Concurrent Separation Logics for Safety, Refinement, and Security

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M. van Krieken,
volgens besluit van het college van decanen
in het openbaar te verdedigen op vrijdag 12 maart 2021
om 11.30 uur precies

door

Daniil Frumin

geboren op 21 januari 1993
te Krasnojarsk, Rusland

Promotor:

- Prof. dr. Herman Geuvers

Copromotoren:

- Dr. Robbert Krebbers
- Dr. Freek Wiedijk

Manuscriptcommissie:

- Prof. dr. Sven Bodo Scholz (voorzitter)
- Prof. dr. Peter O'Hearn (University College London, Verenigd Koninkrijk)
- Prof. dr. Derek Dreyer (MPI-SWS, Duitsland)
- Prof. dr. Philippa Gardner (Imperial College London, Verenigd Koninkrijk)
- Dr. Aslan Askarov (Aarhus University, Denemarken)

Radboud University 

This research has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and the Radboud University.

This research has been funded by NWO (STW project "Sovereign", project number STW.14319) and has furthermore been supported by COST Action EUTypes (CA15123).

Cover design: Yijun Guo
Printed by: Gildeprint
ISBN: 9789464191288

Acknowledgments

While it is only my name that stands on the cover of this thesis, the truth is that this manuscript would not have been possible without the people who have supported me during these years. First and foremost, I would like to thank my daily supervisor Robbert. I always enjoyed our discussions and always looked forward to our day-long meetings full of exciting research and programming. Robbert, I am positive that your influence on my research, my writing and programming styles will extend beyond my PhD (with the main difference being that you can no longer prevent me from using Fraktur).

I would also like to thank my other supervisors, Herman and Freek; working with you was a pleasure, be it preparing for the Type Theory course or just discussing research. Your enthusiasm was contagious. Thank you for supporting me and guiding me through my PhD.

Speaking of my mentors, I can't help but mention Lars Birkedal, who guided me on many occasions. His influence on the work presented in this thesis and my research outlook in general cannot be overstated. Thanks to you, Lars, my visits to Aarhus (and I was privileged enough to have several such visits) were always productive and enlightening.

Robbert and Lars were my main collaborators, but I was lucky enough to work with other amazing people along the way. Niels and Léon, thank you for being great friends as well as great collaborators. I had so much fun doing pair programming/proving with both of you. I will miss your jokes and all the music discussions that we had. I sincerely hope that we can collaborate again in the future.

Thanks to Niels, I also had the pleasure of working with Marco Maggesi and Benedikt Ahrens. I would like to thank them for all the wonderful type theory discussions and all the fresh cannoli we had in Florence.

I would also like to mention the manuscript committee, Aslan Askarov, Derek Dreyer, Philippa Gardner, Peter O'Hearn, and Sven-Bodo Scholz. Thank you for taking the time to read and assess my thesis.

Of course, getting through a PhD was not just about research and writing. At this point I am almost contractually obliged to thank everyone who drank Brouwers and ate kapsalon with me on the third floor, as well as all of the people who dared me to play fußball with them. In particular, Guillaume, Jon, Kenta, Ko, the Joosts, Paulus, Joshua, Bas & Bram (thank you for fixing up the espresso machine, it was worth getting addicted to coffee), Freek V., Carlo. Big thanks to the Foundations group at RU, which includes, aside from some of the already mentioned people, Ike, Jules, Jana, and Jurriaan. Ingrid, thank you for helping me with navigating through the university structures. Shout out to all the Aarhus people who spent time with me there, Aleš, Amin, Marit, Simon F.V., Simon G., Thomas, and all the others. I am also grateful to the Groningen people that helped me with settling in, Jorge, Hylke, Sofia, Marie-Anne.

Marc, thank you for being a great buddy, a great office mate, and agreeing to be my paranymph. I was lucky to be on the same project as you. Sander, my dear friend,

thank you so much. Without your support and friendship I would have gone crazy multiple times over.

My dear parents, Elena and Isak, thank you for offering me your unconditional love and support during my studies and beyond. Pepa, thank you for being there for me, for bearing with me, and for making me a better person. You really inspire and challenge me in many positive ways.

Thank you all.

Dan Frumin
Groningen, January 2021

Моим родителям, Лене и Исаку.

Contents

1 Introduction **1**
- 1.1 Concurrent separation logics . . . 3
- 1.2 Mechanized reasoning . . . 6
- 1.3 Logics introduced in this thesis . . . 8
- 1.4 Contributions and outline . . . 12

2 Background on separation logic **17**
- 2.1 Syntax and semantics of HeapLang . . . 17
- 2.2 Basics of Iris . . . 20
- 2.3 Invariants in Iris . . . 31
- 2.4 Custom ghost state in Iris . . . 34
- 2.5 The Coq mechanization . . . 40
- 2.6 Defining custom logics in Iris . . . 43

3 λMC: a logic for non-determinsim in C expressions **45**
- 3.1 Introduction . . . 45
- 3.2 λMC: A monadic definitional semantics of C . . . 48
- 3.3 Separation logic with weakest preconditions for λMC . . . 53
- 3.4 Soundness of weakest preconditions for λMC . . . 58
- 3.5 A symbolic executor for λMC . . . 63
- 3.6 A verification condition generator for λMC . . . 66
- 3.7 Discussion . . . 69
- 3.8 Related work . . . 71

4 ReLoC: a logic for proving contextual refinements **75**
- 4.1 Introduction . . . 75
- 4.2 The programming language . . . 80
- 4.3 A tour of ReLoC . . . 83
- 4.4 A closer look at ReLoC . . . 95
- 4.5 Relational specifications in ReLoC . . . 101
- 4.6 Speculative reasoning using prophecy variables . . . 114
- 4.7 The logical relations model of ReLoC . . . 121
- 4.8 The Coq mechanization of ReLoC . . . 126
- 4.9 Related work . . . 131
- 4.10 Discussion and conclusion . . . 135

5 SeLoC: a logic for proving non-interference **139**
- 5.1 Introduction . . . 139

5.2 Motivating examples . . . 141
5.3 Preliminaries . . . 146
5.4 Overview of SeLoC . . . 148
5.5 Type system and logical relations . . . 155
5.6 Modular separation logic specifications . . . 160
5.7 Soundness . . . 164
5.8 Mechanization in Coq . . . 167
5.9 Discussion . . . 167
5.10 HOCAP-style modular specifications . . . 170
5.11 Related work . . . 170
5.12 Conclusions and future work . . . 174

Bibliography **175**

Summary **191**

Samenvatting **193**

Titles in the IPA Dissertation Series since 2018 **195**

Research Data Management **199**

About the author **201**


WP e {v. WP
WP K[e] {P
{P} e {v. Q} ≜
□(P -∗ WP e {v. Q}
□P⟧(k, m)
= ⟦P⟧(k,
◇P =
P / ▷P
P ⊢ Q / ▷P ⊢ ▷Q
▷P ⊢ P / ⊢ P

1 Introduction

In a 2002 lecture, Donald Knuth proclaimed that "software is hard" [Knu02]. And computer scientists tend to agree with this sentiment. In fact, many would say that software has only been getting harder since then. Software has been getting more complex, in part due to the proliferation of *concurrent* systems. That is, systems comprised of several components (*threads*), which share common resources, communicate with each other, and transfer resources between each other. Unfortunately for the users and developers of such systems, and somewhat fortunately for the computer science researchers, concurrent systems are notoriously hard to analyze. They are hard to analyze because concurrent systems can exhibit a large amount of possible behaviors. On a more technical level, the complexity of a concurrent system grows exponentially in the number of components, when we measure the complexity in terms of the interleavings of behaviors of the individual components.

To certify concurrent software systems and to ensure the absence of bugs, researchers have applied various formal verification techniques. The goal of formal verification is to establish the formal "correctness" of software. What it means for software to be "correct" depends on the kind of property you want to establish. There are various such properties of interest, some of which are tricky to verify in the context of concurrent programs. This thesis focuses on the verification of three of these properties:

1. safety: a program is *safe* if it does not crash or perform illegal operations (*e.g.*, does not dereference a freed pointer);
2. refinement: a program *refines* another program if the observable behaviors of the first program are included in the observable behaviors of the second one;
3. security: a program is *secure* if its execution does not leak its secret data to attackers.

The goal of this thesis is to devise verification methods for establishing these properties and reasoning about them for concurrent programs.

To evaluate our efforts, we put forward the following desiderata:

- *Local reasoning.* We want to be able to reason about threads in isolation, and talk about the specific parts of the state that a thread/a part of the program reads or modifies. If a program has been verified locally, then it can be put in a bigger context without sacrificing the property that we have already verified.
- *Compositionality.* We want to be able to reason about different program modules compositionally. For example, we want to formulate specifications for program

libraries that are *modular*. That is, the specifications should describe in what context and under which conditions the library satisfies the desired property. Such specifications should be expressive enough to allow many different clients of the library to be verified based only on the specification of the library, without referencing the source code.

- *Clear and machine-checked soundness theorem.* We want to formulate each property clearly in terms of the operational semantics of the programming language and independently of the verification method. The corresponding verification method should come with a *soundness statement*, ensuring that the method yields the desired property and conforms with the operational semantics. The soundness statement and its proof should be mechanized.
- *Effective formal reasoning.* We want to be able to use our methods to verify programs formally. This entails not only that our methods are sound, but that they can be used to verify actual programs. In turn this requires tool support that is formally connected to the verified soundness theorem.

In line with these criteria, we take an approach that is based on *mechanized concurrent separation logics*.

A concurrent separation logic is a *program logic*. That is, a formal system that allow us to reason about programs and verify their properties. It stems from sequential *separation logic*, an extension of Hoare logic which has been designed to handle sequential programs that manipulate pointers [Rey02; ORY01]. O'Hearn and Brookes noticed that separation logic can be naturally extended for reasoning about concurrent stateful programs. This observation led them to develop Concurrent Separation Logic (CSL) [OHe07; Bro07], which won them the 2016 Gödel Prize. By basing our methods on concurrent separation logics, we ensure that we are able to reason *locally* and *compositionally* about concurrent programs. Separation logic satisfies our demand for *local reasoning* because specifications in separation logic only pertain to the parts of the state that are operated on by the program at hand. Sequential separation logic already allows for a form of compositionality: components that operate on independent parts of the state can be verified independently. CSL and its descendants (which we refer to as "concurrent separation logics") allow further compositionality on the level of threads, even in the presence of some shared state.

In order to connect concurrent separation logics to *mechanized soundness theorems* and to enable *effective formal reasoning* in the logics themselves, we mechanize the logics in a *proof assistant*. Proof assistants, like Coq [Coq20], are computer programs that allow the user to formulate machine-checked proofs of mathematical statements. These proofs are written in a formal logic (*e.g.*, dependent type theory, in the case of Coq) and are verified by a computer. Having a machine-checked proof of a mathematical statement gives us more certainty about its correctness: the proof assistant ensures that the fully formal proof does not skip any details, and that every reasoning step is correct.

Mechanizing a program logic in a proof assistant has two advantages. First, as program logics get more complex, the soundness proofs get more complicated. As such, in informal pen-and-paper proofs of soundness mistakes can be easily made. Proof assistants give us additional assurance that programs logics are actually sound.

If we obtain a *mechanized soundness theorem* for the program logic, then we can compose it with a proof about a specific program inside the program logic. This way, we obtain a closed proof that the program satisfies a desired property. The statements in the obtained proof do not refer to the program logic at all.

The second advantage to having a mechanized program logic is that it enables *effective formal reasoning* in the program logic itself. That is, employing the program logic to formally reason about concrete programs. Mechanizing reasoning inside the program logic bolsters our confidence in the correctness of our proofs, because it forces us to pay attention to details that we might have otherwise overlooked in a pen-and-paper proof. Furthermore, a good mechanization alleviates the user's effort by automatically discharging certain subgoals or cases. As a result, the user has to do less tedious work. For example, in one of the chapters in this thesis we design and verify a *verification condition generator*—a procedure that helps with automated verification of certain programs.

In short, in this thesis we develop mechanized concurrent separation logics for reasoning about safety, refinement, and security of concurrent programs. All the material that we present in this thesis has been formalized in the Coq proof assistant, including the soundness proofs and the specific examples and case studies that we consider. The URL for the Coq formalization can be found at the end of this chapter.

In the remainder of the introduction we provide the context needed to understand the place of this thesis in the research landscape, and explain the main contributions. We describe in more details the roles of concurrent separation logic (Section 1.1) and mechanized reasoning (Section 1.2) in this thesis. We then describe more precisely the kind of properties we want to verify with our logics (Section 1.3), and give an outline for the rest of the thesis (Section 1.4).

1.1 Concurrent separation logics

When I first started teaching myself separation logic at the beginning of my PhD, I was somewhat taken aback by the fact that both separation logic and concurrent separation logic are relatively recent inventions/discoveries. In my mind, something that offers so much conceptual clarity with so little technical overhead must have been around for ages, refined and distilled to the form that we are familiar today. But, to my surprise, the origins of separation logic stem from the early 2000s, and not from the late 1970s, as I had initially guessed.

Separation logic has been developed by Reynolds, O'Hearn, Ishtiaq, and Yang [Rey02; IO01; ORY01] as an extension of Hoare logic, based on the ideas introduced by O'Hearn and Pym [OP99] in their logic of bunched implications. Propositions in separation logic are evaluated over the program state (*the heap*). The *points-to* connective of separation logic $\ell \mapsto v$ states that the current heap contains the location ℓ with the value v. The *separating conjunction* $P * Q$ states that the current heap can be subdivided into two disjoint pieces, which satisfy P and Q respectively. In particular $\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2$ implies that the locations ℓ_1 are ℓ_2 are different, because $\ell_1 \mapsto v_1$ and $\ell_2 \mapsto v_2$ have to be satisfied by disjoint parts of the heap. This fact can be written as a separating logic sequent $\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2 \vdash \ell_1 \neq \ell_2$: if a heap

can be subdivided into two disjoint parts containing ℓ_1 and ℓ_2 with appropriate values, then $\ell_1 \neq \ell_2$. Thus, separating conjunction is a crucial connective of separation logic which can be used to talk about pointers that do not alias each other. It has already been suggested in the original papers on separation logic and the logic of bunched implications that a good way of thinking about propositions of separation logic is in terms of *resources*: a proposition P corresponds to some resources (usually, parts of the heap), and validity of P (in the current part of the heap) signifies *ownership* of these resources by a program (or by a thread, in case of a multi-threaded program).

It turned out that the resource interpretation of separation logic, and its capabilities for local reasoning, are well-suited for reasoning about concurrent programs. O'Hearn and Brookes have developed Concurrent Separation Logic [OHe07; Bro07] and proved sound with respect to denotational semantics specifically for this purpose. They observed that in concurrent programs, each thread can manipulate its own local state freely, whereas the specific parts of the state that are accessed by multiple threads are usually accessed within critical regions (ensuring mutual exclusion). CSL has extended separation logic with rules for parallel composition of threads, in which resources joined by separating conjunction can be freely distributed between threads, and a mechanism of transferring ownership of resources through critical regions. A thread that requires a shared resource can obtain it by entering a critical region; the resource is relinquished again once the thread leaves the critical region.

