher et al.[4] that supports assertions for both persistent ownership of shared locations ($\ell_1 \leftrightarrow \ell_2$) and exclusive ownership of private memory locations ($\ell_1 \mapsto_{\mathsf{E}} \mathsf{v}$ and $\ell_2 \mapsto_{\mathsf{I}} \mathsf{v}$). The assertion $\ell_1 \leftrightarrow \ell_2$ states that location ℓ_1 in the *external memory* (*i.e.*, the memory of $\lceil\mathsf{M}\rceil_{\mathtt{r}\rightleftharpoons\mathtt{r}}$) always corresponds to ℓ_2 in the *internal memory* (*i.e.*, the memory of M). The assertion $\ell \mapsto_{\mathsf{E}} \mathsf{v}$ conveys exclusive ownership of the locations ℓ in the external memory, and $\ell \mapsto_{\mathsf{I}} \mathsf{v}$ of ℓ in the internal memory. Additionally, the separation logic provides the assertion $\mathsf{inv}(\mathsf{m}_1, \mathsf{m}_2)$ which connects "$\ell_1 \leftrightarrow \ell_2$", "$\ell \mapsto_{\mathsf{E}} \mathsf{v}$", and "$\ell \mapsto_{\mathsf{I}} \mathsf{v}$" to the heaps m_1 and m_2 such that the heaps overlap in the way described by the assertions. We then instantiate the relations $\mathsf{e}_{\mathsf{I}} \rightharpoonup \mathsf{e}_{\mathsf{E}}$ and $\mathsf{e}_{\mathsf{I}} \leftharpoonup \mathsf{e}_{\mathsf{E}}$ with:

[4] Gäher et al., "Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations", 2022 [Gäh+22].

$$\mathsf{e}_{\mathsf{I}} \rightharpoonup \mathsf{e}_{\mathsf{E}}, \mathsf{e}_{\mathsf{I}} \leftharpoonup \mathsf{e}_{\mathsf{E}} \triangleq \mathsf{type}(\mathsf{e}_{\mathsf{I}}) = \mathsf{type}(\mathsf{e}_{\mathsf{E}}) * \mathsf{inv}(\mathsf{mem}(\mathsf{e}_{\mathsf{I}}), \mathsf{mem}(\mathsf{e}_{\mathsf{E}})) * \forall(\mathsf{v}_1, \mathsf{v}_2) \in \mathsf{vals}(\mathsf{e}_{\mathsf{I}}, \mathsf{e}_{\mathsf{E}}).\ \mathsf{v}_1 \leftrightarrow \mathsf{v}_2$$

This definition consists of three parts: First, the type of the events (*i.e.*, call or return) has to match. Second, the invariant inv has to hold for the memories contained in the events. Third, the values of events (*e.g.*,

function arguments and return values) must be related (*i.e.*, $\ell_1 \leftrightarrow \ell_2$ for locations and equality for integers and Booleans).

Let us now consider how the wrapper $\lceil\cdot\rceil_{\mathtt{r}\rightleftharpoons\mathtt{r}}$ enables the verification of the *Mem2Reg* pass, which transforms $\mathsf{R}_{\mathsf{lin}}$ to $\mathsf{R}_{\mathsf{opt}}$. Recall that this transformation replaces a local variable allocated at some location ℓ in $\mathsf{R}_{\mathsf{lin}}$ with a let-binding without corresponding allocation in $\mathsf{R}_{\mathsf{opt}}$. To justify this transformation, we need to prove that the value stored at ℓ in $\mathsf{R}_{\mathsf{lin}}$ always corresponds to the let-bound value in $\mathsf{R}_{\mathsf{opt}}$ and, in particular, remains constant across external function calls. To this end, we use the $\ell \mapsto_{\mathsf{I}} \mathsf{v}$ assertion, which we obtain at the start of the function when ℓ is allocated on the heap, and we keep it in the privately owned part P_1 of the combinator $\lceil M \rceil_X$ (see §22.3). That is, $\ell \mapsto_{\mathsf{I}} \mathsf{v}$ allows us to track the precise value of ℓ and ensure that it corresponds to the let-bound value in the optimized program. When calling the environment, we do not need to give up $\ell \mapsto_{\mathsf{I}} \mathsf{v}$, because the location is never exposed and thus never appears in $\mathsf{e}_{\mathsf{I}} \rightharpoonup \mathsf{e}_{\mathsf{E}}$ (otherwise the optimization does not fire). Instead, we can keep $\ell \mapsto_{\mathsf{I}} \mathsf{v}$ in the privately owned part of the module P_1 and the definition of wrap ensures that $\ell \mapsto_{\mathsf{I}} \mathsf{v}$ holds across the function call. Thus, we know ℓ still points to v after the function call, which allows us to complete the verification of the *Mem2Reg* pass.

Chapter 25

Related Work

We first compare with other work on semantics and verification of programs with components written in multiple languages. Then we discuss other lines of work that inspired the design of DimSum.

CompCert-based approaches to multi-language verification. Although CompCert's original correctness statement[1] only concerns whole programs, it inspired a long line of work on multi-language verification.[2]

The approach most closely related to DimSum is CompCertO,[3] since its game semantics-based approach has parallels to the event-based approach of DimSum—*e.g.*, CompCertO's language interfaces play a similar role to the event types of DimSum. To scale to the full extent of CompCert, CompCertO makes design choices specific to the languages of CompCert. In particular, it provides only a single linking operator that enforces a well-bracketed call structure (unlike $\mathsf{M}_1 \oplus_{\mathtt{coro}} \mathsf{M}_2$), studies only transition systems without private state across function invocations (unlike $\lceil\cdot\rceil_{\mathtt{r}\rightleftharpoons\mathtt{a}}$), and—even though CompCertO's underlying definition of simulation convention is independent of the memory model—considers only languages with the same memory model (unlike Rec and **Asm**).

Compositional CompCert,[4] Ramananandro et al.,[5] and CompCertM[6] achieve multi-language linking by imposing a common interaction protocol between all languages. This works well in their setting since all CompCert languages share a notion of values and memory, but it is unclear how to scale this setup to more heterogeneous languages like Rec and **Asm**. Similar to DimSum, Ramananandro et al. use events that contain the complete program memory. They define linking on traces of call events (*i.e.*, "behaviors") and prove equivalence between syntactic and semantic linking similar to ASM-LINK-SYN and REC-LINK-SYN. However, their traces erase the branching structure of the program. In DimSum, sensitivity to branching is crucial due to the presence of both demonic and angelic non-determinism (see §22.2), so we define linking directly on transition systems (*i.e.*, "modules").

CompCertX and (Concurrent) Certified Abstraction Layers[7] have been successfully deployed in the verification of the CertiKOS verified operating system. However, they impose restrictions on the interaction between different components, such as forbidding mutual recursion and certain memory sharing patterns. In contrast, our semantic linking operators $\mathbf{M_1} \oplus_{\mathtt{a}} \mathbf{M_2}$ and $\mathsf{M}_1 \oplus_{\mathtt{r}} \mathsf{M}_2$ allow mutual recursion and memory sharing.

[1] Leroy, "Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant", 2006 [Ler06].

[2] Beringer et al., "Verified Compilation for Shared-Memory C", 2014 [Ber+14]; Ramananandro et al., "A Compositional Semantics for Verified Separate Compilation and Linking", 2015 [Ram+15]; Stewart et al., "Compositional CompCert", 2015 [Ste+15]; Kang et al., "Lightweight Verification of Separate Compilation", 2016 [Kan+16]; Gu et al., "Deep Specifications and Certified Abstraction Layers", 2015 [Gu+15]; Wang et al., "An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code", 2019 [WWS19]; Song et al., "CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification", 2020 [Son+20]; Koenig and Shao, "CompCertO: Compiling Certified Open C Components", 2021 [KS21].