Another important conceptual advancement, in the vein of the resource interpretation, is the idea of using separating conjunction to compose intertwined and seemingly overlapping resources. The simplest example of this is the idea of *fractional permissions* [Boy03; Bor+05]: a location ℓ can be owned by a thread only "partially", with some rational fraction $q \in (0, 1]$, denoted as $\ell \overset{q}{\mapsto} v$. Having such a partial resource $\ell \overset{q}{\mapsto} v$ with $q < 1$, allows the thread to read from a location, but not write to it. Writing to a location requires a resource $\ell \overset{1}{\mapsto} v$, which represents the full ownership of the location ℓ.

Initially, when a new location is allocated, the user obtains a full permission $\ell \overset{1}{\mapsto} v$. Then, this full permission can be split into multiple fractional permissions according to the following rule:

$$\ell \overset{q_1}{\longmapsto} v_1 * \ell \overset{q_2}{\longmapsto} v_2 \dashv\vdash \ell \overset{q_1+q_2}{\longmapsto} v_1 * (v_1 = v_2) * (q_1 + q_2 \leq 1).$$

For example, we can split $\ell \overset{1}{\mapsto} v$ into $\ell \overset{0.5}{\longmapsto} v * \ell \overset{0.5}{\longmapsto} v$. The fractional permissions can then be shared among different threads that read from the location ℓ, but do not write to it.

By indexing the points-to predicate with rational numbers, we allow for a single location to be split unbounded number of times (each $\ell \overset{q}{\mapsto} v$ can be split into $\ell \overset{q/2}{\longmapsto} v * \ell \overset{q/2}{\longmapsto} v$). At the same time, the index q is bounded, as it has to be below or equal to 1. This allows the user to re-combine the fractional permissions into a full permission, which grants complete and exclusive access to the location. To see how this works, consider that when a thread owns a resource $\ell \overset{q}{\mapsto} v$, it must be the case that all the other fractional permissions for the location ℓ add up to $\ell \overset{1-q}{\longmapsto} v$. When $q = 1$, this entails that no other thread can have a fractional permission for accessing

ℓ. Thus, by using fractions from the set $(0,1]$ we ensure not only that we can split a writable permission into an unbounded number of read-only permissions, but also that we can re-combine the read-only resources into a writable permission.

The idea of treating physically overlapping resources as logically disjoint has been dubbed the "fiction of separation" [DGW10]. It has proven to be especially powerful in a concurrent setting in combination with abstraction [Din+10]. The "fictional separation" refers to the fact that even if threads are operating on intertwined pieces of state, we can still view them as operating on logically disjoint resources. This can be illustrated with the following example (from [DGW10]). Suppose we have a module T that implements binary trees using linked lists. Under this implementation, a tree with subtrees t_1 and t_2 is represented in such a way that the nodes of t_1 and t_2 are connected with each other somehow (*e.g.*, a node from t_2 might be reachable from a node of t_1). Despite this, a client of T, as long as it respects the abstraction boundaries of the module, may operate on t_1 and t_2 independently. The client might decide to remove or add nodes in t_1; this might trigger a change in the internal representation of the whole tree (*i.e.*, in case of re-balancing), but from the point of view of the client, those operations do not interfere with the subtree t_2. Thus, separation happens at the abstract level of trees and subtrees; this separation is "fictional" because the underlying representations of abstractly independent subtrees are actually intertwined. By allowing a treatment of intertwined resources as disjoint, fictional separation provides a layer of abstraction useful for compositional reasoning.

While early versions of concurrent separation logics have been formulated for a first-order imperative language with first-order locks and structural parallel composition, subsequent work has focused on extending this approach to cover more concurrent programs, such as those involving storable locks [Got+07], fork/join concurrency [Dod+09], fine-grained concurrency with atomic operations [Vaf08; PBO07], higher-order functions [SBP13], hardware interrupts [Fen+09], *etc.*

Increase in the complexity of the programming language features calls an increase in expressivity of the logics and exploration of new reasoning principles. Modern logics support higher-order quantification and impredicative invariants [SB14; BBT07; SBP13; HAN08; App14; Jun+15; Jun+16] for reasoning about semantically cyclic features, such as higher-order store (*i.e.*, references to references to references ...).

As we have seen, the groundwork laid by CSL has proven to be a fertile ground for research, and the proliferation of (a priori incompatible) ideas in the area has resulted in a vast number of different concurrent separation logics aimed at tackling various kinds of properties and programming language paradigms. There turned out to be a need of unifying various ideas and logics in a common foundational framework that would allow the users to express and encode the same methods and proofs using more principled building blocks. A notable answer to this challenging task (and one of the most important ones in the context of this thesis) was the introduction of the concurrent separation logic Iris [Jun+18b].

Iris presents a unified language-agnostic framework for reasoning in higher-order separation logic using two principled and orthogonal tools—invariants and ghost state [Jun+15; Jun+16]—in order to formulate *protocols* (on shared state), that all threads have to adhere to. Moreover, one of the ideas behind Iris is to provide some essential building blocks, from which more complicated constructions, typical in

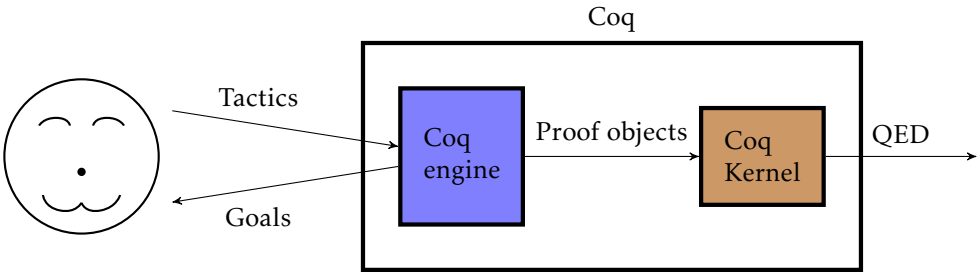


Figure 1.1: Interaction with the Coq proof assistant.

concurrent separation logics, can be recovered. In terms of programming language features, Iris seems to be the state of the art. Due to its impredicative higher-order nature and powerful custom ghost state mechanism, Iris has been used to verify type safety of many type systems (including the type safety and race-freedom of the advanced type system behind the Rust programming language [Jun+18a]), as well as the correctness of fine-grained concurrent algorithms. Due to its generality, extensibility, and expressivity, Iris has proven itself to be a great basis for developing other program logics on top of. In addition, Iris features a flexible and practical Coq formalization (one that enables effective formal reasoning). Due to all of this, we use Iris as starting point and the essential building block for the logics that we present in this thesis.

A historical side-note. For a more detailed history of Concurrent Separation Logic please see a retrospective paper [BO16]. See [Cha20, Section 10] and [OHe19] for surveys of sequential separation logics. For a more detailed history of Iris specifically see [Jun+18b].

1.2 Mechanized reasoning

Proof assistants are software tools that assist the user with the development of formal, machine-checked proofs. All the results in this thesis have been mechanized in the Coq proof assistant [Coq20]. Like many other proof assistants, Coq enables its users to develop formal proofs through an interactive process. A user interacts with Coq by feeding it commands that modify the state of the proof, and looking at the replies from Coq. This process is represented[1] schematically in Figure 1.1, which I have slightly adapted from Geuvers [Geu09].

The way this interaction typically unfolds is as follows. The user begins by formulating a theorem statement. Coq then checks whether the theorem statement is well-formed. If Coq accepts the statement of the theorem, then it provides the user with a *goal* that the user has to prove. The user then issues commands to Coq (in the form of *tactics*), which correspond to formal reasoning steps. For each command, Coq

[1] Two of my fellow students have objected to this representation on the basis that depicting a proof assistant user with a smiley face is inaccurate.

responds to the user with an updated goal. The user then examines the goal, issues new commands, and so on.

In the end, the user finishes the proof (solves the goal) and asks Coq to check whether the proof is correct. Proof checking is delegated to Coq's kernel—a relatively small program that the user trusts to be correct. While processing the user's tactics and updating the goal, under the hood Coq is constructing a *proof object*, which is a compact formal representation of a proof. This proof object is what is being checked at the end by the kernel.

Interactive theorem proving in Coq has seen many applications in verified mathematics and computer science (probably the most well-known proof developments in Coq in the respective areas are a mechanized proof of the odd-order theorem [Gon+13] and a certified C compiler [Ler09]). In particular, Coq has been a popular tool for formalizing programming languages, not least because of its capabilities for specifying inductive definitions for operational semantics and typing derivations. The properties that we investigate in this thesis are formulated in Coq directly in terms of the mechanized operational semantics of a programming language.

All the mechanizations in Coq are written in a formal language based on the dependent type theory called "Calculus of Constructions" (CoC), although in this thesis we will refer to it, somewhat frivolously, as the "Coq logic" or *metalogic*. By itself, Coq does not provide any support for separation logic or any other program logics. Program logics can instead be encoded in the Coq logic and connected to the mechanized operational semantics that way. When we encode a logic in Coq, we refer to it as the *object logic*. These object logics are connected to the properties that we want to show via a soundness statement, which says if some formula over a program e is derivable in the object logic, then e has the desired property. If we obtain a specific mechanized proof in the object logic about a program, then we can compose it with the soundness theorem, obtaining a fully machine-checked proof of a proposition that refers only to the operational semantics. This is something that we deem necessary for a *clear and mechanized soundness theorem*.

Once we have a formally stated and verified soundness theorem, then it is natural to ask: how do we obtain a specific mechanized proof in the object logic, the one that we want to compose with the soundness theorem? In other words, how do we actually use Coq to reason inside the object logic? To resolve this disconnect, the Iris Coq formalization includes a specialized *proof mode* [KTB17; Kre+18] for carrying out proofs inside separation logic in Coq. The Iris proof mode includes tactics for the standard separation logic connectives. In order to "plug in" our logics (which are extensions of Iris) into the proof mode, we further develop custom tactics for the new connectives that we introduce. Thus, using the Iris Coq formalization gives us the benefits of being able to reason formally in our logics using their Coq formalizations, enabling *effective formal reasoning*.

A historical side-note. For a historical overview on proof assistants and mechanized reasoning see [Geu09] and [Rin+19, Section 4].

1.3 Logics introduced in this thesis

In this thesis we introduce three domain-specific separation logics for reasoning about the three properties mentioned in the title: safety, refinement, and security. We develop those program logics to achieve the properties that we stated in the beginning of the introduction. In the following chapters we demonstrate how we achieve these points. Below we briefly explain these properties and the key ideas behind the separation logics that we propose.

1.3.1 Safety

The first property that we consider is *safety*. We say that a program is safe if no execution of it results in some kind of *undefined behavior*: behavior that is not accounted for by the semantics. A common example of undefined behavior is dereferencing a dangling pointer. The C programming languages has many other kinds of undefined behavior, one class of which are *sequence point violations* [ISO12, 6.5p2]. To explain sequence point violations, we need to recall that the order of evaluation of operands in a C expression is unspecified. For example, in the expression

```
e = (e1++) + (e2++);
```

the operands `(e1++)` and `(e2++)` can be evaluated in any order. Furthermore, a compiler can interleave the execution of the operands. Hence, the unspecified evaluation order of operands corresponds to operands being executed *concurrently*, and the executions are synchronized at *sequence points*, *e.g.*, at the end of a full expression (`;`) or before and after a function call.

With this in mind, data races on locations, during the evaluation of the operands, are considered undefined behavior. For example, the following program has undefined behavior due to a sequence point violation:

```
x = (x++) + (x++);
```

Despite the sequence point restrictions, performing side effects in operands is a frequent pattern in C. We can see an example of that in the following program that copies n bytes from `q` to `p`:

```
while (n--) { *(p++) = *(q++); }
```

Note that this program is correct (and makes sense) only if the pointers `p` and `q` are not aliased (do not point to the same piece of memory).

It has already been suggested in [Kre14] that a concurrent separation logic can be used to reason about the unspecified evaluation order (which corresponds to concurrent execution) and to prove the absence of sequence point violations (which corresponds to absence of data races). However, the logic of [Kre14] was not amendable to automation (the rules were not algorithmic), and it was difficult to extend it with new features (such as protocols on shared state).

Based on the ideas in [Kre14], we have developed a program logic λMC (Chapter 3) suitable for proving that a program does not exhibit a sequence point violation or dereferences a dangling pointer. To facilitate practical reasoning about programs in

λMC, we have devised a verification condition generator (VCG), which is integrated into λMC. The VCG can be used to solve goals and to verify subprograms that are amendable to such automated analysis, while still leaving room for interactive proving when necessary. The verification condition generator uses a novel symbolic execution algorithm that symbolically computes a program in a symbolic heap, producing both an updated part of a heap, and a symbolic *frame* – the part of the heap that is untouched by the program. The frame is used to automatically determine how to distribute parts of the symbolic heap for symbolic execution of subexpressions. The symbolic execution algorithm and the VCG itself are defined as executable functions in Coq and are proven correct using the λMC program logic.

The logic of [Kre14] and its soundness were given via an ad-hoc model, which limited its expressiveness and made it hard to extend the logic. Our program logic has been developed on top of Iris, and thereby inherits all advanced features of Iris (like its expressive support for ghost state and invariants), without having to model these explicitly. We have constructed λMC by composing a program logic for a monadic fragment of an ML-like language, with a novel definitional translation of a fragment of C into the mentioned monadic fragment.

The logic λMC is named like this because our definitional translation targets a functional language (λ) and uses monadic combinators (M).

1.3.2 Refinement

The second property that we consider is *refinement* between two programs, specifically *contextual refinement*. Contextual refinement tells us when observable behavior of one program is included in the observable behavior of another program, and it is one half of contextual equivalence—the "golden standard" for the notion of program equivalence. A program e_1 *contextually refines* a program e_2 (denoted as $e_1 \precsim_{ctx} e_2$), when e_2 can be substituted for e_1 in a larger program (referred to as a *context*) without changing the overall observable behavior.

Contextual refinement has several practical applications. For example, it is used for showing that an optimized version of a data structure refines a naive implementation, or for showing that certain algebraic laws about program semantics hold. In the setting of concurrent programs, contextual refinement is useful for establishing *linearizability* [HW90; Fil+10], which is a commonly established correctness criterion for concurrent programs. Roughly, a module is linearizable if it behaves, according to an arbitrary client, as if it was sequential. To establish that a module is linearizable it suffices to establish that the module refines a version of itself, where all the operations have been protected by a lock.

Proving contextual refinements directly is tricky, because in order to establish contextual refinement we have to consider program executions in an arbitrary context. Additional complications may arise from specific programming language features like concurrency and mutable state. For example, consider the program (written here in an ML-like language)

$$\mathsf{let}\ x = f\ ()\ \mathsf{in}\ (x, x)$$

that computes $f\ ()$ and returns a tuple with the result of the computation. In a pure setting, this program would refine (and vice versa) the program

$$(f\ (), f\ ())$$

computing $f\ ()$ twice. However, in presence of effects like mutable state this refinement does not hold, because there is no guarantee that invoking f twice will produce the same result. As another example, consider the program.

$$\mathsf{let}\ x = \mathsf{ref}(0)\ \mathsf{in}\ x \leftarrow 10; !x$$

that allocates a new mutable reference x, assigns 10 to it, and then dereferences it. This program refines a simple program that just returns the number 10:

$$(\mathsf{let}\ x = \mathsf{ref}(0)\ \mathsf{in}\ x \leftarrow 10; !x) \precsim_{ctx} 10.$$

This is because the reference x is *local* to the left-hand side, and no context can access it. However, if we are in a concurrent setting and we consider a version where x is free, then the following refinement no longer holds:

$$x \leftarrow 10; !x \precsim_{ctx} 10.$$

It does not hold because the program on the left-hand side can be placed in a context where another thread assigns some value to x.

To make reasoning about contextual refinements more tractable, researchers have formulated models for reasoning about contextual refinements in a more local way. One such class of models is based on *logical relations* [PA93; DAB09; Dre+10; Tur+13; TDB13], allowing to reduce contextual refinement to a more local notion of *logical refinement*. Recently, Krebbers, Timany, and Birkedal [KTB17] and Timany [Tim18] have established a logical relations model, using Iris, for an ML-like programming language with references and concurrency. This model allows for mechanized verification of contextual refinements for concurrent programs, and it has been used to established type safety of System F, extended with references, concurrency, and recursive types. However, proofs of logical refinements in this development involve unfolding the definition of logical refinement and reasoning directly in the model.

Building up on their work, we have developed a relational logic ReLoC (Chapter 4) for proving logical refinements between concurrent programs. In ReLoC, we abstract from the model of [KTB17], and provide a full-fledged relational logic. The core of ReLoC is the *refinement judgment* $e \precsim e' : \tau$, which expresses that a program e logically refines a program e' at type τ. To reason about the refinement judgment, ReLoC includes symbolic execution rules and type-directed structural rules. The refinement judgments in ReLoC are first-class (*i.e.*, can be combined with the logical connectives like any other proposition), which allows us to provide a template for *relational specifications* of programs. Relational specifications generalize standard Hoare logic-style specifications in unary program logics: they abstract away from the implementation of the program and express its behavior in terms of pre- and postconditions.

Furthermore, we show how to provide *logically atomic relational specifications* for compound programs that are not atomic, but only appear to take effect at a single instant in time, *e.g.*, because the modify the global state in a consistent, atomic way. With logically atomic relational specifications, we generalize the ideas of logical atomicity from the logics HOCAP [SBP13] and TaDA [RDG14] to a relational setting.

We provide a mechanization of ReLoC in Coq, complete with the tactics for reasoning inside the program logic. This allows us to mechanize a lot of example refinements, some of which have not been mechanized before (partially due to the complexity of formal reasoning).

The name ReLoC stands for "Relational Logic for Concurrency".

1.3.3 Security

The final property that we consider in this thesis is security. A program is deemed secure if it does not have any *information leaks*, *i.e.*, if secret information is not revealed to an attacker observing the program. In programming language research the typical condition for enforcing security is *non-interference*: it says that changing secret information in the program does not lead to observably different behavior.

A program can violate non-interference in several different ways. A program can directly leak secret information from a secrete reference h to a publicly-observable reference l:

$$l \leftarrow !\,h.$$

However, information leaks can be indirect, as in the following example:

$$\mathsf{if}\ (!\,h > 0)\ \mathsf{then}\ l \leftarrow 0\ \mathsf{else}\ l \leftarrow 1.$$

The program leaks the information by branching on the value of h, and assigning a different value to l based on this information. By contrast, the following program is secure, because the two branches are indistinguishable:

$$\mathsf{if}\ (!\,h > 0)\ \mathsf{then}\ l \leftarrow 0\ \mathsf{else}\ l \leftarrow 0.$$

Additional sources of information leaks can come from concurrent interleavings. For example, in the program

$$l \leftarrow !\,h;\ l \leftarrow 0$$

the value in h may be leaked, even though the reference l gets overwritten. Depending on whether the location l is shared, another thread may observe the leaked value of h, prior to it being overwritten by 0. The scheduler presents another potential source of information leaks; for example the program below leaks the value of h if run under a uniform scheduler:

$$(l \leftarrow \mathbf{true}) \,\|\, (l \leftarrow \mathbf{false}) \,\|\, (l \leftarrow !\,h).$$

In order to reason about non-interference of concurrent programs we have developed a relational logic SeLoC (Chapter 5). SeLoC is sound with respect to one of the strongest notions of non-interference for concurrent programs: strong low-bisimulations[2] introduced by Sabelfeld and Sands [SS00]. To our knowledge, SeLoC is

[2] More specifically, a flow-sensitive version of it.

the first program logic capable of foundational reasoning about such non-interference criterion for fine-grained concurrent programs.

Using SeLoC as the base, we build a type system that tracks information flow in the program. Type systems and type system-like logics have received a lot of attention in the context of non-interference [PS03; MSS11; MSE18; EM19; Kar+18]. Such systems have the advantage of being highly compositional and admitting strong automation (via type checking and type inference). However, conventional type systems lack capabilities to reason about functional correctness of programs. Thus, conventional type systems cannot be used to verify non-interference of programs whose secrecy depends on run-time behavior. For example, in case of *value-dependent classification* [ZM07; Mur+16; NBG13; LC15; GTA19], a classification of a variable (whether it contains secret or publicly-observable data) depends on a run-time flag.

In SeLoC, we take an approach that combines the program logic and the type system. Specifically, we provide an interpretation of the type system in SeLoC and show that it is sound: if a program is well-typed, then it is secure. Program modules that are well-typed can be combined with program modules that are not well-typed, but nonetheless verified in the program logic. The resulting combined program satisfies non-interference.

The name SeLoC stands for "Security Logic for Concurrency".

1.4 Contributions and outline

This thesis contains a chapter with preliminaries, and three chapters that constitute the main research content of the thesis. In Chapter 2 we recall the separation logic preliminaries and formally define the semantics of the programming language that we use throughout this thesis. The material presented in this chapter is not original. The main chapters Chapters 3 to 5 introduce the original contributions of this thesis, namely the three logics for verifying the three properties outlined in the introduction. Each of these three chapters contains its own introduction and related work section, and each of the three chapters can be read in isolation. Below we give an overview of the main chapters and their contributions.

Chapter 3: λMC: a logic for non-determinsim in C expressions. In this chapter we describe λMC: a semantics-by-translation for a subset of C and a corresponding program logic for proving the absence of sequence point violations. The program logic is based on the program logic described in [Kre14], however our logic is more expressive and more amendable to automated verification. To demonstrate that, we develop a verification condition generator that integrates automated verification with interactive proving in λMC. This chapter is based on the publication

- Dan Frumin, Léon Gondelman, Robbert Krebbers. *"Semi-Automated Reasoning About Non-Determinism in C"* [FGK19a]. Presented at the 28th European Symposium on Programming (ESOP), 2019.

The main contributions of this paper are:

- A novel semantics-by-translation of a fragment of C, that takes into account unspecified evaluation order of subexpressions and undefined behavior due to sequence points violations;
- A separation logic with a weakest precondition calculus, based on the logic from [Kre14], but developed in Iris instead of a custom model;
- A symbolic execution algorithm that takes as input a program and a symbolic heap, and calculates a symbolic postcondition and a symbolic frame.
- A mechanized verification condition generator (VCG), that is defined on top of the symbolic execution algorithm, and that is verified w.r.t. the semantics of λMC.
- A mechanization of the logic and integration of the VCG, which we demonstrate on a number of examples, including a version of a `memcpy` function.

The notations for various logical connectives and the names of rules in Chapter 3 have been changed to be compatible with the rest of the thesis. Section 3.3 contains an additional example with comments.

Chapter 4: ReLoC: a logic for proving contextual refinements. In this chapter we describe ReLoC: a logic for proving refinements of concurrent programs. We provide a way of giving relational specifications in ReLoC for programs, which allows for modular verification of refinements of involved algorithms and data structures. In contrast to earlier work on refinements for languages with higher-order state and concurrency (notably, [KTB17; Tim18]), ReLoC provides type-directed structural rules and symbolic execution rules for manipulating refinement judgments, whereas previously, such proofs were carried out by unfolding the refinement judgment into its definition in the model. This chapter is based on the publications

- Dan Frumin, Robbert Krebbers, Lars Birkedal. *"ReLoC: A Mechanized Relational Logic for Fine-Grained Concurrency"* [FKB18]. Presented at the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), 2018.
- Dan Frumin, Robbert Krebbers, Lars Birkedal. *"ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity"* [FKB20b]. An extended and updated version of the LICS 2018 paper, currently under submission.

The main contributions of these papers are:

- A novel relational logic for reasoning about refinements of fine-grained higher-order concurrent programs;
- Novel ways of writing modular relational specifications for concurrent libraries, based on atomic triples of TaDA [RDG14] and HOCAP-style [SBP13] specifications;
- A mechanization of ReLoC that supports interactive and practical reasoning inside the ReLoC. We demonstrate the practicality of ReLoC on a number of case studies, including verification of concurrent stacks, a ticket-based lock, a name generation ADT, and algebraic laws for non-deterministic choice and parallel and sequential composition operators.

Chapter 5: SeLoC: a logic for proving non-interference. In this chapter we describe SeLoC: a logic for proving non-interference of concurrent programs. SeLoC is a relational logic that captures the notion of *strong-low bisimulation* of concurrent programs. Using the method of *semantic typing* [Dre+19], we implement an information-flow aware type system on top of SeLoC. By using semantic typing we can compositionally verify programs that consist of both typeable and untypeable parts. The former parts can be verified with a type checker and the latter parts can be proved manually using the logic. This chapter is based on the publication

- Dan Frumin, Robbert Krebbers, Lars Birkedal. *"Compositional Non-Interference for Fine-Grained Concurrent Programs"* [FKB21b]. To appear at the 42nd IEEE Symposium on Security and Privacy (S&P), 2021.

The main contributions of this paper are:

- A novel relational logic for proving non-interference of fine-grained concurrent programs;
- A relational version of weakest precondition and a soundness proof that involves a bisimulation construction out of the closed proofs in SeLoC;
- A type system for information flow analysis, that is built on top of the SeLoC and proven sound using logical relations;
- A mechanization of the logic and the type system, as well as a number of case studies. The case studies include a secure implementation of a set data structure, and an implementation of references with value-dependent classifications.

The material in Chapter 5 that was previously present in the appendix of the corresponding paper [FKB21b] has been expanded and moved into the main body of the chapter. In particular, Section 5.7, Section 5.5 have been expanded and contain material that was previously in the appendix of [FKB21b]; Section 5.9 was previously only present in the appendix of [FKB21b].

Statement of contributions. As has been mentioned, the main chapters of this thesis are based on several peer-reviewed publications, and are copied almost verbatim, with the format and the typography adapted to fit the presentation of the thesis. Those publications were written with co-authors, so I should clarify my exact involvement.

For the publications pertaining to Chapters 4 and 5, I am the main author, and my co-authors served the supervisorial roles. I carried out the main research, under the supervision of my co-authors and took the lead in writing the papers.

For the publication leading to Chapter 3, both me and my colleague Léon Gondelman were the main authors, and Robbert Krebbers was a supervisor to the project. Together with Léon, we have actively worked on all the parts in direct collaboration. Léon took the lead in developing the VCG and the reification/reflection process, and I took the lead in developing the logic and the symbolic executor.

Chapter 2 is heavily based on the background sections of the published papers, but amalgamated together and expanded for the sake of presentation.

Coq sources. All the material presented in this thesis has been mechanized in the Coq proof assistant. The Coq sources are available in the following Git repository under the BSD license:

https://github.com/co-dan/thesis

A "frozen" version of is available under a persistent DOI: 10.5281/zenodo.4445839. See also the appendix on "Research Data Management" for more details on the associated Coq source code.

2 Background on separation logic

In this chapter we introduce some preliminary materials on separation logic in general and Iris specifically. As has been mentioned in the introduction, all of the program logics presented in this thesis take the concurrent separation logic Iris as a foundation on top which they are built.

Iris is a state-of-the-art concurrent separation logic framework with a flexible mechanism for specifying protocols on shared state and a Coq mechanization allowing one to write formal proofs in an easy manner. Iris, in all its generality, is *programming language independent*, and can be instantiated with an arbitrary language (for example, Rust [Jun+18a] or Scala's core calculus DOT [Gia+20]). But by default Iris comes equipped with a functional ML-like programming language HeapLang, which supports general recursion, higher-order functions, dynamic references and fork-based concurrency. It is an expressive programming language that we will use throughout this thesis.

We begin this chapter with Section 2.1, in which we formally describe the syntax and operational semantics of HeapLang. In Section 2.2 we describe the program logic over HeapLang, focusing on sequential programs. To verify concurrent programs with shared state, Iris includes a mechanism of invariants, which we describe in Section 2.3. Invariants are used for reasoning about resources shared between different threads. In order to further impose protocols on the way the resources are shared, invariants are combined with *custom ghost state* theories, of which we give an overview in Section 2.4. In Section 2.5 we briefly describe the Iris Coq formalization that we use throughout this thesis. We finish the chapter with an overview of several different ways to define custom separation logics on top of Iris in Section 2.6.

This chapter should serves as a basis which makes it easier to understand the rest of the thesis. An interested reader is referred to the overview paper [Jun+18b] and the lecture notes [BB20] to learn more about Iris.

2.1 Syntax and semantics of HeapLang

In this section we formally define the semantics of the programming language that we will use throughout this thesis. This language is called HeapLang, and it is the default language that comes with the Iris framework formalization in Coq. The syntax of

$$
\begin{array}{rll}
v \in \mathit{Val} ::= & i \mid \ell \mid () & \text{(Integers } i \in \mathbb{Z}\text{, locations } \ell \in \mathit{Loc}\text{, unit value)} \\
& \mid \mathsf{true} \mid \mathsf{false} & \text{(Booleans)} \\
& \mid (v_1, v_2) & \text{(Pairs)} \\
& \mid \mathsf{inl}\ (v) \mid \mathsf{inr}\ (v) & \text{(Sums)} \\
& \mid \mathsf{rec}\ f\ x = e & \text{(Recursive } \lambda\text{-functions)} \\
\circledast \in \mathit{BinOp} ::= & * \mid + \mid - \mid \ldots & \text{(Binary operations)} \\
e \in \mathit{Expr} ::= & x \mid v \mid e_1\ e_2 & \text{(Variables, values and function application)} \\
& \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \text{(Booleans)} \\
& \mid (e_1, e_2) \mid \pi_1(e) \mid \pi_2(e) & \text{(Pairs)} \\
& \mid \mathsf{inl}\ (e) \mid \mathsf{inr}\ (e) & \text{(Sums)} \\
& \mid (\mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}\ (x) \to e_1 \mid \mathsf{inr}\ (x) \to e_2) & \\
& \mid e_1 \circledast e_2 & \text{(Binary operations)} \\
& \mid \mathsf{ref}(e) \mid\ !e \mid e_1 \leftarrow e_2 & \text{(Heap operations)} \\
& \mid \mathsf{CAS}(e_1, e_2, e_3) \mid \mathsf{fork}\ \{e\} & \text{(Concurrency and atomic primitives)}
\end{array}
$$

Figure 2.1: The syntax of the HeapLang language.

HeapLang is shown in Figure 2.1. HeapLang is untyped, although we consider type systems for it in Chapters 4 and 5.