[3] Koenig and Shao, "CompCertO: Compiling Certified Open C Components", 2021 [KS21].

[4] Stewart et al., "Compositional CompCert", 2015 [Ste+15].

[5] Ramananandro et al., "A Compositional Semantics for Verified Separate Compilation and Linking", 2015 [Ram+15].

[6] Song et al., "CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification", 2020 [Son+20].

[7] Gu et al., "Deep Specifications and Certified Abstraction Layers", 2015 [Gu+15]; Wang et al., "An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code", 2019 [WWS19]; Gu et al., "Certified Concurrent Abstraction Layers", 2018 [Gu+18]; Vale et al., "Layered and Object-Based Game Semantics", 2022 [Val+22].

Syntactic multi-languages. Syntactic multi-languages[8] embed source, target, and intermediate languages into a common multi-language (with "boundary" terms to translate between languages) and then use the multi-language to, among other things, state and verify compiler correctness theorems. As this line of work demonstrates, syntactic multi-languages scale well to typed, higher-order languages. In DimSum, we have so far put the focus on different kinds of languages: untyped, low-level languages comparable to C and assembly.

Specifications in syntactic multi-languages use contextual equivalence, which is canonical but has the downside of including the operations (and semantics) of all involved languages in every specification. In contrast, when we prove $[\![\mathsf{main} \cup_{\mathtt{r}} \mathsf{memmove}]\!]_{\mathtt{r}} \oplus_{\mathtt{r}} [\![\mathsf{locle}_{\mathtt{spec}}]\!]_{\mathtt{s}} \preceq [\![\mathsf{main}_{\mathtt{spec}}]\!]_{\mathtt{s}}$ in §21, we only care about the semantics of Rec, not that the modules are embedded in an **Asm** context.

Mates et al.[9] prove correctness of a compiler from a source language without call/cc to a target language with call/cc and show that compiled code can be linked with a thread library loosely similar to our coroutine library (§23.3). They do not provide a high-level abstraction like $\mathsf{M}_1 \oplus_{\mathsf{coro}} \mathsf{M}_2$ and instead require clients to reason about the implementation of the library. In their case, the distinction does not matter much because the target language is reasonably high-level, but for Rec and **Asm**, it would involve reasoning about low-level stack and register manipulation.

Recently, Patterson et al.[10] proved type safety for several, very different multi-languages by giving a realizability model in an untyped target language. While effective for type safety, the downside of this approach is that most reasoning happens at the target language. In contrast, an important goal of DimSum is to lift reasoning to source-level languages as shown by §21 and §23.3.

Pilsner. Building on the work of Hur and collaborators,[11] Pilsner[12] verifies two compilers from a higher-order stateful source language to an assembly target language and shows that the compiled programs can be safely linked. However, Pilsner prohibits linking with target-level libraries whose observable functionality is inexpressible in the source; as such, it rules out the examples in §21 and §23.3.

Other approaches. The Cito compiler[13] simplifies its compositional compiler correctness statement by requiring the user to provide specifications for all external functions. In contrast, while DimSum supports giving specifications for low-level libraries as seen in §21, they are not required by our compiler correctness theorem.

Conditional Contextual Refinement (CCR)[14] uses dual (demonic and angelic) non-determinism to encode a wrapper that can transform the values of function arguments and results (as first shown by Back[15]) and enforce separation logic preconditions and postconditions. DimSum's wrappers are inspired by this idea but apply it to interoperation between different languages and memory models instead of program verification. While CCR allows linking between different languages (*e.g.*, between an

[8] Matthews and Findler, "Operational Semantics for Multi-Language Programs", 2007 [MF07]; Perconti and Ahmed, "Verifying an Open Compiler Using Multi-language Semantics", 2014 [PA14]; Patterson et al., "FunTAL: Reasonably Mixing a Functional Language with Assembly", 2017 [Pat+17]; Mates et al., "Under Control: Compositionally Correct Closure Conversion with Mutable State", 2019 [MPA19]; Patterson et al., "Semantic Soundness for Language Interoperability", 2022 [Pat+22].

[9] Mates et al., "Under Control: Compositionally Correct Closure Conversion with Mutable State", 2019 [MPA19].

[10] Patterson et al., "Semantic Soundness for Language Interoperability", 2022 [Pat+22].

[11] Benton and Hur, "Biorthogonality, Step-indexing and Compiler Correctness", 2009 [BH09]; Benton and Hur, *Realizability and Compositional Compiler Correctness for a Polymorphic Language*, 2010 [BH10]; Hur and Dreyer, "A Kripke Logical Relation Between ML and Assembly", 2011 [HD11]; Hur et al., "The Marriage of Bisimulations and Kripke Logical Relations", 2012 [Hur+12].

[12] Neis et al., "Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language", 2015 [Nei+15].

[13] Wang et al., "Compiler Verification Meets Cross-Language Linking via Data Abstraction", 2014 [WCC14]; Pit-Claudel et al., "Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs", 2020 [Pit+20].

[14] Song et al., "Conditional Contextual Refinement", 2023 [Son+23].

[15] Back, "Changing Data Representation in the Refinement Calculus", 1989 [Bac89].

implementation language and a specification language), this linking shares the drawbacks of some CompCert-based approaches in that it is restricted to languages with well-bracketed call structure.

Properties of wrappers. The idea of translating between different languages via wrappers originates in the work on multi-language semantics.[16] This prior work on syntactic wrappers identified desirable properties for such wrappers, including boundary cancellation[17] and embedding-projection pairs.[18] It would be interesting to investigate how these properties can be phrased in terms of DimSum's semantic wrappers.

Process algebra. DimSum's way of modeling and relating the semantics of modular components takes inspiration from the π-calculus[19] and Communicating Sequential Processes (CSP).[20] The π-calculus and its predecessor CCS pioneered the idea of characterizing the behavior of a component in an arbitrary context using labeled transition systems, where the labels represented potential interaction with the environment, and comparing behavior using (bi-)simulations. Notably, CSP and Session-Typed variants of π-calculus[21] include dual internal and external choice constructs. They are, however, modeling concurrent process interaction (*i.e.*, offering and selecting among a set of actions), and not, as in DimSum, rely/guarantee-style contracts with the environment.

Fully abstract traces. The work on fully abstract trace semantics[22] uses events to describe the interaction between program components similar to the events of **Asm** and Rec. However, prior work either focuses on proving full abstraction of the trace semantics or uses fully abstract trace semantics to prove full abstraction of a compiler, not for reasoning about multi-language programs.

[16] Matthews and Findler, "Operational Semantics for Multi-Language Programs", 2007 [MF07]; Ahmed and Blume, "An Equivalence-Preserving CPS Translation via Multi-Language Semantics", 2011 [AB11].

[17] Perconti and Ahmed, "Verifying an Open Compiler Using Multi-language Semantics", 2014 [PA14].

[18] New and Ahmed, "Graduality from Embedding-Projection Pairs", 2018 [NA18].

[19] Milner et al., "A Calculus of Mobile Processes, I/II", 1992 [MPW92].

[20] Hoare, "Communicating Sequential Processes", 1978 [Hoa78].

[21] Padovani, "Session Types = Intersection Types + Union Types", 2010 [Pad10].

[22] Jeffrey and Rathke, "Java Jr: Fully Abstract Trace Semantics for a Core Java Language", 2005 [JR05]; Laird, "A Fully Abstract Trace Semantics for General References", 2007 [Lai07]; Abadi and Plotkin, "On Protection by Layout Randomization", 2010 [AP10]; Patrignani et al., "Secure Compilation to Protected Module Architectures", 2015 [Pat+15].