The standard operations on references are $\mathsf{ref}(e)$ for allocation, $!e$ for dereferencing, and $e_1 \leftarrow e_2$ for assignment. The *atomic* compare-and-set operation $\mathsf{CAS}(e_1, e_2, e_3)$ checks if the value stored at the location e_1 is equal to e_2, and, if so, sets the value at e_1 to e_3. The $\mathsf{fork}\ \{e\}$ construct creates a new thread, which will execute the expression e. The are no built-in synchronization constructs. Rather, constructs like locks are defined using $\mathsf{CAS}(-,-,-)$ (as we will see in Section 2.4.2). Likewise, parallel composition is defined using $\mathsf{fork}\ \{-\}$ (Section 2.4.1).

The construct $\mathsf{rec}\ f\ x = e$ is a recursive λ-function, whose body e can refer to the function f itself and the argument x. As usual, function application associates to the left: $e_1\ e_2\ e_3$ is parsed as $(e_1\ e_2)\ e_3$.

Arrays in HeapLang are omitted in the thesis, but they are used in the Coq mechanization.

Syntactic sugar. We use syntactic sugar to define non-recursive functions, let-bindings, and sequential composition. We let $(\lambda x.\ e) \triangleq (\mathsf{rec}\ \_\ x = e)$ and $(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \triangleq ((\lambda x.\ e_2)\ e_1)$ and $(e_1; e_2) \triangleq (\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2)$. The underscore _ denotes an anonymous binder, *i.e.*, a fresh variable that does not appear in the body of the binding expression. We model option types with sum types. Thus, we write None for $\mathsf{inl}\ (())$ and $\mathsf{Some}(e)$ for $\mathsf{inr}\ (e)$, and we similarly overload the $(\mathsf{match}\ e\ \mathsf{with}\ \mathsf{None} \to e_1 \mid \mathsf{Some}(x) \to e_2)$ construction.

Pure reductions: $e_1 \to_{\mathsf{pure}} e_2$

$$(\mathsf{rec}\ f\ x = e)\ v \to_{\mathsf{pure}} e[v/x][\mathsf{rec}\ f\ x = e/f] \qquad \mathsf{if}\ \mathbf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to_{\mathsf{pure}} e_1$$

$$\mathsf{if}\ \mathbf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \to_{\mathsf{pure}} e_2 \qquad \pi_i\ (v_1, v_2) \to_{\mathsf{pure}} v_i$$

$$\begin{pmatrix}\mathsf{match}\ \mathbf{inl}\ (v)\ \mathsf{with}\\ \mathbf{inl}\ (x) \to e_1\\ |\ \mathbf{inr}\ (x) \to e_2\end{pmatrix} \to_{\mathsf{pure}} e_1[v/x] \qquad \begin{pmatrix}\mathsf{match}\ \mathbf{inr}\ (v)\ \mathsf{with}\\ \mathbf{inl}\ (x) \to e_1\\ |\ \mathbf{inr}\ (x) \to e_2\end{pmatrix} \to_{\mathsf{pure}} e_2[v/x]$$

$$\frac{\circledast \in \{*, +, -, \ldots\} \qquad i_1, i_2 \in \mathbb{Z} \qquad i_3 = i_1 \circledast i_2}{i_1 \circledast i_2 \to_{\mathsf{pure}} i_3}$$

Thread-local call-by-value head-reduction $(e, \sigma) \to_{\mathsf{h}} (e', \sigma')$:

$$\frac{e_1 \to_{\mathsf{pure}} e_2}{(e_1, \sigma) \to_{\mathsf{h}} (e_2, \sigma)} \qquad \frac{\sigma(\ell) = \bot}{(\mathsf{ref}(v), \sigma) \to_{\mathsf{h}} (\ell, \sigma[\ell \leftarrow v])} \qquad \frac{\sigma(\ell) = v}{(!\ell, \sigma) \to_{\mathsf{h}} (v, \sigma)}$$

$$\frac{\sigma(\ell) = v}{(\ell \leftarrow v', \sigma) \to_{\mathsf{h}} ((), \sigma[\ell \leftarrow v'])} \qquad \frac{\sigma(\ell) \neq v_1}{(\mathsf{CAS}(\ell, v_1, v_2), \sigma) \to_{\mathsf{h}} (\mathbf{false}, \sigma)}$$

$$\frac{\sigma(\ell) = v_1}{(\mathsf{CAS}(\ell, v_1, v_2), \sigma) \to_{\mathsf{h}} (\mathbf{true}, \sigma[\ell \leftarrow v_2])}$$

Thread-local reduction $(e, \sigma) \to_{\mathsf{t}} (\vec{e'}, \sigma')$:

$$\frac{(e, \sigma) \to_{\mathsf{h}} (e', \sigma')}{(K[\,e\,], \sigma) \to_{\mathsf{t}} (K[\,e'\,], \sigma')} \qquad (K[\,\mathsf{fork}\ \{e\}\,], \sigma) \to_{\mathsf{t}} (K[\,()\,]\ e, \sigma)$$

Thread-pool reduction $(\vec{e}, \sigma) \to_{\mathsf{tp}} (\vec{e'}, \sigma')$:

$$\frac{(e, \sigma) \to_{\mathsf{t}} (\vec{e'}, \sigma')}{(\vec{e_1}\ e\ \vec{e_2}, \sigma) \to_{\mathsf{tp}} (\vec{e_1}\ \vec{e'}\ \vec{e_2}, \sigma')}$$

Figure 2.2: The operational semantics of HeapLang.

Whenever we want to define a recursive function, we often abuse the notation and write

$$\mathsf{f}\ x_1\ \ldots\ x_n = e \qquad \text{for a definition} \qquad \mathsf{f} \triangleq (\mathsf{rec}\ \mathsf{f}\ x_1\ \ldots\ x_n = e).$$

Operational semantics. The operational semantics of HeapLang is call-by-value and it involves several reduction relations: head reductions $\to_{\mathsf{h}}$, thread-local reductions $\to_{\mathsf{t}}$ and thread-pool reductions $\to_{\mathsf{tp}}$. The reductions are defined in Figure 2.2.

Head reductions $(e,\sigma)\to_{\mathsf{h}}(e',\sigma')$ are defined on pairs of an expression e and a *state* σ. A state is a finite partial map from locations to values $Loc \xrightarrow{\text{fin}} Val$. Among the head reductions, some are *pure* reductions $\to_{\mathsf{pure}}$, *i.e.*, they are deterministic and do not modify the state.

Head reductions $\to_{\mathsf{h}}$ are lifted to thread-local reductions $(e,\sigma)\to_{\mathsf{t}}(\vec{e'},\sigma')$ using standard *call-by-value right-to-left evaluation contexts* (in the style of Felleisen and Hieb [FH92]):

$$\begin{aligned} K \in ECtx ::= {} & [\bullet] \mid e_1\ K \mid K\ v_2 \mid e_1 \circledast K \mid K \circledast v_2 \mid \mathsf{if}\ K\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\ & \mid (e_1, K) \mid (K, v_2) \mid \pi_i(K) \\ & \mid \mathsf{inl}\ (K) \mid \mathsf{inr}\ (K) \mid (\mathsf{match}\ K\ \mathsf{with}\ \mathsf{inl}\ (x) \to e_1 \mid \mathsf{inr}\ (x) \to e_2) \\ & \mid \mathsf{ref}(K) \mid\ !K \mid e_1 \leftarrow K \mid K \leftarrow v_2 \\ & \mid \mathsf{CAS}(e_1, e_2, K) \mid \mathsf{CAS}(e_1, K, v_3) \mid \mathsf{CAS}(K, v_2, v_3) \end{aligned}$$

In addition, the thread-local reduction handles the $\mathsf{fork}\ \{-\}$ operation. That is why the second component of the thread-local reduction contains a list $\vec{e'}$ of expressions: it consists of all the additional forked-off threads. Since the only way of creating a new thread in HeapLang is the $\mathsf{fork}\ \{-\}$ operation, the second component thread-local reduction contains either a single expression or two expressions.

Thread-local reductions are in turn lifted to thread-pool reductions $(\vec{e},\sigma)\to_{\mathsf{tp}}(\vec{e'},\sigma')$. Thread-pool reductions are defined on configurations $\rho = (\vec{e},\sigma)$ consisting of a thread-pool $\vec{e}$ (a list of expressions corresponding to the threads) and a state σ. Thread-pool reductions are defined by the interleaving semantics, *i.e.*, by picking a thread from the thread-pool and executing it, thread-locally, for one step.

We say than an expression e is *reducible* in state σ, if $(e,\sigma)\to_{\mathsf{t}}(\vec{e'},\sigma')$ for some configuration $(\vec{e'},\sigma')$.

2.2 Basics of Iris

In this section we introduce the basics of Iris, mainly through its fragment used for reasoning about sequential programs.

Syntax of Iris. The grammar of Iris propositions is as follows:

$$\begin{aligned} A, B \in \mathit{Type} &::= 0 \mid 1 \mid \mathbb{N} \mid \mathit{Val} \mid \mathit{Expr} \mid \mathit{Loc} \mid \mathit{iProp} \mid A + B \mid A \times B \mid A \to B \mid \ldots \\ P, Q \in \mathit{iProp} &::= \mathsf{True} \mid \mathsf{False} \mid P \implies Q && \text{(Intuitionistic logic)} \\ &\quad \mid \forall x : A.\, P \mid \exists x : A.\, P \mid P \land Q \mid P \lor Q \mid \ldots \\ &\quad \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v \mid \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} && \text{(Separation logic)} \\ &\quad \mid \boxed{P}^{\mathcal{N}} \mid \triangleright P \mid \Box P \mid {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \mid \ldots && \text{(Iris-specific connectives)} \end{aligned}$$

Iris contains the usual connectives of intuitionistic higher-order logic like quantifiers $\forall x.\, P$ (we usually omit the type annotations on quantifiers), conjunction $P \land Q$, and so on. It also features the familiar separation logic connectives like separating conjunction $P * Q$, magic wand $P \mathrel{-\!\!*} Q$, and the points-to predicate $\ell \mapsto v$ (where $\ell \in \mathit{Loc}$ is a HeapLang location and $v \in \mathit{Val}$ is a HeapLang value). We write $\ell \mapsto -$ for $(\exists v.\, \ell \mapsto v)$.

Iris also contains the *later* modality $\triangleright$, the *persistence* modality $\Box$, the *update* modality ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$, and the *invariant assertion* $\boxed{P}^{\mathcal{N}}$. We introduce these connectives in passing throughout this section. Some of these connectives are annotated by *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$ and *invariant names* $\mathcal{N} \in \mathit{InvName}$, which are needed for bookkeeping related to Iris's invariant mechanism. We will cover them more closely when we discuss the invariant mechanism in Iris.

As a program logic, Iris features the *weakest precondition connective* $\mathsf{wp}\ e\ \{\Phi\}$, which states that reducing the expression e is safe, and whenever e reduces to some value v, the value satisfies the postcondition $\Phi : \mathit{Val} \to \mathit{iProp}$ (see Theorem 2.1 for the soundness statement of wp). We write $\mathsf{wp}\ e\ \{v.\, \Phi(v)\}$ for $\mathsf{wp}\ e\ \{\lambda v.\, \Phi(v)\}$. The weakest precondition connective does not represent the preconditions that one find in Hoare triples. Instead, these preconditions are encoded using the magic wand. For example, the proposition

$$(\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2) \quad \mathrel{-\!\!*} \quad \big(\mathsf{wp}\ (\ell_1 \leftarrow w_1; !\ell_2)\ \{v.\, v = v_2\}\big), \tag{2.1}$$

which can also be written without the superfluous parentheses as

$$\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2 \mathrel{-\!\!*} \mathsf{wp}\ (\ell_1 \leftarrow w_1; !\ell_2)\ \{v.\, v = v_2\},$$

states that under the precondition $\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2$ it is safe to execute the program $\ell_1 \leftarrow w_1; !\ell_2$, and after the execution the program returns v_2.

Inference rules and derivability. As standard in logic, Iris has a derivability relation $P \vdash Q$. Figure 2.3 presents a selection of Iris rules pertaining to the intuitionistic logic connectives and separation logic connectives. The rules for other connectives will be presented throughout this chapter.

We say that Q is *derivable* if $\mathsf{True} \vdash Q$. In many situations, we use magic wand $\mathrel{-\!\!*}$ instead of the derivability relation $\vdash$, because Iris has the standard deduction property:

$$P \vdash Q \mathrel{-\!\!*} R \qquad \text{iff} \qquad P * Q \vdash R$$

Intuitionistic logic:

$$\frac{P}{P} \qquad \frac{P \vdash Q \qquad Q \vdash R}{P \vdash R} \qquad \frac{P}{\mathsf{True}} \qquad \frac{\mathsf{False}}{P} \qquad \frac{P \wedge Q}{P} \qquad \frac{P \wedge Q}{Q} \qquad \frac{P \vdash Q \qquad P \vdash R}{P \vdash Q \wedge R}$$

$$\frac{P}{P \vee Q} \qquad \frac{Q}{P \vee Q} \qquad \frac{P \vdash R \qquad Q \vdash R}{P \vee Q \vdash R} \qquad \frac{P \wedge Q \vdash R}{P \vdash Q \implies R}$$

Separation logic:

$$\frac{P * (Q * R)}{(P * Q) * R} \qquad \frac{P * Q}{Q * P} \qquad \frac{P}{\mathsf{True} * P} \qquad \frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \qquad \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

Figure 2.3: Selected rules of Iris.

Most of the inference rule we present can be internalized as Iris propositions by a magic wand or a derivability relation between the separating conjunction of the antecedents and the consequent. We thus use the following notations:

$$\frac{P_1 \qquad \cdots \qquad P_n}{Q} \quad \text{is notation for} \quad (P_1 * \cdots * P_n) \mathrel{-\!\!*} Q,$$

$$\frac{P}{Q} \quad \text{is notation for} \quad (P \mathrel{-\!\!*} Q) \wedge (Q \mathrel{-\!\!*} P).$$

For instance, the proposition in Equation (2.1) is presented as the following inference rule:

$$\frac{\ell_1 \mapsto v_1 \qquad \ell_2 \mapsto v_2}{\mathsf{wp}\,(\ell_1 \leftarrow w_1; !\ell_2)\,\{v.\, v = v_2\}}$$

We use the derivability relation $\vdash$ to explicitly state the rules that cannot be internalized, *e.g.*, $\dfrac{\vdash P}{\vdash Q}$ states that if P is derivable, then Q is derivable. This is weaker than $\dfrac{P}{Q}$, which denotes that P can be derived from Q, *i.e.*, $P \vdash Q$.

We would also like to note that Iris is an *intuitionistic separation logic* (also often called *affine separation logic*) in the sense of [Rey02]. That is, Iris admits $P * Q \vdash P$, and the predicate $\ell \mapsto v$ holds in all heaps that contain at least the location ℓ with the value v.

Coq logic and Iris. The full formal grammar of Iris is defined in a way that is usual for typed higher-order logics parameterized over some signature. That is, the logic includes terms of various types, equality predicates, and all the standard rules of intuitionistic higher-order logic.

In practice (*i.e.*, what happens in the Coq formalization) we view Iris as a superset of Coq logic. Every Coq proposition is a proposition of Iris, and all the common Coq types such as lists, trees, and other inductive types are types of Iris.[1] We write $\ulcorner \phi \urcorner : \mathit{iProp}$ for an embedding of the Coq proposition $\phi : \texttt{Prop}$ into Iris. The embedding function $\ulcorner - \urcorner$ commutes with the intuitionistic connectives. On paper we omit the embedding for common connectives like equality predicates, set membership, and so on.

Soundness of Iris. Iris admits the following soundness theorem (stated and proved in [Jun+18b, Theorem 6]), that connects derivability in Iris with operational semantics of HeapLang:

Theorem 2.1. Let ϕ a Coq predicate over the type *Val* of HeapLang values. Suppose that $\mathsf{wp}\, e\, \{v.\, \ulcorner \phi(v) \urcorner\}$ is derivable in Iris, and $(e, \sigma) \to^*_{\mathsf{tp}} (e'_1 e'_2 \ldots e'_n, \sigma')$. Then:

1. For any e'_i (with $1 \le i \le n$), either e'_i is a value or e'_i is reducible in σ'.
2. If e'_1 is a value, then $\phi(e'_1)$ holds.

There are other stronger variations of the soundness theorem in the Iris Coq formalization and the accompanying documentation [Iri20], but the statement above is the simplest one, and it will serve as the basis for the soundness statements of the logics presented in the upcoming chapters of this thesis.

2.2.1 Weakest precondition calculus

The rules governing the $\mathsf{wp}\, e\, \{\Phi\}$ connective are presented in Figure 2.4 (the later modality $\triangleright$ and the mask annotations $\mathcal{E}$ can be ignored for now and will be explained in Sections 2.2.3 and 2.3). Most of the rules in that figure are *symbolic execution* rules in the sense that they allow us to symbolically execute an expression e in a symbolic heap which is described through the separation logic connectives. For example, consider the rule WP-STORE; ignoring the $\triangleright$ modality and the mask $\mathcal{E}$ (which will be explained in Sections 2.2.3 and 2.3), it can be written as:

$$\frac{\ell \mapsto v \qquad \ell \mapsto w \mathrel{-\!\!*} \Phi(())}{\mathsf{wp}\, \ell \leftarrow w\, \{\Phi\}}$$

It says that in order to symbolically execute $\ell \leftarrow w$, we need to provide a points-to connective $\ell \mapsto v$ saying that the location ℓ exists and we have the permission to write to it. The expression $\ell \leftarrow w$ evaluates to $()$, hence at the end we need to prove $\Phi(())$. Recall that separation logic is substructural: when a user of the rule provides $\ell \mapsto v$ as a precondition, they "give it up" and no longer have access to it. Instead, for proving $\Phi(())$ the user gets a new assumption $\ell \mapsto w$.

[1] Formally, treating a Coq type as an Iris type requires one to prove that the type has a structure of an ordered family of equivalences (OFE). This technicality is due to Iris's step-indexed model, which is out of the scope for this thesis.

$$\textsc{wp-val}\ \frac{\Phi(v)}{\mathsf{wp}_{\mathcal{E}}\, v\, \{\Phi\}} \qquad \textsc{wp-bind}\ \frac{\mathsf{wp}_{\mathcal{E}}\, e\, \left\{v.\ \mathsf{wp}_{\mathcal{E}}\, K[\,v\,]\, \{\Phi\}\right\}}{\mathsf{wp}_{\mathcal{E}}\, K[\,e\,]\, \{\Phi\}} \qquad \textsc{wp-wand}\ \frac{\mathsf{wp}_{\mathcal{E}}\, e\, \{\Psi\} \qquad (\forall v.\ \Psi(v) \mathrel{-\!\!*} \Phi(v))}{\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

$$\textsc{wp-pure}\ \frac{e \to_{\mathsf{pure}} e' \qquad \triangleright\, \mathsf{wp}_{\mathcal{E}}\, e'\, \{\Phi\}}{\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}} \qquad \textsc{wp-alloc}\ \frac{\forall \ell \in \mathit{Loc}.\ \ell \mapsto v \mathrel{-\!\!*} \triangleright\, \Phi(\ell)}{\mathsf{wp}_{\mathcal{E}}\, \mathsf{ref}(v)\, \{\Phi\}}$$

$$\textsc{wp-load}\ \frac{\triangleright\, \ell \overset{q}{\mapsto} v \qquad \triangleright(\ell \overset{q}{\mapsto} v \mathrel{-\!\!*} \Phi(v))}{\mathsf{wp}_{\mathcal{E}}\, !\ell\, \{\Phi\}} \qquad \textsc{wp-store}\ \frac{\triangleright\, \ell \mapsto v \qquad \triangleright(\ell \mapsto w \mathrel{-\!\!*} \Phi(()))}{\mathsf{wp}_{\mathcal{E}}\, \ell \leftarrow w\, \{\Phi\}}$$

$$\textsc{wp-cas-fail}\ \frac{\triangleright\, \ell \mapsto w \qquad w \neq v_1 \qquad \triangleright(\ell \mapsto w \mathrel{-\!\!*} \Phi(\mathbf{false}))}{\mathsf{wp}_{\mathcal{E}}\, \mathsf{CAS}(\ell, v_1, v_2)\, \{\Phi\}} \qquad \textsc{wp-cas-suc}\ \frac{\triangleright\, \ell \mapsto v_1 \qquad \triangleright(\ell \mapsto v_2 \mathrel{-\!\!*} \Phi(\mathbf{true}))}{\mathsf{wp}_{\mathcal{E}}\, \mathsf{CAS}(\ell, v_1, v_2)\, \{\Phi\}}$$

$$\textsc{wp-fork}\ \frac{\mathsf{wp}\, e\, \{\_.\ \mathsf{True}\} \qquad \Phi(())}{\mathsf{wp}_{\mathcal{E}}\, \mathsf{fork}\, \{e\}\, \{\Phi\}} \qquad \textsc{wp-upd}\ \frac{\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\ \Rrightarrow_{\mathcal{E}} \Phi(v)\}}{\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

$$\textsc{wp-atomic}\ \frac{\mathsf{atomic}(e) \qquad {}^{\top}\!\Rrightarrow^{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \left\{v.\ {}^{\mathcal{E}}\!\Rrightarrow^{\top} \Phi(v)\right\}}{\mathsf{wp}\, e\, \{\Phi\}}$$

Figure 2.4: Weakest precondition rules.

The weakest precondition rules in Iris are given in what is called a "backwards" style [IO01; Rey02]. Contrast the presented WP-STORE rule with an equivalent one:

$$\frac{\ell \mapsto v}{\mathsf{wp}\, \ell \leftarrow w\, \{v'.\ v' = () * \ell \mapsto w\}}$$

This "backwards" style may seem inside-out at first, but it is quite intuitive when reasoning from conclusion to assumptions, as is usually done in proof assistants like Coq. To demonstrate this backwards style reasoning, consider an example where the assignment expression appears as part of a larger program.

Proposition 2.2. The following proposition (Equation (2.1)) holds:

$$\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2 \mathrel{-\!\!*} \mathsf{wp}\, (\ell_1 \leftarrow w_1; !\ell_2)\, \{v.\ v = v_2\}.$$

Proof. In this proof we will ignore the later modality $\triangleright$; we will return to it later in Section 2.2.3.

We have to prove

$$\mathsf{wp}\, (\ell_1 \leftarrow w_1; !\ell_2)\, \{v.\ v = v_2\}$$

```
llnil () = None                          llcons x l = Some(x, l)
llhead l = match l with                  lltail l = match l with
             | None → false                           | None → false
             | Some(h, _) → h                         | Some(_, t) → t
```

Figure 2.5: Purely functional linked lists.

under the assumptions $\ell_1 \mapsto v_1$ and $\ell_2 \mapsto v_2$. We start by applying WP-BIND with $K = [\bullet]; !\ell_2$. Our new goal becomes

$$\mathsf{wp}\ \ell \leftarrow w_1\ \{v.\ \mathsf{wp}\ (v; !\ell_2)\ \{v.\ v = v_2\}\},$$

with the assumptions unchanged.

We then apply WP-STORE, feeding $\ell_1 \mapsto v_1$ as a premise. After that it remains to show $\ell_1 \mapsto w_1 \mathrel{-\!\!*} \mathsf{wp}\ ((); !\ell_2)\ \{v.\ v = v_2\}$ under the assumption $\ell_2 \mapsto v_2$. Or, equivalently, by introduction of $\mathrel{-\!\!*}$, it remains to show

$$\mathsf{wp}\ ((); !\ell_2)\ \{v.\ v = v_2\}$$

under the assumptions $\ell_1 \mapsto w_1$ and $\ell_2 \mapsto v_2$.

By applying WP-PURE we can reduce the goal to $\mathsf{wp}\ !\ell_2\ \{v.\ v = v_2\}$. After which we apply WP-LOAD with $\ell_2 \mapsto v_2$ as the assumption. It remains to show $v_2 = v_2$. □

2.2.2 Representation predicates

In separation logic, specification are often expressed in terms of *representation predicates*, which establish relationships between logical data representation (*e.g.*, a mathematical list) and data representation in the programming language (*e.g.*, a linked list encoded in HeapLang). Iris supports defining representation predicates though all the facilities of Coq and higher-order logic; in particular, giving definitions by recursion. Let us demonstrate the usage of representation predicates in Iris on an example. Consider an implementation of purely functional linked lists given in Figure 2.5. The implementation provides a rather simple example, but we will use it to demonstrate some of the main concepts of Iris.

The lists are represented using option types with **None** being the empty list and $\mathsf{Some}(h, t)$ being the list with the head h and the tail t. On the level of the logic, we represent lists with the predicate $\mathsf{is\_list}(v, \vec{w}) : \mathit{iProp}$ which says that the linked list v represents a mathematical list (that is, a Coq-level list) $\vec{w}$ of values. This predicate is defined by recursion on the list $\vec{w}$:

$$\mathsf{is\_list}(v, \vec{w}) \triangleq \begin{cases} v = \mathsf{None} & \text{if } \vec{w} \text{ is } \epsilon \\ v = \mathsf{Some}(w_0, t) * \mathsf{is\_list}(t, \vec{w'}) & \text{if } \vec{w} = w_0 :: \vec{w'} \end{cases}$$

Using this predicate we prove the following specifications for the linked list functions:

$$\textsc{wp-llnil}\quad \mathsf{wp}\ \mathsf{llnil}\ ()\ \{v.\ \mathsf{is\_list}(v, \epsilon)\} \qquad \textsc{wp-llcons}\quad \frac{\mathsf{is\_list}(t, \vec{w})}{\mathsf{wp}\ \mathsf{llcons}\ h\ t\ \{v.\ \mathsf{is\_list}(v, h :: \vec{w})\}}$$

$$\textsc{wp-llhead}\quad \frac{\mathsf{is\_list}(v, h :: \vec{w})}{\mathsf{wp}\ \mathsf{llhead}\ v\ \{v'.\ (v' = h) * \mathsf{is\_list}(v, h :: \vec{w})\}} \qquad \textsc{wp-lltail}\quad \frac{\mathsf{is\_list}(v, h :: \vec{w})}{\mathsf{wp}\ \mathsf{lltail}\ v\ \{v'.\ \mathsf{is\_list}(v', \vec{w})\}}$$

$$\textsc{islist-dup}\quad \frac{\mathsf{is\_list}(v, \vec{w})}{\mathsf{is\_list}(v, \vec{w}) * \mathsf{is\_list}(v, \vec{w})}$$

The rule ISLIST-DUP is proven by induction on $\vec{w}$. All other specifications are proven by examining the definition of is_list and using WP-PURE.

The specifications that we gave exposes the is_list predicate and its definition. We could go further and turn is_list into an *abstract predicate* [PB05] using existential quantification [BBT07], thus hiding its definition from the clients, as described in [BB20, Section 4.2]. However, we do not do this in practice, as dealing with existentially quantified predicates introduce additional overhead. In the Coq formalization we just take extra care to make sure that whenever we verify a client of the linked list module, we do not appeal to the definition of the is_list predicate, but only to the associated proof rules.

2.2.3 Recursion and later modality

The rules in Figure 2.4 and the invariant rules feature the later modality $\triangleright$. This modality is customary in logics based on step-indexing [AM01; Nak00] and it has multiple uses in Iris. In this subsection we describe how to use the later modality for verifying recursive programs.

The key rules for the $\triangleright$ modality are:

$$\triangleright\textsc{-intro}\quad \frac{P}{\triangleright P} \qquad \triangleright\textsc{-mono}\quad \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \textsc{Löb}\quad \frac{\triangleright P \vdash P}{\vdash P} \qquad \triangleright\textsc{-sep}\quad \frac{\triangleright (P * Q)}{\triangleright P * \triangleright Q} \qquad \triangleright\textsc{-forall}\quad \frac{\forall x.\ \triangleright P}{\triangleright \forall x.\ P}$$

$$\triangleright\textsc{-exists}\quad \frac{\triangleright \exists x : A.\ P \qquad A \text{ is inhabited}}{\exists x : A.\ \triangleright P}$$

Furthermore, the $\triangleright$ modality distributes over intuitionistic conjunction and disjunction, and existential quantification over inhabited types.

Löb induction is encoded in the rule LÖB: it allows to prove some goal P using $\triangleright P$ as an induction hypothesis. Löb induction can be used to prove correctness of recursive programs, as demonstrated by the following proposition.

Proposition 2.3. Let $\Omega = (\lambda x.\ x\ x)(\lambda x.\ x\ x)$. The following proposition is derivable:

$$\mathsf{wp}\ \Omega\ \{\_.\ \mathsf{False}\}.$$

Proof. By application of Löb it suffices to show

$$\triangleright\, \mathsf{wp}\ \Omega\ \{\_.\ \mathsf{False}\} \vdash \mathsf{wp}\ \Omega\ \{\_.\ \mathsf{False}\}.$$

Note that $\Omega \to_{\mathsf{pure}} \Omega$; hence we can apply wp-pure to get a new goal

$$\triangleright\, \mathsf{wp}\ \Omega\ \{\_.\ \mathsf{False}\} \vdash \triangleright\, \mathsf{wp}\ \Omega\ \{\_.\ \mathsf{False}\},$$

which holds trivially. □

The proposition that we just proved might seem strange, because of the postcondition False. However, Iris is a program logic for proving *partial correctness*, that is, the postcondition requires to hold whenever the program terminates, but there is no requirement for the program to terminate.

Löb induction is useful for proving specifications for recursive programs that are not infinite loops. Consider the following function that we add to the linked list module in Figure 2.5:

$$\begin{aligned}\mathsf{llmember}\ x\ l = \mathsf{match}\ l\ \mathsf{with}\\ \mid \mathbf{None} \to \mathbf{false}\\ \mid \mathbf{Some}(h, t) \to \mathsf{if}\ h = x\ \mathsf{then}\ \mathbf{true}\\ \qquad\qquad \mathsf{else}\ \mathsf{llmember}\ x\ t\end{aligned}$$

The function goes through the linked list l looking for the element x. We can give it the following specification:

wp-llist-member

$$\frac{\mathsf{is\_list}(l, \vec{w})}{\mathsf{wp}\ \mathsf{llmember}\ x\ l \left\{ b.\ \mathsf{is\_list}(l, \vec{w}) * \begin{pmatrix} ((b = \mathbf{true}) * \exists i.\ w_i = x) \lor \\ ((b = \mathbf{false}) * \neg(\exists i.\ w_i = x)) \end{pmatrix} \right\}}$$

Since llmember is a recursive function, we can use Löb induction to prove it.[2]

Proposition 2.4. The rule wp-llist-member holds.