Chapter 26

Conclusion

This dissertation presented several approaches to advance formal verification towards more realistic, automated, and foundational verification of low-level programs. Concretely, this dissertation presented the following projects:

Lithium provides a domain specific language for building verification tools that combine automated verification with foundational proofs. It forms the basis of the proof automation of RefinedC and Islaris, and it is also used by ongoing work for the foundational verification of Rust code. Lithium as presented in this dissertation significantly evolved from its initial presentation,[1] and it is likely to continue evolving in the future, for example to add support for Iris's concurrency features like view-shifts or to improve the support for deducing pure facts from atoms.

[1] Sammler et al., "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types", 2021 [Sam+21b].

RefinedC shows that one can achieve foundational verification of C code with a high-degree of automation. RefinedC achieves this by combining Lithium-based proof automation with an ownership and refinement type system that uses the structure of the program to guide the proof search. Since its initial presentation described in this dissertation, RefinedC has spawned several follow-up projects, including work on memory models that enable the verification of integer-pointer casts,[2] a new approach to the verification of bitfield manipulating programs,[3] and several other ongoing projects.

[2] Lepigre et al., "VIP: Verifying Real-World C Idioms with Integer-Pointer Casts", 2022 [Lep+22].

[3] Zhu et al., "BFF: Foundational and Automated Verification of Bitfield-Manipulating Programs", 2022 [Zhu+22].

Islaris presents a new approach for scaling the verification of assembly code to authoritative and comprehensive models of real-world architectures, including Armv8-A and RISC-V. The core of this approach is a combination of (1) symbolic execution using an SMT solver to prune and simplify the large ISA model and (2) automated verification in Coq based on Lithium. We have shown that this approach allows verification of intricate case studies involving assembly code that interacts with systems features provided by the architecture.

DimSum describes a decentralized formalism for reasoning about multi-language programs. In particular, this formalism supports reasoning about the interaction between languages with different memory models and interaction behaviors (like C and assembly). This dissertation just presented a first step towards exploring this decentralized approach of multi-language semantics and verification. It would be interesting future work to explore how DimSum's approach scales to other programming language features—like closures, garbage collection, or concurrency—that are not considered in this dissertation. The core concept of DimSum—modules as transition

systems with dual non-determinism which communicate via events—is in principle quite general (as shown *e.g.*, by the field of process algebra), so we are hopeful that DimSum will provide a viable foundation for the verification of realistic multi-language programs.

Future vision: Unified verification framework for low-level programs. This dissertation presented different approaches that address various aspects of the verification of low-level programs—verification of C code, assembly code, and multi-language programs. One particularly interesting direction for future work is to combine all of these approaches into a unified framework for verifying low-level programs. In particular, when verifying an operating system or hypervisor, ideally one would be to be able to verify the C code using RefinedC, the assembly code using Islaris, and link everything together using DimSum. Additionally, in such a setting one could build a translation validation pipeline similar to the work by Sewell et al.[4] that allows transferring the verification results for the C code to the compiled assembly code. There are also many other aspects that would be interesting to explore, like scaling verification to large real-world projects, verifying security properties, and improving the usability. Overall, we hope that the work in this dissertation can provide a foundation for improving the reliability and security of the low-level programs that form the foundation of modern computer systems.

[4] Sewell et al., "Translation Validation for a Verified OS Kernel", 2013 [SMK13].

Bibliography

[17] *The RISC-V Instruction Set Manual. Volume I: User-Level ISA; Volume II: Privileged Architecture*. https://riscv.org/specifications/. May 2017.

[AB11] Amal Ahmed and Matthias Blume. "An Equivalence-Preserving CPS Translation via Multi-Language Semantics". In: *ICFP*. ACM, 2011, pp. 431–444. DOI: 10.1145/2034773.2034830.

[Alu+98] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. "Alternating Refinement Relations". In: *CONCUR*. Vol. 1466. LNCS. Springer, 1998, pp. 163–178. DOI: 10.1007/BFb0055622.

[Ama+13] Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, et al. "Certified Complexity (CerCo)". In: *FOPARA*. Vol. 8552. LNCS. Springer, 2013, pp. 1–18. DOI: 10.1007/978-3-319-12466-7_1.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory". In: *ACM Trans. Program. Lang. Syst.* 36.2 (2014), 7:1–7:74. DOI: 10.1145/2627752.

[And92] Jean-Marc Andreoli. "Logic Programming with Focusing Proofs in Linear Logic". In: *J. Log. Comput.* 2.3 (1992), pp. 297–347. DOI: 10.1093/logcom/2.3.297.

[AP01] Pablo A. Armelín and David J. Pym. "Bunched Logic Programming". In: *IJCAR*. Vol. 2083. LNCS. Springer, 2001, pp. 289–304. DOI: 10.1007/3-540-45744-5_21.

[AP10] Martín Abadi and Gordon D. Plotkin. "On Protection by Layout Randomization". In: *CSF*. IEEE Computer Society, 2010, pp. 337–351. DOI: 10.1109/CSF.2010.30.

[App14] Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014. URL: https://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers.

[Arm+19a] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, et al. "ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 71:1–71:31. DOI: 10.1145/3290384.

[Arm+19b] Alasdair Armstrong, Alastair Reid, Thomas Bauereiss, Peter Sewell, Kathryn Gray, and Anthony Fox. *Sail ARMv8.5-A ISA model*. https://github.com/rems-project/sail-arm. 2019.

[Arm+21] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. "Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models". In: *CAV*. Vol. 12759. LNCS. Springer, 2021, pp. 303–316. DOI: 10.1007/978-3-030-81685-8_14.

[Arm21] Arm. *Arm Architecture Reference Manual. Armv8, for A-profile architecture profile*. https://developer.arm.com/documentation/ddi0487/. DDI 0487G.b. 8.7 EAC updated. 8696 pages. July 2021.

[Ast+19] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. "Leveraging Rust Types for Modular Specification and Verification". In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 147:1–147:30. DOI: 10.1145/3360573.

[Bac89] Ralph-Johan Back. "Changing Data Representation in the Refinement Calculus". In: *HICSS*. Vol. 2. 1989, pp. 231–242. DOI: 10.1109/HICSS.1989.47997.

[Bat+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. "Mathematizing C++ Concurrency". In: *POPL*. ACM, 2011, pp. 55–66. DOI: 10.1145/1926385.1926394.

[Bau+16] Christoph Baumann, Mats Näslund, Christian Gehrmann, Oliver Schwarz, and Hans Thorsen. "A High Assurance Virtualization Platform for ARMv8". In: *EuCNC*. IEEE, 2016, pp. 210–214. DOI: 10.1109/EuCNC.2016.7561034.

[Bau+22] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. "Verified Security for the Morello Capability-enhanced Prototype Arm Architecture". In: *ESOP*. Vol. 13240. LNCS. Springer, 2022, pp. 174–203. DOI: 10.1007/978-3-030-99336-8_7.

[BCO04] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "A Decidable Fragment of Separation Logic". In: *FSTTCS*. Vol. 3328. LNCS. Springer, 2004, pp. 97–109. DOI: 10.1007/978-3-540-30538-5_9.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "Smallfoot: Modular Automatic Assertion Checking with Separation Logic". In: *FMCO*. Vol. 4111. LNCS. Springer, 2005, pp. 115–137. DOI: 10.1007/11804192_6.

[Bed15a] Bedrock team. *Verification of a singly linked list*. https://github.com/mit-plv/bedrock/blob/e3ff3c2cba9976ac4351caaabb4bf/Bedrock/Examples/SinglyLinkedList.v. 2015.