Proof. Let us use a shorthand

$$\Psi(b) \triangleq \mathsf{is\_list}(l, \vec{w}) * (((b = \mathbf{true}) * \exists i.\ w_i = x) \lor ((b = \mathbf{false}) * \neg(\exists i.\ w_i = x))).$$

By Löb induction it suffices to prove

$$\mathsf{is\_list}(l, \vec{w}) \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{llmember}\ x\ l\ \{\Psi\}$$

[2] In this particular case we can get away with doing induction on the length of the list. However, in general we might not have a measure that is decreasing at a recursive call. We will treat one such example in Section 2.4.2.

for arbitrary $l, \vec{w}$ under the assumption

$$\triangleright(\forall l,\ \vec{w}.\ \mathsf{is\_list}(l, \vec{w}) \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{llmember}\ x\ l\ \{\Psi\}).$$

By applying WP-PURE we can contract the beta-redex and get a goal

$$\triangleright \mathsf{wp} \left(\begin{array}{l} \mathsf{match}\ l\ \mathsf{with} \\ \mid \mathsf{None} \rightarrow \mathsf{false} \\ \mid \mathsf{Some}(h, t) \rightarrow \\ \qquad \mathsf{if}\ h = x\ \mathsf{then}\ \mathsf{true} \\ \qquad \mathsf{else}\ \mathsf{llmember}\ x\ t \end{array} \right) \{\Psi\}$$

under the assumptions

$$\triangleright(\forall l,\ \vec{w}.\ \mathsf{is\_list}(l, \vec{w}) \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{llmember}\ x\ l\ \{\Psi\}) * \mathsf{is\_list}(l, \vec{w}).$$

Using $\triangleright$-INTRO and distributivity of $\triangleright$ over $*$ we can rewrite the assumption as a single formula behind the $\triangleright$ modality:

$$\triangleright\Big((\forall l,\ \vec{w}.\ \mathsf{is\_list}(l, \vec{w}) \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{llmember}\ x\ l\ \{\Psi\}) * \mathsf{is\_list}(l, \vec{w})\Big).$$

At this point both our goal and our hypothesis are behind the $\triangleright$ modality; hence we can apply $\triangleright$-MONO to get rid of the $\triangleright$ modality on both sides.

We then proceed further by symbolic execution using WP-PURE (and $\triangleright$-INTRO whenever necessary). We need to consider several cases:

1. The list l is None. Then $\vec{w}$ is ϵ and $\Psi(\mathsf{false})$ holds.
2. The list l is $\mathsf{Some}(x, t)$. Then $\vec{w}$ contains x at the head position and $\Psi(\mathsf{true})$ holds.
3. The list l is $\mathsf{Some}(h, t)$ with $h \neq x$. Then the goal reduces to

   $$\mathsf{wp}\ \mathsf{llmember}\ x\ t\ \{\Psi\}.$$

   Since we have $\mathsf{is\_list}(t, \vec{w'})$, where $\vec{w'}$ is the tail of the list $\vec{w}$, we can solve this goal by applying the rule WP-WAND and using induction hypothesis.

□

2.2.4 Persistence modality

The weakest precondition connective in Iris is a first-class citizen, and it can be combined with other propositions through all the logical connectives. We have already seen the weakest precondition appear on the right-hand side of the magic wand, but in general the weakest precondition can appear at any place in a separation logic formula. This is especially useful for specifying and verifying higher-order

functions. As an example, consider the following addition to the list module in Figure 2.5:

$$\begin{aligned}&\mathsf{llfilter}\ f\ l = \mathsf{match}\ l\ \mathsf{with}\\ &\quad \mid \mathbf{None} \to \mathsf{llnil}\ ()\\ &\quad \mid \mathbf{Some}(h,t) \to \mathsf{if}\ f\ h\ \mathsf{then}\ \mathsf{llcons}\ h\ (\mathsf{llfilter}\ f\ t)\\ &\qquad\qquad\qquad \mathsf{else}\ \mathsf{llfilter}\ f\ t\end{aligned}$$

The function $\mathsf{llfilter}\ f\ l$ iterates over the list l, executing $f\ x$ for each element x of the list, keeping only those elements for which $f\ x$ returns **true**. In order to specify $\mathsf{llfilter}\ f\ l$ we might want to require that the function f satisfies some specification of the form $(\forall v.\ \mathsf{wp}\ f\ v\ \{v'.\ (v' = \mathbf{true} * P(v)) \lor (v' = \mathbf{false})\}$. However, recall that separation logic is substructural— if we require this specification for f as an assumption, then we can only use it once, despite the fact that for execution of $\mathsf{llfilter}\ f\ l$ we have to execute f multiple times. To address this issue, we require the specification for f to be *persistent*.

Intuitively, a proposition is *persistent* if, once established, it will remain valid for the rest of the verification. The notion of persistence is expressed in Iris through the *persistence* modality $\Box$. A proposition P is persistent if $P \vdash \Box P$ holds. The persistence modality satisfies the following rules:

$$\frac{\Box P * \Box P}{\Box P}\ \Box\text{-DUP} \qquad \frac{\Box P}{P}\ \Box\text{-ELIM} \qquad \frac{P \vdash Q}{\Box P \vdash \Box Q}\ \Box\text{-MONO} \qquad \frac{\Box P}{\Box\Box P}\ \Box\text{-IDEMP} \qquad \frac{\mathsf{True}}{\Box\mathsf{True}}\ \Box\text{-TRUE} \qquad \frac{\Box P \land Q}{\Box P * Q}\ \Box\text{-CONJ}$$

$$\frac{\Box P * \Box Q}{\Box(P * Q)}\ \Box\text{-SEP} \qquad \frac{\Box \triangleright P}{\triangleright \Box P}\ \Box\text{-}\triangleright \qquad \frac{\forall x.\ \Box P}{\Box \forall x.\ P}\ \Box\text{-FORALL} \qquad \frac{\Box \exists x.\ P}{\exists x.\ \Box P}\ \Box\text{-EXISTS}$$

The rules $\Box$-DUP and $\Box$-ELIM say that the $\Box P$ is duplicable, and one can get the proposition out of the persistence modality. The rule $\Box$-IDEMP says that $\Box P$ itself is persistent. The rules $\Box$-ELIM, $\Box$-MONO and $\Box$-IDEMP say that $\Box$ is in fact a co-monad. The rules $\Box$-CONJ and $\Box$-$\triangleright$ dictate the interaction of the persistence modality with conjunction and the later modality. Finally, $\Box$ commutes with some logical connectives like separating conjunction ($\Box$-SEP), the later modality ($\Box$-$\triangleright$), as well as universal and existential quantification ($\Box$-FORALL and $\Box$-EXISTS). By $\Box$-TRUE and $\Box$-MONO, from a closed proof $\vdash P$ of P we can obtain a closed proof $\vdash \Box P$ of $\Box P$.

Examples of persistent propositions include pure propositions (that is, propositions involving only intuitionistic logic connectives) and invariants, which we will encounter in Section 2.3.

Using the persistence modality we can then formulate a proper specification[3] for

[3] This particular specification can be strengthened by using the assumption $(\text{✻}_{v \in \vec{w}} \mathsf{wp}\ f\ v\ \{v'.\ (v' = \mathbf{true} * P(v)) \lor (v' = \mathbf{false})\})$ instead of the currently present assumption $\Box(\forall v.\ \mathsf{wp}\ f\ v\ \{v'.\ (v' = \mathbf{true} * P(v)) \lor (v' = \mathbf{false})\})$. However, the specification that we give is still useful; and the usage of the persistence modality for specifying higher-order function is idiomatic for logical relations models that will be presented in Chapters 4 and 5.

$$\begin{aligned}
\text{mset_create}\ () &= \text{ref}(\text{llnil}\ ()) \\
\text{mset_member}\ x\ v &= \text{let}\ l = !\,v\ \text{in}\ \text{llmember}(x, l) \\
\text{mset_add}\ x\ v &= \text{let}\ l = !\,v\ \text{in}\ v \leftarrow \text{llcons}\ x\ l \\
\text{mset_clear}\ v &= v \leftarrow \text{llnil}\ ()
\end{aligned}$$

Figure 2.6: Implementation of mutable sets.

llfilter:

$$\frac{\text{WP-LLFILTER}\quad \square(\forall v.\ \text{wp}\ f\ v\ \{v'.\ (v' = \textbf{true} * P(v)) \vee (v' = \textbf{false})\}) \qquad \text{is\_list}(l, \vec{w})}{\text{wp llfilter}\ f\ l \left\{ v'.\ \exists \vec{w'}.\ \text{is\_list}(v', \vec{w'}) * \mathop{\Large *}_{w'_i \in \vec{w'}} P(w'_i) \right\}}$$

This specification is proven using Löb induction and the rules for the $\square$ modality.

2.2.5 Stacking representation predicates

To see how the specification for linked lists can be used in practice, we will derive an specification for a small library of mutable sets (which we will use in Chapter 3), and define the representation predicate for mutable sets in terms of the representation predicate for lists.

An implementation of mutable sets using linked lists is given in Figure 2.6. A mutable set v is just a reference pointing to a linked list, whose elements comprise the contents of the set. All the operations querying and updating the set go through the list interface.

Using the specification for lists from the previous sections we can derive the specification for mutable sets as given in Figure 2.7. To prove the rules in that specification, we instantiate the predicate is_mset as follows:

$$\begin{aligned}
\text{is_mset}(v, X) \triangleq \exists hd,\ \vec{w},\ \ell.\ (v = \ell) * \ell \mapsto hd * \text{is_list}(hd, \vec{w}) * \\
\text{no_dup}(\vec{w}) * (X = \text{list_to_set}(\vec{w}))
\end{aligned}$$

We use an auxiliary function $\text{list\_to\_set} : \mathit{List}(\mathit{Val}) \to \wp(\mathit{Val})$ which converts a list of values to a set of values (ignoring the order), and an auxiliary predicate $\text{no\_dup} : \mathit{List}(\mathit{Val}) \to \mathit{iProp}$, which asserts that the list has no duplicates.

Let us show how to define one of the rules for mutable sets.

Proposition 2.5. The rule WP-MSET-CLEAR holds.

Proof. Suppose that $\text{is\_mset}(v, X)$ hold. Then, $v = \ell$ and $X = \text{list\_to\_set}(\vec{w})$ for some ℓ, $\vec{w}$. Furthermore, $\text{is\_list}(hd, \vec{w})$ and $\ell \mapsto hd$ for some hd. The goal is

$$\text{wp}\ (\ell \leftarrow \text{llnil}\ ())\ \{v'.\ v' = ()\ *\ \text{is\_mset}(\ell, \emptyset)\}.$$

WP-MSET-CREATE
$$\mathsf{wp}\ \mathsf{mset\_create}\ ()\ \{v.\ \mathsf{is\_mset}(v,\emptyset)\}$$

WP-MSET-CLEAR
$$\frac{\mathsf{is\_mset}(v,X)}{\mathsf{wp}\ \mathsf{mset\_clear}\ v\ \{v'.\ v' = () * \mathsf{is\_mset}(v,\emptyset)\}}$$

WP-MSET-ADD
$$\frac{\mathsf{is\_mset}(v,X) \qquad x \notin X}{\mathsf{wp}\ \mathsf{mset\_add}\ x\ v\ \{v'.\ v' = () * \mathsf{is\_mset}(v,\{x\} \cup X)\}}$$

WP-MSET-MEMBER
$$\frac{\mathsf{is\_mset}(v,X)}{\mathsf{wp}\ \mathsf{mset\_member}\ x\ v\ \left\{b.\ \mathsf{is\_mset}(v,X) * \left((b = \mathbf{true} * x \in X) \vee (b = \mathbf{false} * x \notin X)\right)\right\}}$$

Figure 2.7: Mutable sets specification.

By WP-BIND it suffices to prove

$$\mathsf{wp}\ \mathsf{llnil}\ ()\ \{v.\ \mathsf{wp}\ (\ell \leftarrow v)\ \{v'.\ v' = () * \mathsf{is\_mset}(\ell,\emptyset)\}\}.$$

Note that for llnil we have the rule WP-LLNIL: $\mathsf{wp}\ \mathsf{llnil}\ ()\ \{v.\ \mathsf{is\_list}(v,\epsilon)\}$. Thus, by WP-WAND we have to prove

$$\forall v.\ \mathsf{is\_list}(v,\epsilon) \mathrel{-\!\!*} \mathsf{wp}\ (\ell \leftarrow v)\ \{v'.\ v' = () * \mathsf{is\_mset}(\ell,\emptyset)\}.$$

We do this by symbolically executing the assignment, exchanging the assumption $\ell \mapsto hd$ for $\ell \mapsto v$. The return value of the assignment is always (); so, to prove the postcondition, we have to establish $\mathsf{is\_mset}(\ell,\emptyset)$. The latter can be done by unfolding the definition of is_mset and noting that $\mathsf{list\_to\_set}(\epsilon) = \emptyset$. □

The other specifications can be proven similarly.

2.3 Invariants in Iris

The rules of separation logic presented so work nicely for sequential programs. But the goal of a concurrent separation logic is to bring local reasoning to multi-threaded programs that operate on shared state. In order to share resources between different threads, Iris includes the mechanism of *invariants*. Invariants in Iris are not attached to particular synchronization primitives like locks or critical regions. Instead, invariants in Iris exist "on their own", and the resources shared via such invariants can be accessed by any thread, but only for the duration of a single reduction step. However, more standard rules for sharing resources through synchronization primitives can be derived from the Iris invariants (we will see an example of that in Section 2.4.2).

To explain the invariant mechanism in Iris, let us start with an example. Consider the following program:

$$\begin{aligned}\mathsf{prog}_1 \triangleq\ & \mathsf{let}\ x = \mathsf{ref}(0)\ \mathsf{in}\\ & \mathsf{fork}\ \{x \leftarrow 1\};\\ & !x\end{aligned}$$

We want to prove that prog_1 returns either 0 or 1. We can try proving this by symbolically allocating the location x to get a predicate $x \mapsto 0$. Then, we might wish to apply WP-FORK; we would be left with the following goal:

$$x \mapsto 0 \mathrel{-\!\!*} \Big(\big(\mathsf{wp}\ (x \leftarrow 1)\ \{\_.\ \mathsf{True}\}\big) * \big(\mathsf{wp}\ ((); !x)\ \{x \mapsto 0 \lor x \mapsto 1\}\big)\Big).$$

We would have to prove two weakest precondition connectives: one for the forked-off thread, and one for the main thread. Both of the threads require access to the resource $x \mapsto 0$. However, as it stands, we cannot share the resource $x \mapsto 0$ across the two weakest preconditions joined by the separating conjunction.