[Bed15b] Bedrock team. *Verification of memcpy*. https://github.com/mit-plv/bedrock/blob/e3ff3c2cba9976ac4351caaabb4bf/Bedrock/Examples/Arr.v. 2015.

[Ber+14] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. "Verified Compilation for Shared-Memory C". In: *ESOP*. Vol. 8410. LNCS. Springer, 2014, pp. 107–127. DOI: 10.1007/978-3-642-54833-8_7.

[BH09] Nick Benton and Chung-Kil Hur. "Biorthogonality, Step-indexing and Compiler Correctness". In: *ICFP*. ACM, 2009, pp. 97–108. DOI: 10.1145/1596550.1596567.

[BH10] Nick Benton and Chung-Kil Hur. *Realizability and Compositional Compiler Correctness for a Polymorphic Language*. Tech. rep. MSR-TR-2010-62. Microsoft Research, 2010. URL: https://sf.snu.ac.kr/publications/cccmsrtr.pdf.

[BJ16] Alexander Bakst and Ranjit Jhala. "Predicate Abstraction for Linked Data Structures". In: *VMCAI*. Vol. 9583. LNCS. Springer, 2016, pp. 65–84. DOI: 10.1007/978-3-662-49122-5_3.

[BR70] J. N. Buxton and Brian Randell. "Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO". In: 1970. URL: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF.

[Cao+18] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. "VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs". In: *J. Autom. Reason.* 61.1-4 (2018), pp. 367–422. DOI: 10.1007/s10817-018-9457-5.

[Cao+19] Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W. Appel. "Proof Pearl: Magic Wand as Frame". In: *CoRR* abs/1909.08789 (2019). URL: http://arxiv.org/abs/1909.08789.

[Cha16] Arthur Charguéraud. "Higher-order Representation Predicates in Separation Logic". In: *CPP*. ACM, 2016, pp. 3–14. DOI: 10.1145/2854065.2854068.

[Cha23] Tej Chajed. *SimpLang*. https://github.com/tchajed/iris-simp-lang/. 2023.

[Chl11] Adam Chlipala. "Mostly-Automated Verification of Low-Level programs in Computational Separation Logic". In: *PLDI*. ACM, 2011, pp. 234–245. DOI: 10.1145/1993498.1993526.

[Chl13] Adam Chlipala. "The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier". In: *ICFP*. ACM, 2013, pp. 391–402. DOI: 10.1145/2500365.2500592.

[Chl15] Adam Chlipala. "From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification". In: *POPL*. ACM, 2015, pp. 609–622. DOI: 10.1145/2676726.2677003.

[CJT15] Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. "Automatic Induction Proofs of Data-Structures in Imperative Programs". In: *PLDI*. ACM, 2015, pp. 457–466. DOI: 10.1145/2737924.2737984.

[CKS81] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. "Alternation". In: *J. ACM* 28.1 (1981), pp. 114–133. DOI: 10.1145/322234.322243.

[Coh+09] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. "VCC: A Practical System for Verifying Concurrent C". In: *TPHOLs*. Vol. 5674. LNCS. Springer, 2009, pp. 23–42. DOI: 10.1007/978-3-642-03359-9_2.

[Com23] CompCert team. *CompCert Arm semantics*. https://github.com/AbsInt/CompCert/blob/94d2111fa24ddbe8e71f151b985d610c2a8d74f7/aarch64/Asm.v. 2023.

[Con+07] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. "Dependent Types for Low-Level Programming". In: *ESOP*. Vol. 4421. LNCS. Springer, 2007, pp. 520–535. DOI: 10.1007/978-3-540-71316-6_35.

[Con+09] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. "Unifying Type Checking and Property Checking for Low-Level Code". In: *POPL*. ACM, 2009, pp. 302–314. DOI: 10.1145/1480881.1480921.

[Coq20] Coq-std++ team. *An extended "standard library" for Coq*. 2020. URL: https://gitlab.mpi-sws.org/iris/stdpp.

[Coq23] Coq team. *The Coq proof assistant*. https://coq.inria.fr/. 2023.

[CS16] Brian Campbell and Ian Stark. "Extracting Behaviour from an Executable Instruction Set Model". In: *FMCAD*. IEEE, 2016, pp. 33–40. URL: https://doi.org/10.1109/FMCAD.2016.7886658.

[Cuo+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A Software Analysis Perspective". In: *SEFM*. Vol. 7504. LNCS. Springer, 2012, pp. 233–247. DOI: 10.1007/978-3-642-33826-7_16.

[Dan+20] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. "RustBelt Meets Relaxed Memory". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 34:1–34:29. DOI: 10.1145/3371102.

[Dan+22] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. "Compass: Strong and Compositional Library Specifications in Relaxed Memory Separation Logic". In: *PLDI*. ACM, 2022, pp. 792–808. DOI: 10.1145/3519939.3523451.

[Das+19] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Rosu. "A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture". In: *PLDI*. ACM, 2019, pp. 1133–1148. DOI: 10.1145/3314221.3314601.

[Deg12] Ulan Degenbaev. "Formal Specification of the x86 Instruction Set Architecture". PhD thesis. Saarland University, 2012. URL: http://scidok.sulb.uni-saarland.de/volltexte/2012/4707/.

[Del00] David Delahaye. "A Tactic Language for the System Coq". In: *LPAR*. Vol. 1955. LNCS. Springer, 2000, pp. 85–95. DOI: 10.1007/3-540-44404-1_7.

[DF01] Robert DeLine and Manuel Fähndrich. "Enforcing High-Level Protocols in Low-Level Software". In: *PLDI*. ACM, 2001, pp. 59–69. DOI: 10.1145/378795.378811.

[DMS22] Thibault Dardinier, Peter Müller, and Alexander J. Summers. "Fractional Resources in Unbounded Separation Logic". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022), pp. 1066–1092. DOI: 10.1145/3563326.

[Ell+18] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. "Checked C: Making C Safe by Extension". In: *SecDev*. IEEE Computer Society, 2018, pp. 53–60. DOI: 10.1109/SecDev.2018.00015.

[EM18] Marco Eilers and Peter Müller. "Nagini: A Static Verifier for Python". In: *CAV (1)*. Vol. 10981. LNCS. Springer, 2018, pp. 596–603. DOI: 10.1007/978-3-319-96145-3_33.

[Erb+21] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. "Integration Verification across Software and Hardware for a Simple Embedded System". In: *PLDI*. ACM, 2021, pp. 604–619. DOI: 10.1145/3453483.3454065.

[Fen+08] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. "Combining Domain-Specific and Foundational Logics to Verify Complete Software Systems". In: *VSTTE*. Vol. 5295. LNCS. Springer, 2008, pp. 54–69. URL: https://doi.org/10.1007/978-3-540-87873-5_8.

[FGK19] Dan Frumin, Léon Gondelman, and Robbert Krebbers. "Semi-automated Reasoning About Non-determinism in C Expressions". In: *ESOP*. Vol. 11423. LNCS. Springer, 2019, pp. 60–87. DOI: 10.1007/978-3-030-17184-1_3.

[Flo67] Robert W. Floyd. "Nondeterministic Algorithms". In: *J. ACM* 14.4 (1967), pp. 636–644. DOI: 10.1145/321420.321422.

[Flu+16] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. "Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA". In: *POPL*. ACM, 2016, pp. 608–621. DOI: 10.1145/2837614.2837615.

[FM10] Anthony C. J. Fox and Magnus O. Myreen. "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture". In: *ITP*. Vol. 6172. LNCS. Springer, 2010, pp. 243–258. DOI: 10.1007/978-3-642-14052-5_18.

[Fox+17] Anthony C. J. Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. "Verified Compilation of CakeML to Multiple Machine-Code Targets". In: *CPP*. ACM, 2017, pp. 125–137. URL: https://doi.org/10.1145/3018610.3018621.

[Fox12] Anthony C. J. Fox. "Directions in ISA Specification". In: *ITP*. Vol. 7406. LNCS. Springer, 2012, pp. 338–344. DOI: 10.1007/978-3-642-32347-8_23.

[Fox15] Anthony C. J. Fox. "Improved Tool Support for Machine-Code Decompilation in HOL4". In: *ITP*. Vol. 9236. LNCS. Springer, 2015, pp. 187–202. DOI: 10.1007/978-3-319-22102-1_12.

[FP91] Timothy S. Freeman and Frank Pfenning. "Refinement Types for ML". In: *PLDI*. ACM, 1991, pp. 268–277. DOI: 10.1145/113445.113468.

[FW05] Carsten Fritz and Thomas Wilke. "Simulation relations for alternating Büchi automata". In: *Theor. Comput. Sci.* 338.1-3 (2005), pp. 275–314. DOI: 10.1016/j.tcs.2005.01.016.

[Gäh+22] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. "Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–31. DOI: 10.1145/3498689.

[GAK12] David Greenaway, June Andronick, and Gerwin Klein. "Bridging the Gap: Automatic Verified Abstraction of C". In: *ITP*. Vol. 7406. LNCS. Springer, 2012, pp. 99–115. DOI: 10.1007/978-3-642-32347-8_8.

[GJ13] Shilpi Goel and Warren A. Hunt Jr. "Automated Code Proofs on a Formal Model of the X86". In: *VSTTE*. Vol. 8164. LNCS. Springer, 2013, pp. 222–241. DOI: 10.1007/978-3-642-54108-7_12.

[GJK17] Shilpi Goel, Warren A. Hunt Jr., and Matt Kaufmann. "Engineering a Formal, Executable x86 ISA Simulator for Software Verification". In: *Provably Correct Systems*. NASA Monographs in Systems and Software Engineering. Springer, 2017, pp. 173–209. DOI: 10.1007/978-3-319-48628-4_8.

[Goe+14] Shilpi Goel, Warren A. Hunt Jr., Matt Kaufmann, and Soumava Ghosh. "Simulation and Formal Verification of x86 Machine-Code Programs that make System Calls". In: *FMCAD*. IEEE, 2014, pp. 91–98. DOI: 10.1109/FMCAD.2014.6987600.

[Gra+15] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. "An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors". In: *MICRO*. ACM, 2015, pp. 635–646. DOI: 10.1145/2830772.2830775.

[Gre+14] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. "Don't Sweat the Small Stuff: Formal Verification of C Code Without the Pain". In: *PLDI*. ACM, 2014, pp. 429–439. DOI: 10.1145/2594291.2594296.

[Gu+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. "Deep Specifications and Certified Abstraction Layers". In: *POPL*. ACM, 2015, pp. 595–608. DOI: 10.1145/2676726.2676975.

[Gu+18] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. "Certified Concurrent Abstraction Layers". In: *PLDI*. ACM, 2018, pp. 646–661. DOI: 10.1145/3192366.3192381.

[Gu+19] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. "Building Certified Concurrent OS Kernels". In: *Commun. ACM* 62.10 (2019), pp. 89–99. DOI: 10.1145/3356903.

[Gua+16] Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. "Provably secure memory isolation for Linux on ARM". In: *J. Comput. Secur.* 24.6 (2016), pp. 793–837. DOI: 10.3233/JCS-160558.

[Haf23] Hafnium. *Hafnium*. https://review.trustedfirmware.org/plugins/gitiles/hafnium/hafnium/+/HEAD/README.md. 2023.

[HAN08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. "Oracle Semantics for Concurrent Separation Logic". In: *ESOP*. Vol. 4960. LNCS. Springer, 2008, pp. 353–367. DOI: 10.1007/978-3-540-78739-6_27.

[HD11] Chung-Kil Hur and Derek Dreyer. "A Kripke Logical Relation Between ML and Assembly". In: *POPL*. ACM, 2011, pp. 133–146. DOI: 10.1145/1926385.1926402.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Rosu. "Defining the Undefinedness of C". In: *PLDI*. ACM, 2015, pp. 336–345. URL: https://doi.org/10.1145/2737924.2737979.

[Heu+13] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. "Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions". In: *ECOOP*. Vol. 7920. LNCS. Springer, 2013, pp. 451–476. DOI: 10.1007/978-3-642-39038-8_19.

[Heu+16] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stratified Synthesis: Automatically Learning the x86-64 Instruction Set". In: *PLDI*. ACM, 2016, pp. 237–250. URL: https://doi.org/10.1145/2908080.2908121.

[HM91] Joshua S. Hodas and Dale Miller. "Logic Programming in a Fragment of Intuitionistic Linear Logic". In: *LICS*. IEEE Computer Society, 1991, pp. 32–42. DOI: 10.1109/LICS.1991.151628.

[Hoa78] C. A. R. Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (1978), pp. 666–677. DOI: 10.1145/359576.359585.

[HPW96] James Harland, David J. Pym, and Michael Winikoff. "Programming in Lygon: An Overview". In: *AMAST*. Vol. 1101. LNCS. Springer, 1996, pp. 391–405. DOI: 10.1007/BFb0014329.

[Hur+12] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. "The Marriage of Bisimulations and Kripke Logical Relations". In: *POPL*. ACM, 2012, pp. 59–72. DOI: 10.1145/2103656.2103666.

[Isa23] Isabelle team. *The Isabelle proof assistant*. https://isabelle.in.tum.de/. 2023.

[Jac+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods*. Vol. 6617. LNCS. Springer, 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4.

[JBK13] Jonas Braband Jensen, Nick Benton, and Andrew Kennedy. "High-Level Separation Logic for Low-Level Code". In: *POPL*. ACM, 2013, pp. 301–314. DOI: 10.1145/2429069.2429105.

[Jim+02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. "Cyclone: A Safe Dialect of C". In: *USENIX*. 2002, pp. 275–288. URL: http://www.usenix.org/publications/library/proceedings/usenix02/jim.html.

[JR05] Alan Jeffrey and Julian Rathke. "Java Jr: Fully Abstract Trace Semantics for a Core Java Language". In: *ESOP*. Vol. 3444. LNCS. Springer, 2005, pp. 423–438. DOI: 10.1007/978-3-540-31987-0_29.

[Jun+15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *POPL*. ACM, 2015, pp. 637–650. DOI: 10.1145/2676726.2676980.

[Jun+16] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. "Higher-Order Ghost State". In: *ICFP*. ACM, 2016, pp. 256–269. DOI: 10.1145/2951913.2951943.

[Jun+18a] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154.

[Jun+18b] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151.

[Jun+20] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. "The Future is Ours: Prophecy Variables in Separation Logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 45:1–45:32. DOI: 10.1145/3371113.

[Jun+21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "Safe Systems Programming in Rust". In: *Commun. ACM* 64.4 (Apr. 2021), pp. 144–152. URL: https://cacm.acm.org/magazines/2021/4/251364-safe-systems-programming-in-rust/fulltext.

[Jun20] Ralf Jung. "Understanding and Evolving the Rust Programming Language". PhD thesis. Saarland University, Saarbrücken, Germany, 2020. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647.

[Kai+17] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris". In: *ECOOP*. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 17:1–17:29. DOI: 10.4230/LIPIcs.ECOOP.2017.17.

[Kan+16] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. "Lightweight Verification of Separate Compilation". In: *POPL*. ACM, 2016, pp. 178–190. DOI: 10.1145/2837614.2837642.

[Ken+13] Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. "Coq: The world's best macro assembler?" In: *PPDP*. ACM, 2013, pp. 13–24. DOI: 10.1145/2505879.2505897.