In order to share the points-to predicate $x \mapsto 0$ between the main thread of prog_1 and the forked-off thread, we will use the Iris invariants. Invariants in Iris are propositions of the form $\boxed{P}^{\mathcal{N}}$, which allow the user to share P between different threads, as long as all the operations in all of the threads respect P (the role of invariant names $\mathcal{N}$ is discussed later in Section 2.3.1). Invariants $\boxed{P}^{\mathcal{N}}$ are persistent: $\boxed{P}^{\mathcal{N}} \vdash \Box \boxed{P}^{\mathcal{N}}$. So, invariant propositions are duplicable (via $\Box$-DUP) and can be shared across separating conjunction. Additionally, we will make use of the following rules for invariants:

WP-INV-ALLOC
$$\frac{\triangleright P \qquad (\boxed{P}^{\mathcal{N}} \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\})}{\mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}}$$

WP-INV-ACCESS
$$\frac{\mathsf{atomic}(e) \qquad \boxed{P}^{\mathcal{N}} \qquad \triangleright P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}\uparrow}\ e\ \{v.\ \triangleright P * \Phi(v)\}}{\mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}}$$

The first rule WP-INV-ALLOC allows us to allocate the resources P into an invariant $\boxed{P}^{\mathcal{N}}$, which then can be used for proving the program e. The second rule WP-INV-ACCESS allows us to temporarily access the contents of the invariant $\boxed{P}^{\mathcal{N}}$ while performing an *atomic* operation e. An operation is atomic (denoted as $\mathsf{atomic}(e)$) if it reduces to a value in one step. Examples of atomic operations are assignment, dereferencing, compare-and-set, and so on. Because an atomic operation takes exactly one step of execution, no other thread can be interleaved with it. This makes it sound to access and temporarily break an invariant, for a duration of an atomic operation.

Invariants in Iris are *impredicative* [SB14; Jun+18b]. That is, the proposition P in the invariant $\boxed{P}^{\mathcal{N}}$ can itself contain another invariant assertion $\boxed{Q}^{\mathcal{N}'}$. As a consequence, to ensure soundness of the logic, all rules for invariants only provide access to $\triangleright P$, *i.e.*, P guarded by the *later* modality $\triangleright$. To still be able to use such guarded resources in symbolic execution rules in Figure 2.4, the premises of those rules are also guarded, *i.e.*, the premise $\triangleright \ell \mapsto v$ in WP-STORE.

Let us see how those invariant rules are used to prove the specification for prog_1.

Proposition 2.6. The following proposition holds: $\mathsf{wp}\ \mathsf{prog}_1\ \{v.\ v = 0 \lor v = 1\}$.

Proof. We start by applying the rule WP-ALLOC, after which we have to prove

$$\mathsf{wp}\ (\mathsf{fork}\ \{x \leftarrow 1\}; !x)\ \{v.\ v = 0 \lor v = 1\}$$

under the assumption $x \mapsto 0$, for some arbitrary location x.

We apply WP-INV-ALLOC picking the invariant to be $\boxed{x \mapsto 0 \lor x \mapsto 1}^{\mathcal{N}}$. Clearly, we can establish $(x \mapsto 0 \lor x \mapsto 1)$ from $x \mapsto 0$. Then we apply WP-BIND and WP-FORK, obtaining two new goals:

1. $\mathsf{wp}\ x \leftarrow 1\ \{\_.\ \mathsf{True}\}$
2. $\mathsf{wp}\ ((); !x)\ \{v.\ v = 0 \lor v = 1\}$

Since the location x is now shared in the invariant, we can use the assumption $\boxed{x \mapsto 0 \lor x \mapsto 1}^{\mathcal{N}}$ to prove both of those goals.

For the first goal we apply WP-INV-ACCESS, reducing it to the following (ignoring the later modality $\triangleright$ for now):

$$(x \mapsto 0 \lor x \mapsto 1) \twoheadrightarrow \mathsf{wp}_{\top \setminus \mathcal{N}^{\uparrow}}\ (x \leftarrow 1)\ \{\_.\ x \mapsto 0 \lor x \mapsto 1\}.$$

That is, we have to show that executing $x \leftarrow 1$ preserves the invariant that we have established. Proving this goal is a matter of case analysis on the disjunction and applying WP-STORE.

Similarly, we prove the second goal by applying WP-PURE, WP-INV-ACCESS and WP-LOAD. During the symbolic execution operation we gain access to the invariant and thus know that whatever value we dereference from x will be either 0 or 1. □

2.3.1 Namespaces and masks

A crucial aspect of the invariant mechanism is that an invariant $\boxed{P}^{\mathcal{N}}$ cannot be opened twice: that would not be sound.[4] For example, if we could open $\boxed{\ell \mapsto v}^{\mathcal{N}}$ twice in a row, we would get access to $\ell \mapsto v * \ell \mapsto v$, which is a contradiction because one cannot own the same location twice.

To enforce that invariants can only be opened once, the invariant proposition $\boxed{P}^{\mathcal{N}}$ is tagged with a *namespace* $\mathcal{N} \in \mathit{InvName}$, and the weakest precondition connective $\mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}$ is annotated with a *mask* $\mathcal{E} \subseteq \mathit{InvName}$ of potential invariants that can be opened. By default, all invariants are accessible for the weakest precondition connective, so we write $\mathsf{wp}\ e\ \{\Phi\}$ for $\mathsf{wp}_{\top}\ e\ \{\Phi\}$ where $\top$ is the largest mask (the set $\mathit{InvName}$ itself).

An invariant namespace is a (non-empty) list of strings or values: $\mathit{InvName} = \mathit{List}(\mathit{String} + \mathit{Val})$. When opening an invariant (*e.g.*, using WP-INV-ACCESS) with the namespace $\mathcal{N}$ and removing it from the mask on the weakest precondition ($\mathcal{E} \setminus \mathcal{N}^{\uparrow}$), we coerce the namespace $\mathcal{N}$ into a mask by taking its upwards extension: $\mathcal{N}^{\uparrow} = \{\mathcal{N} x_1 \ldots x_n \mid n \in \mathbb{N}, x_i \in \mathit{String} + \mathit{Val}\}$.

2.3.2 Invariants and the update modality

The rules for the invariants that we have presented so far are fairly limited: they only allow to open an invariant when proving a weakest precondition proposition and they only allow to open one invariant at a time. In truth, the rules WP-INV-ALLOC and WP-INV-ACCESS are not primitive rules of Iris but are derived from the primitive

[4] This sometimes is referred to as a reentrancy issue of the invariant mechanism.

invariant rules and WP-UPD and WP-ATOMIC, by using the *update modality* ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$. The intuition behind ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$ is to express that under the assumption that the invariants in $\mathcal{E}_1$ are accessible initially, one can obtain P, and end up in the situation where the invariants in $\mathcal{E}_2$ are accessible. Thus, for showing P we can open the invariants from $\mathcal{E}_1$ and have to restore the invariants from $\mathcal{E}_2$ (the invariants from $\mathcal{E}_1 \setminus \mathcal{E}_2$ may remain open). Furthermore, this modality allows one to perform changes to Iris's ghost state, as we will see in Section 2.4.

The key rules of the update modality are:

$$\Rightarrow\text{-INTRO}\ \frac{P}{{}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} P} \qquad \Rightarrow\text{-MONO}\ \frac{P \vdash Q}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q} \qquad \Rightarrow\text{-IDEMP}\ \frac{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_3} P}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_3} P} \qquad \Rightarrow\text{-SEP}\ \frac{P * {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} (P * Q)}$$

These rules say that the update modality is a strong indexed monad. To use the update modality in practice we have the following derived rule:

$$\Rightarrow\text{-ELIM}\ \frac{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \qquad P \mathrel{-\!\!*} {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_3} Q}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_3} Q}$$

This rule is derivable from $\Rightarrow$-MONO, and $\Rightarrow$-IDEMP, and we refer to the application of this rule as *eliminating* the update modality.

For convenience we have the following notations. We write $\Rightarrow_{\mathcal{E}}$ for ${}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}}$, and we write $\Rightarrow$ for $\Rightarrow_{\top}$. We write $P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}\; Q$ for $P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q$, with a similar convention for the mask annotations.

The following rules connect the invariants with the update modality:

$$\text{INV-ALLOC}\ \frac{\triangleright P}{\Rightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}} \qquad \text{INV-ACCESS}\ \frac{\mathcal{N}^{\uparrow} \subseteq \mathcal{E} \qquad \boxed{P}^{\mathcal{N}}}{{}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}\setminus\mathcal{N}^{\uparrow}} \triangleright P * (\triangleright P \;{}^{\mathcal{E}\setminus\mathcal{N}^{\uparrow}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True})}$$

As one can check, using $\Rightarrow$-ELIM and $\Rightarrow$-INTRO we can derive WP-INV-ALLOC from INV-ALLOC and WP-UPD, and we can derive WP-INV-ACCESS from INV-ACCESS and WP-ATOMIC.

2.4 Custom ghost state in Iris

The invariant mechanism of Iris is especially powerful when combined with Iris's facilities for defining *custom ghost state*. Ghost state (also known as auxiliary state) is dubbed so in contrast with the physical state of the program, like (parts of) the heaps. Unlike physical state, ghost state is a purely logical construct, which is not a priori related to the behavior of the program, but is introduced only for the purposes for specification and verification of the program.

Iris supports custom, user-defined ghost state theories encoded using *resource algebras* [Jun+16]. The details on the encoding of ghost state using resource algebras can be found in [Jun+18b, Section 3] and [BB20, Section 7.4]. In this thesis we will

not give encodings of any custom ghost state theories, but rather present the ghost theories "as is", in terms of the abstract predicates and the associated proof rules. An interested reader is referred to the Coq formalization for the details on how those ghost theories are defined.

In the rest of this section we present examples of two custom ghost state theories, and demonstrate how to use them for reasoning about concurrent programs.

2.4.1 Parallel composition

Consider the following implementation of parallel composition operation using the fork $\{-\}$ operation:

$$\begin{aligned}
\mathsf{join}\ x &= \mathsf{match}\ !x\ \mathsf{with}\ \mathbf{Some}(v) \to v \\
&\quad \mid \mathbf{None} \to \mathsf{join}\ x \\
\mathsf{par}\ f_1\ f_2 &= \mathsf{let}\ x = \mathsf{ref}(\mathbf{None})\ \mathsf{in} \\
&\quad \mathsf{fork}\ \{x \leftarrow \mathbf{Some}(f_1\ ())\}; \\
&\quad \mathsf{let}\ v_2 = f_2\ ()\ \mathsf{in}\ (\mathsf{join}\ x, v_2)
\end{aligned}$$

We use the notation $e_1 \,\|\, e_2$ to denote $\mathsf{par}\ (\lambda\_.\ e_1)\ (\lambda\_.\ e_2)$. The $e_1 \,\|\, e_2$ expression spawns a new thread for executing e_1 and storing its result in a local reference x. The main thread then proceeds to execute e_2, and, when this execution is completed, it waits for the spawned thread to write the result to x.

We wish to give the standard separation logic specification to this parallel composition operation:

$$\frac{\mathsf{wp}\ e_1\ \{\Psi_1\} \qquad \mathsf{wp}\ e_2\ \{\Psi_2\} \qquad (\forall v_1\ v_2.\ \Psi_1(v_1) * \Psi_2(v_2) \mathrel{-\!\!*} \Phi((v_1, v_2)))}{\mathsf{wp}\ e_1 \,\|\, e_2\ \{\Phi\}}\ \textsc{wp-par}$$

In order to prove WP-PAR, the location x has to be shared between the two threads. Furthermore, the location x has to satisfy the following protocol: either it contains **None** (the expression e_1 is not computed), or it contains $\mathbf{Some}(v_1)$ with v_1 satisfying the predicate Ψ_1 (the expression e_1 has been computed). Moreover, we want to ensure that once the spawned thread computes the expression e_1 to a value v_1 satisfying Ψ_1, the cumulative resources $\Psi_1(v_1)$ can be transferred to the original thread.

To that extent we introduce the a custom ghost state theory that contains a predicate token_γ that satisfies the following rules:

$$\frac{}{\Rightarrow_{\mathcal{E}} \exists \gamma.\ \mathsf{token}_\gamma}\ \textsc{new-token} \qquad \frac{\mathsf{token}_\gamma \qquad \mathsf{token}_\gamma}{\mathsf{False}}\ \textsc{token-excl}$$

This ghost theory contains a single proposition token (indexed by a ghost name $\gamma \in \mathit{GName}$). This proposition is *exclusive*, *i.e.*, only one such proposition can exist, as witnessed by TOKEN-EXCL. Moreover, a new token_γ with a "fresh" ghost name can be allocated using NEW-TOKEN.

To prove WP-PAR we will use the following invariant:

$$I_{\|}(\gamma, x) = \boxed{x \mapsto \mathsf{None} \lor \exists v.\, x \mapsto \mathsf{Some}(v) * (\Psi_1(v) \lor \mathsf{token}_\gamma)}^{\mathcal{N}}.$$

It says that either x contains **None**, or x contains $\mathsf{Some}(v)$. In the later case the invariant either owns the resources $\Psi_1(v)$ or an exclusive token token_γ. The **None** branch corresponds to the situation when the forked-off thread has not finished the computation yet. The $\mathsf{Some}(v)$ case corresponds to the situation when the forked-off thread has finished a computation with a value v. Furthermore, the computation that the forked-off thread has performed satisfies the predicate $\Psi_1(v)$. Either this predicate is in the invariant, or it has been taken out by the main thread in exchange for the token token_γ.

This invariant allows us to prove the following useful lemma for join:

Lemma 2.7. The following proposition holds:

$$I_{\|}(\gamma, x) * \mathsf{token}_\gamma \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{join}\ x\ \{\Psi_1\}$$

for any γ, x.

Proof. By Löb induction we can assume $\triangleright(I_{\|}(\gamma, x) * \mathsf{token}_\gamma \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{join}\ x\ \{\Psi_1\})$. Using WP-PURE we can contract the beta-redex, and get rid of the $\triangleright$ modality, reducing our goal to:

$$(\mathsf{token}_\gamma \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{join}\ x\ \{\Psi_1\}) * \mathsf{token}_\gamma \mathrel{-\!\!*} \mathsf{wp}\ (\mathsf{match}\ !x\ \mathsf{with}\ \ldots)\ \{\Psi_1\}.$$

We use WP-ATOMIC and WP-LOAD to symbolically dereference x. There are two cases to consider.

1. We get $x \mapsto \mathsf{None}$ out of the invariant. In this case we can readily close the invariant and use WP-PURE to "restart" the computation of join. At this point we just appeal to the induction hypothesis, which we can use because we have not spent the resource token_γ.
2. We get $x \mapsto \mathsf{Some}(v)$ for some v our of the invariant. Furthermore, we get $\Psi_1(v) \lor \mathsf{token}_\gamma$ out of the invariant as well. Since we have token_γ, we note that the second disjunct is inconsistent with our assumptions. Indeed, were we able to get token_γ out of the invariant, we would have two copies of token_γ, which is impossible by TOKEN-EXCL. Hence it must be the case that $\Psi_1(v)$.
   Then we close the invariant by exchanging this resource $\Psi_1(v)$ for token_γ which we have. Then we just apply WP-PURE and obtain the desired goal. □

Using this lemma we can verify the rule WP-PAR:

Proposition 2.8. The rule WP-PAR is sound.