[Keu+22] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. "Verified Symbolic Execution with Kripke Specification Monads (and No Meta-programming)". In: *Proc. ACM Program. Lang.* 6.ICFP (2022), pp. 194–224. DOI: 10.1145/3547628.

[KFM22] Hrutvik Kanabar, Anthony C. J. Fox, and Magnus O. Myreen. "Taming an Authoritative Armv8 ISA Specification: L3 Validation and CakeML Compiler Verification". In: *ITP*. Vol. 237. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 20:1–20:22. DOI: 10.4230/LIPIcs.ITP.2022.20.

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. "seL4: Formal Verification of an OS Kernel". In: *SOSP*. ACM, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596.

[Kle+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive Formal Verification of an OS Microkernel". In: *ACM Trans. Comput. Syst.* 32.1 (2014), 2:1–2:70. DOI: 10.1145/2560537.

[Kre+17] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. "The Essence of Higher-Order Concurrent Separation Logic". In: *ESOP*. Vol. 10201. LNCS. Springer, 2017, pp. 696–723. DOI: 10.1007/978-3-662-54434-1_26.

[Kre14] Robbert Krebbers. "An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C". In: *POPL*. ACM, 2014, pp. 101–112. DOI: 10.1145/2535838.2535878.

[Kre15] Robbert Krebbers. "The C standard formalized in Coq". PhD thesis. Radboud University Nijmegen, 2015. URL: https://robbertkrebbers.nl/thesis.html.

[KS21] Jérémie Koenig and Zhong Shao. "CompCertO: Compiling Certified Open C Components". In: *PLDI*. ACM, 2021, pp. 1095–1109. DOI: 10.1145/3453483.3454097.

[KW13] Robbert Krebbers and Freek Wiedijk. "Separation Logic for Non-local Control Flow and Block Scope Variables". In: *FoSSaCS*. Vol. 7794. LNCS. Springer, 2013, pp. 257–272. DOI: 10.1007/978-3-642-37075-5_17.

[Lai07] James Laird. "A Fully Abstract Trace Semantics for General References". In: *ICALP*. Vol. 4596. LNCS. Springer, 2007, pp. 667–679. DOI: 10.1007/978-3-540-73420-8_58.

[LB08] Xavier Leroy and Sandrine Blazy. "Formal verification of a C-like memory model and its uses for verifying program transformations". In: *JAR* 41.1 (2008), pp. 1–31. DOI: 10.1007/s10817-008-9099-0.

[Lee+17] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. "Taming Undefined Behavior in LLVM". In: *PLDI*. ACM, 2017, pp. 633–647. DOI: 10.1145/3062341.3062343.

[Lep+22] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. "VIP: Verifying Real-World C Idioms with Integer-Pointer Casts". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–32. DOI: 10.1145/3498681.

[Ler+12] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. *The CompCert Memory Model, Version 2*. Tech. rep. RR-7987. Inria, 2012. URL: https://inria.hal.science/hal-00703441.

[Ler06] Xavier Leroy. "Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant". In: *POPL*. ACM, 2006, pp. 42–54. DOI: 10.1145/1111037.1111042.

[Li+21] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. "A Secure and Formally Verified Linux KVM Hypervisor". In: *IEEE Symposium on Security and Privacy*. IEEE, 2021, pp. 1782–1799. DOI: 10.1109/SP40001.2021.00049.

[LM09] K. Rustan M. Leino and Peter Müller. "A Basis for Verifying Multi-threaded Programs". In: *ESOP*. Vol. 5502. LNCS. Springer, 2009, pp. 378–393. DOI: 10.1007/978-3-642-00590-9_27.

[Lor+20] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. "Armada: Low-Effort Verification of High-Performance Concurrent Programs". In: *PLDI*. ACM, 2020, pp. 197–210. DOI: 10.1145/3385412.3385971.

[LP14] Wonyeol Lee and Sungwoo Park. "A Proof System for Separation Logic with Magic Wand". In: *POPL*. ACM, 2014, pp. 477–490. DOI: 10.1145/2535838.2535871.

[LS09] Dirk Leinenbach and Thomas Santen. "Verifying the Microsoft Hyper-V Hypervisor with VCC". In: *FM*. Vol. 5850. LNCS. Springer, 2009, pp. 806–809. DOI: 10.1007/978-3-642-05089-3_51.

[LSQ18] Quang Loc Le, Jun Sun, and Shengchao Qin. "Frame Inference for Inductive Entailment Proofs in Separation Logic". In: *TACAS (1)*. Vol. 10805. LNCS. Springer, 2018, pp. 41–60. DOI: 10.1007/978-3-319-89960-2_3.

[Mak+21] Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. "Gillian, Part II: Real-World Verification for JavaScript and C". In: *CAV (2)*. Vol. 12760. LNCS. Springer, 2021, pp. 827–850. DOI: 10.1007/978-3-030-81688-9_38.

[MAN17] William Mansky, Andrew W. Appel, and Aleksey Nogin. "A Verified Messaging System". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 87:1–87:28. DOI: 10.1145/3133911.

[MCB14] Gregory Malecha, Adam Chlipala, and Thomas Braibant. "Compositional Computational Reflection". In: *ITP*. Vol. 8558. LNCS. Springer, 2014, pp. 374–389. DOI: 10.1007/978-3-319-08970-6_24.

[Mem+16] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. "Into the Depths of C: Elaborating the De Facto Standards". In: *PLDI*. ACM, 2016, pp. 1–15. DOI: 10.1145/2908080.2908081.

[Mem+19] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. "Exploring C Semantics and Pointer Provenance". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 67:1–67:32. DOI: 10.1145/3290380.

[MF07] Jacob Matthews and Robert Bruce Findler. "Operational Semantics for Multi-Language Programs". In: *POPL*. ACM, 2007, pp. 3–10. DOI: 10.1145/1190216.1190220.

[MG07] Magnus O. Myreen and Michael J. C. Gordon. "Hoare Logic for Realistically Modelled Machine Code". In: *TACAS*. Vol. 4424. LNCS. Springer, 2007, pp. 568–582. DOI: 10.1007/978-3-540-71209-1_44.

[MGS08] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. "Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic". In: *FMCAD*. IEEE, 2008, pp. 1–8. URL: https://doi.org/10.1109/FMCAD.2008.ECP.24.

[MGS12] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. "Decompilation into Logic - Improved". In: *FMCAD*. IEEE, 2012, pp. 78–81. URL: https://ieeexplore.ieee.org/document/6462558/.

[Mil78] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999. ISBN: 978-0-521-65869-0.

[MKG22] Ike Mulder, Robbert Krebbers, and Herman Geuvers. "Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris". In: *PLDI*. ACM, 2022, pp. 809–824. DOI: 10.1145/3519939.3523432.

[MMS08] Stefan Maus, Michal Moskal, and Wolfram Schulte. "Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving". In: *AMAST*. Vol. 5140. LNCS. Springer, 2008, pp. 284–298. DOI: 10.1007/978-3-540-79980-1_22.

[Mor+12] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. "RockSalt: Better, Faster, Stronger SFI for the x86". In: *PLDI*. ACM, 2012, pp. 395–404. DOI: 10.1145/2254064.2254111.

[MPA19] Phillip Mates, Jamie Perconti, and Amal Ahmed. "Under Control: Compositionally Correct Closure Conversion with Mutable State". In: *PPDP*. ACM, 2019, 16:1–16:15. DOI: 10.1145/3354166.3354181.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. "A Calculus of Mobile Processes, I/II". In: *Inf. Comput.* 100.1 (1992), pp. 1–40. DOI: 10.1016/0890-5401(92)90008-4.

[MSS16a] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution". In: *CAV (1)*. Vol. 9779. LNCS. Springer, 2016, pp. 405–425. DOI: 10.1007/978-3-319-41528-4_22.

[MSS16b] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *VMCAI*. Vol. 9583. LNCS. Springer, 2016, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2.

[Mun+21] Prashanth Mundkur, Jon French, Brian Campbell, Robert Norton-Wright, Alasdair Armstrong, Thomas Bauereiss, Shaked Flur, Christopher Pulte, and Peter Sewell. *Sail RISC-V ISA model*. https://github.com/riscv/sail-riscv. 2021.

[Myr09] Magnus Oskar Myreen. "Formal verification of machine-code programs". PhD thesis. University of Cambridge, UK, 2009. URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.611450.

[NA18] Max S. New and Amal Ahmed. "Graduality from Embedding-Projection Pairs". In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 73:1–73:30. DOI: 10.1145/3236768.

[Nei+15] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. "Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language". In: *ICFP*. ACM, 2015, pp. 166–178. DOI: 10.1145/2784731.2784764.

[Nel+19] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. "Scaling symbolic evaluation for automated verification of systems code with Serval". In: *SOSP*. ACM, 2019, pp. 225–242. DOI: 10.1145/3341301.3359641.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. "CCured: Type-Safe Retrofitting of Legacy Code". In: *POPL*. ACM, 2002, pp. 128–139. DOI: 10.1145/503272.503286.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *CSL*. Vol. 2142. LNCS. Springer, 2001, pp. 1–19. DOI: 10.1007/3-540-44802-0_1.

[PA14] James T. Perconti and Amal Ahmed. "Verifying an Open Compiler Using Multi-language Semantics". In: *ESOP*. Vol. 8410. LNCS. Springer, 2014, pp. 128–148. DOI: 10.1007/978-3-642-54833-8_8.

[PA19] Daniel Patterson and Amal Ahmed. "The Next 700 Compiler Correctness Theorems (Functional Pearl)". In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 85:1–85:29. DOI: 10.1145/3341689.

[Pad10] Luca Padovani. "Session Types = Intersection Types + Union Types". In: *ITRS*. Vol. 45. EPTCS. 2010, pp. 71–89. DOI: 10.4204/EPTCS.45.6.

[Pat+15] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. "Secure Compilation to Protected Module Architectures". In: *ACM Trans. Program. Lang. Syst.* 37.2 (2015), 6:1–6:50. DOI: 10.1145/2699503.

[Pat+17] Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. "FunTAL: Reasonably Mixing a Functional Language with Assembly". In: *PLDI*. ACM, 2017, pp. 495–509. DOI: 10.1145/3062341.3062347.

[Pat+22] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. "Semantic Soundness for Language Interoperability". In: *PLDI*. ACM, 2022, pp. 609–624. DOI: 10.1145/3519939.3523703.

[Pat20] Marco Patrignani. "Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets". In: *CoRR* abs/2001.11334 (2020). URL: https://arxiv.org/abs/2001.11334.

[Pit+20] Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. "Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs". In: *IJCAR*. Vol. 12167. LNCS. 2020, pp. 119–137. DOI: 10.1007/978-3-030-51054-1_7.

[PMS21] Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. "Formally Validating a Practical Verification Condition Generator". In: *CAV (2)*. Vol. 12760. LNCS. Springer, 2021, pp. 704–727. DOI: 10.1007/978-3-030-81688-9_33.

[PP13] François Pottier and Jonathan Protzenko. "Programming with Permissions in Mezzo". In: *ICFP*. ACM, 2013, pp. 173–184. DOI: 10.1145/2500365.2500598.

[PS12] Matthew J. Parkinson and Alexander J. Summers. "The Relationship Between Separation Logic and Implicit Dynamic Frames". In: *Log. Methods Comput. Sci.* 8.3 (2012). DOI: 10.2168/LMCS-8(3:1)2012.

[Pul+18] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. "Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 19:1–19:29. DOI: 10.1145/3158107.

[Pul+23] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. "CN: Verifying Systems C Code with Separation-Logic Refinement Types". In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 1–32. DOI: 10.1145/3571194.

[PWZ14] Ruzica Piskac, Thomas Wies, and Damien Zufferey. "Automating Separation Logic with Trees and Data". In: *CAV*. Vol. 8559. LNCS. Springer, 2014, pp. 711–728. DOI: 10.1007/978-3-319-08867-9_47.

[Ram+15] Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jérémie Koenig, and Yuchen Fu. "A Compositional Semantics for Verified Separate Compilation and Linking". In: *CPP*. ACM, 2015, pp. 3–14. DOI: 10.1145/2676724.2693167.

[RES10] Grigore Rosu, Chucky Ellison, and Wolfram Schulte. "Matching Logic: An Alternative to Hoare/Floyd Logic". In: *AMAST*. Vol. 6486. LNCS. Springer, 2010, pp. 142–162. DOI: 10.1007/978-3-642-17796-5_9.

[Rew03] Ingrid Rewitzky. "Binary Multirelations". In: *Theory and Applications of Relational Structures as Knowledge Instruments*. Vol. 2929. LNCS. Springer, 2003, pp. 256–271. DOI: 10.1007/978-3-540-24615-2_12.

[Rey+16] Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. "A Decision Procedure for Separation Logic in SMT". In: *ATVA*. Vol. 9938. LNCS. 2016, pp. 244–261. DOI: 10.1007/978-3-319-46520-3_16.

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *LICS*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[RIS19] RISC-V team. *ISA Formal Spec Public Review*. https://github.com/riscvarchive/ISA_Formal_Spec_Public_Review/blob/master/comparison_table.md. 2019.

[RKJ08] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. "Liquid Types". In: *PLDI*. ACM, 2008, pp. 159–169. DOI: 10.1145/1375581.1375602.

[RKJ10] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. "Low-Level Liquid Types". In: *POPL*. ACM, 2010, pp. 131–144. DOI: 10.1145/1706299.1706316.

[RMV22] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. "Extending Intel-x86 Consistency and Persistency: Formalising the Semantics of Intel-x86 Memory Types and Non-Temporal Stores". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–31. DOI: 10.1145/3498683.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2010. DOI: 10.1007/978-1-84882-258-0.

[RS10] Grigore Rosu and Traian-Florin Serbanuta. "An Overview of the K Semantic Framework". In: *J. Log. Algebraic Methods Program.* 79.6 (2010), pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012.

[Rus23] Rust team. *The Rust programming language*. https://rust-lang.org. 2023.

[Sam+21a] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. *Artifact and Appendix of "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types"*. Mar. 2021. DOI: 10.5281/zenodo.4649822.

[Sam+21b] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types". In: *PLDI*. ACM, 2021, pp. 158–174. DOI: 10.1145/3453483.3454036.

[Sam+22] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. "Islaris: Verification of Machine Code Against Authoritative ISA Semantics". In: *PLDI*. ACM, 2022, pp. 825–840. DOI: 10.1145/3519939.3523434.

[Sam+23] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. "DimSum: A Decentralized Approach to Multi-language Semantics and Verification". In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 775–805. DOI: 10.1145/3571220.

[San+20] José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. "Gillian, Part i: A Multi-language Platform for Symbolic Execution". In: *PLDI*. ACM, 2020, pp. 927–942. DOI: 10.1145/3385412.3386014.

[Sar+09] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. "The Semantics of x86-CC Multiprocessor Machine Code". In: *POPL*. ACM, 2009, pp. 379–391. DOI: 10.1145/1480881.1480929.

[Sar+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. "Understanding POWER Multiprocessors". In: *PLDI*. ACM, 2011, pp. 175–186. DOI: 10.1145/1993498.1993520.

[SB14] Kasper Svendsen and Lars Birkedal. "Impredicative Concurrent Abstract Predicates". In: *ESOP*. Vol. 8410. LNCS. Springer, 2014, pp. 149–168. DOI: 10.1007/978-3-642-54833-8_9.

[Sch16] Malte Schwerhoff. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". PhD thesis. ETH Zurich, Zürich, Switzerland, 2016. DOI: 10.3929/ethz-a-010835519.

[Šev+13] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. "CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency". In: *J. ACM* 60.3 (2013), 22:1–22:50. DOI: 10.1145/2487241.2487248.

[Sew+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors". In: *Commun. ACM* 53.7 (2010), pp. 89–97. DOI: 10.1145/1785414.1785443.

[Sha+05] Zhong Shao, Valery Trifonov, Bratin Saha, and Nikolaos Papaspyrou. "A Type System for Certified Binaries". In: *ACM Trans. Program. Lang. Syst.* 27.1 (2005), pp. 1–45. URL: https://doi.org/10.1145/1053468.1053469.

[Sim+20] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. "ARMv8-A System Semantics: Instruction Fetch in Relaxed Architectures". In: *ESOP*. Vol. 12075. LNCS. Springer, 2020, pp. 626–655. DOI: 10.1007/978-3-030-44914-8_23.

[Sim+22] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. "Relaxed virtual memory in Armv8-A". In: *ESOP*. Vol. 13240. LNCS. Springer, 2022, pp. 143–173. DOI: 10.1007/978-3-030-99336-8_6.

[SK20] Hira Taqdees Syeda and Gerwin Klein. "Formal Reasoning Under Cached Address Translation". In: *J. Autom. Reason.* 64.5 (2020), pp. 911–945. DOI: 10.1007/s10817-019-09539-7.

[SMK13] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. "Translation Validation for a Verified OS Kernel". In: *PLDI*. ACM, 2013, pp. 471–482. DOI: 10.1145/2491956.2462183.

[SNB15] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. "Mechanized Verification of Fine-grained Concurrent Programs". In: *PLDI*. ACM, 2015, pp. 77–87. DOI: 10.1145/2737924.2737964.

[Son+20] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. "CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 23:1–23:31. DOI: 10.1145/3371091.

[Son+23] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. "Conditional Contextual Refinement". In: *POPL*. ACM, 2023. DOI: 10.1145/3571232.

[Spi+21] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. "Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic". In: *PLDI*. ACM, 2021. URL: https://doi.org/10.1145/3453483.3454031.

[Spr+20] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. "Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 152:1–152:31. DOI: 10.1145/3428220.

[Ste+15] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. "Compositional CompCert". In: *POPL*. ACM, 2015, pp. 275–287. DOI: 10.1145/2676726.2676985.

[Ste14] Andrei Stefanescu. "MatchC: A Matching Logic Reachability Verifier Using the K Framework". In: *Electron. Notes Theor. Comput. Sci.* 304 (2014), pp. 183–198. DOI: 10.1016/j.entcs.2014.05.010.

[Swa+06] Nikhil Swamy, Michael W. Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. "Safe Manual Memory Management in Cyclone". In: *Sci. Comput. Program.* 62.2 (2006), pp. 122–144. DOI: 10.1016/j.scico.2006.02.003.

[SWM00] Frederick Smith, David Walker, and J. Gregory Morrisett. "Alias Types". In: *ESOP*. Vol. 1782. LNCS. Springer, 2000, pp. 366–381. DOI: 10.1007/3-540-46425-5_24.

[Ta+18] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. "Automated Lemma Synthesis in Symbolic-Heap Separation Logic". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 9:1–9:29. DOI: 10.1145/3158097.

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. USA: Prentice Hall Press, 2014. ISBN: 013359162X. URL: https://dl.acm.org/doi/book/10.5555/2655363.

[Tok23] Tokei team. *Tokei*. https://github.com/XAMPPRocky/tokei. 2023.

[Tom+20] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. "ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs". In: *ESOP*. Vol. 12075. LNCS. Springer, 2020, pp. 684–714. DOI: 10.1007/978-3-030-44914-8_25.

[Val+22] Arthur Oliveira Vale, Paul-André Melliès, Zhong Shao, Jérémie Koenig, and Léo Stefanesco. "Layered and Object-Based Game Semantics". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–32. DOI: 10.1145/3498703.

[Var95] Moshe Y. Vardi. "Alternating Automata and Program Verification". In: *Computer Science Today*. Vol. 1000. LNCS. Springer, 1995, pp. 471–485. DOI: 10.1007/BFb0015261.

[VCC16] VCC team. *Verification of a singly linked list*. https://github.com/microsoft/vcc/blob/47f3f33d459f5fd9233203ec3d5d2fc803/vcc/Docs/Tutorial/c/7.2.list.c. 2016.

[Ver19] Verifast team. *Verification of a binary search tree*. https://github.com/verifast/verifast/blob/8bc966726de829749eaf916ec3863bf294/examples/sorted_bintree.c. 2019.

[VJP15] Frédéric Vogels, Bart Jacobs, and Frank Piessens. "Featherweight VeriFast". In: *Log. Methods Comput. Sci.* 11.3 (2015). DOI: 10.2168/LMCS-11(3:19)2015.

[Vog+11] Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. "Annotation Inference for Separation Logic Based Verifiers". In: *FMOODS/FORTE*. Vol. 6722. LNCS. Springer, 2011, pp. 319–333. DOI: 10.1007/978-3-642-21461-5_21.

[VST20] VST team. *Verification of a binary search tree*. https://github.com/PrincetonUniversity/VST/blob/14e6b3a79a9685a478786436c6f0a45dc44c3d52/progs/verif_bst.v. 2020.

[Wan+12] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. "Undefined behavior: What happened to my code?" In: *APSys*. ACM, 2012, p. 9. DOI: 10.1145/2349896.2349905.

[WC19] Qinshi Wang and Qinxiang Cao. "VST-A: A Foundationally Sound Annotation Verifier". In: *CoRR* abs/1909.00097 (2019). URL: http://arxiv.org/abs/1909.00097.

[WCC14] Peng Wang, Santiago Cuellar, and Adam Chlipala. "Compiler Verification Meets Cross-Language Linking via Data Abstraction". In: *OOPSLA*. ACM, 2014, pp. 675–690. DOI: 10.1145/2660193.2660201.

[Win+09] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. "Mind the Gap". In: *TPHOLs*. Vol. 5674. LNCS. Springer, 2009, pp. 500–515. DOI: 10.1007/978-3-642-03359-9_34.

[Wol+21] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. "Gobra: Modular Specification and Verification of Go Programs". In: *CAV (1)*. Vol. 12759. LNCS. Springer, 2021, pp. 367–379. DOI: 10.1007/978-3-030-81685-8_17.

[WWS19] Yuting Wang, Pierre Wilke, and Zhong Shao. "An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 62:1–62:30. DOI: 10.1145/3290375.

[Xi07] Hongwei Xi. "Dependent ML: An Approach to Practical Programming with Dependent Types". In: *J. Funct. Program.* 17.2 (2007), pp. 215–286. DOI: 10.1017/S0956796806006216.

[Yan+08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. "Scalable Shape Analysis for Systems Code". In: *CAV*. Vol. 5123. LNCS. Springer, 2008, pp. 385–398. DOI: 10.1007/978-3-540-70545-1_36.

[Zhu+22] Fengmin Zhu, Michael Sammler, Rodolphe Lepigre, Derek Dreyer, and Deepak Garg. "BFF: Foundational and Automated Verification of Bitfield-Manipulating Programs". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022), pp. 1613–1638. DOI: 10.1145/3563345.