Proof. Suppose $\mathsf{wp}\ e_1\ \{\Psi_1\}$ and $\mathsf{wp}\ e_2\ \{\Psi_2\}$. By symbolic execution, using WP-PURE and WP-ALLOC it suffices to show

$$\mathsf{wp} \begin{pmatrix} \mathsf{fork}\ \{x \leftarrow \mathsf{Some}((\lambda\_.\ e_1)\ ())\}; \\ \mathsf{let}\ v_2 = (\lambda\_.\ e_2)\ ()\ \mathsf{in} \\ (\mathsf{join}\ x, v_2) \end{pmatrix} \{\Phi\}.$$

under the assumption $x \mapsto \mathsf{None}$ for an arbitrary location x. At this point we can use WP-UPD and NEW-TOKEN to allocate a new token token_γ for a fresh $\gamma \in \mathit{GName}$. We can then use INV-ALLOC to establish the invariant $I_{\|}$, using the points-to connective $x \mapsto \mathsf{None}$.

We then apply WP-BIND and WP-FORK, and get two new goals:

$$\mathsf{wp}\ (x \leftarrow \mathsf{Some}((\lambda\_.\ e_1)\ ()))\ \{\_.\ \mathsf{True}\} \quad * \quad \mathsf{wp}\ (\mathsf{let}\ v_2 = (\lambda\_.\ e_2)\ ()\ \mathsf{in}\ (\mathsf{join}\ x, v_2))\ \{\Phi\}.$$

We can share the invariant $I_{\|}$ between both goals, and we use token_γ to show the second goal.

For the first goal we use WP-PURE to reduce it to $\mathsf{wp}\ (x \leftarrow \mathsf{Some}(e_1))\ \{\_.\ \mathsf{True}\}$. We then use our assumption $\mathsf{wp}\ e_1\ \{\Psi_1\}$, in combination with WP-BIND and WP-WAND to reduce the goal to showing $\Psi_1(v_1) \mathrel{-\!\!*} \mathsf{wp}\ (x \leftarrow \mathsf{Some}(v_1))\ \{\_.\ \mathsf{True}\}$ for an arbitrary v_1. We can prove this goal by using WP-ATOMIC and WP-STORE. Because we have the resources $\Psi_1(v_1)$ we can successfully "close" the invariant by picking the correct disjunct.

For the second goal, after applying WP-PURE we have to show

$$\mathsf{token}_\gamma \mathrel{-\!\!*} \mathsf{wp}\ (\mathsf{let}\ v_2 = e_2\ \mathsf{in}\ (\mathsf{join}\ x, v_2))\ \{\Phi\}.$$

Similarly to the first goal, we appeal to the assumption $\mathsf{wp}\ e_2\ \{\Psi_2\}$ and WP-BIND, WP-WAND to show

$$\mathsf{token}_\gamma * \Psi_2(v_2) \mathrel{-\!\!*} \mathsf{wp}\ (\mathsf{join}\ x, v_2)\ \{\Phi\}$$

for an arbitrary v_2. This goal then follows from Lemma 2.7 and the assumption $(\forall v_1\, v_2.\ \Psi_1(v_1) * \Psi_2(v_2) \mathrel{-\!\!*} \Phi((v_1, v_2)))$. □

2.4.2 Spin lock

Another use of the ghost theory from Section 2.4.1 is for specification of locks. The desired specification is presented in Figure 2.8, and it is formulated in a style of iCAP [SB14] or [HAN08]. Locks prevent simultaneous access to some resources. We can think of it as locks "owning" resources in the same way that threads can own them. The specification we give allows for dynamic creation of locks: when a new lock is created with the newlock function, we associate a predicate $\mathsf{is\_lock}(\gamma, lk, R)$ with it. The predicate says that the lock lk has a ghost name γ and controls the resources R. When symbolically execute the newlock function, we have to transfer the resources R into the lock, as per WP-NEWLOCK. When a client enters a critical section by calling the acquire function, it obtains those resources R as well as a token $\mathsf{locked}(\gamma)$ signifying the ownership of the critical section. Both R and $\mathsf{locked}(\gamma)$ have to be given up when a client exists a critical section upon invoking release.

The lock specification in Figure 2.8 can be proven for many different kinds of locks. However, for this example we take a simple *spin lock* implementation that adheres to the specification. The implementation is defined as follows:

$$\begin{aligned}
\mathsf{newlock}\ () &= \mathsf{ref}(\mathsf{false}) \\
\mathsf{acquire}\ lk &= \mathsf{if}\ \mathsf{CAS}(lk, \mathsf{false}, \mathsf{true})\ \mathsf{then}\ () \\
&\qquad\quad \mathsf{else}\ \mathsf{acquire}\ lk \\
\mathsf{release}\ lk &= lk \leftarrow \mathsf{false}
\end{aligned}$$

$$\frac{R \qquad (\forall\, \gamma\, lk.\ \mathsf{is\_lock}(\gamma, lk, R) \mathrel{-\!\!*} \Phi\ lk)}{\mathsf{wp\ newlock}\ ()\ \{\Phi\}}\ \textsc{wp-newlock} \qquad \frac{\mathsf{is\_lock}(\gamma, lk, R)}{\Box\, \mathsf{is\_lock}(\gamma, lk, R)}$$

$$\frac{\mathsf{locked}(\gamma) \qquad \mathsf{locked}(\gamma)}{\mathsf{False}} \qquad \frac{\mathsf{is\_lock}(\gamma, lk, R) \qquad (R * \mathsf{locked}(\gamma) \mathrel{-\!\!*} \Phi(()))}{\mathsf{wp\ acquire}\ lk\ \{\Phi\}}\ \textsc{wp-acquire}$$

$$\frac{\mathsf{is\_lock}(\gamma, lk, R) \qquad R \qquad \mathsf{locked}(\gamma) \qquad \Phi(())}{\mathsf{wp\ release}\ lk\ \{\Phi\}}\ \textsc{wp-release}$$

Figure 2.8: Specification for locks.

A spin lock consists of a single reference lk to a boolean. If the boolean is **false**, then the lock is in the unlocked state; if the boolean is **true** then someone is using a lock.

In order to acquire the lock, a thread repeatedly tries to atomically change the value of the reference lk from **false** to **true**. If a thread does not succeed in doing so, then another thread must be currently using the lock, and the original thread remains *spinning* on the location lk.[5]

To release the lock, a thread just sets the value of lk to **false**. There is no race on the location lk at this point, because we only expect only one thread calling the release function.

We can verify the specifications in Figure 2.8 against the spin lock implementation by picking the following definitions for representation predicates, using the custom ghost state theory from the previous subsection:

$$\mathsf{is\_lock}(\gamma, v, R) \triangleq \exists \mathcal{N}.\ \exists lk \in Loc.\ v = lk * \boxed{\exists b \in \mathbb{B}.\ lk \mapsto b * (b = \mathbf{true} \vee \mathsf{locked}(\gamma) * R)}^{\mathcal{N}}$$

$$\mathsf{locked}(\gamma) \triangleq \mathsf{token}_\gamma$$

As an example, let us go through the proof of one of the rules.

Proposition 2.9. The rule WP-ACQUIRE for the spin lock implementation.

Proof. By Löb induction, we can assume the induction hypothesis:

$$\triangleright\Big(\mathsf{is\_lock}(\gamma, lk, R) * (R * \mathsf{locked}(\gamma) \mathrel{-\!\!*} \Phi(())) \mathrel{-\!\!*} \mathsf{wp\ acquire}\ lk\ \{\Phi\}\Big).$$

[5] In practice, implementations for spin locks also use some sort of time out mechanism to avoid aimlessly being in a busy loop, see [HS08, Chapter 7]. However, the presence or absence of such time outs is irrelevant in our case because from the point of view of the program behavior, busy waiting is indistinguishable from busy waiting interleaved with time outs.

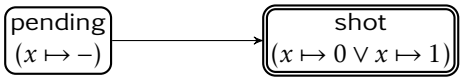


Figure 2.9: "One shot" transition system.

We can get rid of the $\triangleright$ by symbolically executing the goal wp acquire lk $\{\Phi\}$. At the point of the CAS(lk, **false**, **true**) operation, we open the invariant (which is given through the predicate is_lock(γ, lk, R)) and consider two possible scenarios:

1. If $b =$ **true**, then the CAS operation fails. We close the invariant and proceed by symbolic execution, until we can apply the induction hypothesis.
2. If $b =$ **false**, then the CAS operation succeeds. We can take out the resources locked(γ) and R from the invariant, before closing it. We then simply apply one of the premises of the rule: $R * \mathsf{locked}(\gamma) \mathrel{-\!\!*} \Phi(())$.

□

2.4.3 Transition systems as protocols

For the next example, consider the program prog_2, which is a slight modification of the program prog_1 from Section 2.3:

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{ref}(2)\ \mathsf{in}\\ \mathsf{fork}\ \{x \leftarrow 1\};\\ x \leftarrow 0;\\ !x\end{array}$$

We should be able to prove that prog_2, just like prog_1 returns either 0 or 1. Unfortunately, we cannot use the same invariant $\boxed{x \mapsto 0 \lor x \mapsto 1}^{\mathcal{N}}$, because in its initial state the location x contains 2. So we would not be able to establish this invariant at the beginning!

To prove that prog_2 returns either 0 or 1 we need to establish a stronger protocol on the location x. What we want to say is that *initially* the location x is arbitrary, but by the time we reach the dereferencing $!x$, the value store in x falls within the set $\{0, 1\}$. Furthermore, once the value of x becomes an element of $\{0, 1\}$ it stays within this set of possible values. This can be visualized as the transition system in Figure 2.9. The transition system can be encoded using the following ghost theory:[6]

NEW-PENDING
$\Rightarrow\!\!\Vdash \exists \gamma.\ \mathsf{pending}_\gamma$

PENDING-NOT-SHOT
$$\frac{\mathsf{pending}_\gamma \qquad \mathsf{shot}_\gamma}{\mathsf{False}}$$

SHOT-PERSISTENT
$$\frac{\mathsf{shot}_\gamma}{\Box \mathsf{shot}_\gamma}$$

PENDING-SHOOT
$$\frac{\mathsf{pending}_\gamma}{\Rightarrow\!\!\Vdash \mathsf{shot}_\gamma}$$

The predicate $\mathsf{pending}_\gamma$ corresponds to the initial state and the predicate shot_γ corresponds to the terminal state. The state can be changed from $\mathsf{pending}_\gamma$ to shot_γ

[6] This ghost theory is implemented with the "oneshot" algebra from [Jun+18b, Section 2.1.].

(PENDING-SHOOT). Because shot is a *final* state, the corresponding resource shot_γ is *persistent* (SHOT-PERSISTENT): this corresponds to the fact that once the protocol ends in the state shot it cannot evolve further.

Using this ghost theory we can formulate an invariant for proving the specification for prog_2.

Proposition 2.10. The following proposition holds:

$$\mathsf{wp}\ \mathsf{prog}_2\ \{v.\ v = 0 \lor v = 1\}$$

Proof. As in Proposition 2.6 we start with WP-ALLOC, obtaining $x \mapsto 2$. We can then allocate $\mathsf{pending}_\gamma$ using NEW-PENDING. We can use those two propositions for establish the following invariant:

$$\boxed{\left(\mathsf{pending}_\gamma * x \mapsto -\right) \lor \left(\mathsf{shot}_\gamma * (x \mapsto 0 \lor x \mapsto 1)\right)}^{\mathcal{N}}.$$

Then, after applying WP-FORK, we obtain two new goals:

1. $\mathsf{wp}\ x \leftarrow 1\ \{\_.\ \mathsf{True}\}$
2. $\mathsf{wp}\ ((); x \leftarrow 0; !x)\ \{v.\ v = 0 \lor v = 1\}$

For the first goal we symbolically execute the assignment $x \leftarrow 1$ by using the invariant. Depending on the disjunct in the invariant we might either already be in the shot state, or we have to update $\mathsf{pending}_\gamma$ to shot_γ using PENDING-SHOOT.

Similarly in the second goal we symbolically execute the assignment $x \leftarrow 0$. Irregardless of the disjunct in the invariant, we can obtain the predicate $\mathsf{pending}_\gamma$ which is duplicable by SHOT-PERSISTENT. That means that we can keep a copy of shot_γ for ourselves, in addition to storing another copy of shot_γ in the invariant.

Then, when we symbolically dereference x, we notice that the left disjunct in the invariant cannot be true: because $\mathsf{pending}_\gamma$ from the invariant is incompatible with shot_γ that we have (PENDING-NOT-SHOT). □

2.5 The Coq mechanization

Iris comes with a Coq formalization that includes not only the soundness theorem and the proof rules, but also an *interactive proof mode* for carrying out tactic-based proofs in Iris (and other separation logics), as if the user is dealing with regular Coq proofs. In this section we give a taste of how this interactive proof mode is used to formally reason about separation logic proofs in Coq.

Interactive separation logic proofs. The Iris Proof Mode (IPM) [KTB17] and its successor MoSeL [Kre+18] allow us to carry out separation logic proofs interactively, in the style of regular tactic-based proofs in Coq. IPM provides a convenient representation of sequents for separation logic and tactics for manipulating them, allowing for interactive proof development in the style of regular proofs in Coq. To illustrate this, consider the following separation logic tautology:

```
------------------*
P -* (P -* Q) -* Q
```

(a) Before executing any tactics.

```
"H1" : P
"H2" : P -* Q
------------------*
Q
```

(b) After executing `iIntros "H1 H2".`

Figure 2.10: Interactive proof of lemma `example` in IPM.

```
Lemma example (P Q : iProp Σ) : P -* (P -* Q) -* Q.
Proof. iIntros "H1 H2". iApply ("H2" with "H1"). Qed.
```

Here, `iProp Σ` is the type of Iris propositions.[7]

The intermediate results of running the `example` proof script can be seen in Figure 2.10. Applying `iIntros "H1 H2"` introduces the hypothesis `P` and `P -* Q` into the IPM context, giving them names `H1` and `H2`, respectively. Then, `iApply ("H2" with "H1")` applies the separating implication `P -* Q` to the goal, using the hypothesis `H1 : P` as the assumption.

The bar `------------------*` in Figure 2.10 separates the separation logic context from the goal; the proof state in Figure 2.10a corresponds to the sequent $\mathsf{True} \vdash P \mathrel{-\!\!*} (P \mathrel{-\!\!*} Q) \mathrel{-\!\!*} Q$, and the proof state in Figure 2.10b corresponds to the sequent $P * (P \mathrel{-\!\!*} Q) \vdash Q$. In addition to the separation logic context, there is usually a Coq-level context which contains identifiers and hypothesis introduced on the Coq level. Consider, for example, the following modification of the `example` lemma:

```
Lemma example₂ (P Q : nat → iProp Σ) :
  (∀ x, P x -* Q (x+1)) -* (∃ x, P x) -* (∃ x, Q x).
Proof.
  iIntros "H1 H2". iDestruct "H2" as (y) "H2".
  iExists (y+1). iApply ("H1" with "H2").
Qed.
```

In this lemma, `P` and `Q` are Iris predicates over the type `nat` of natural numbers in Coq.

The intermediate results of running the `example₂` proof script can be seen in Figure 2.11. In this example, the bar `______________________` separates the Coq context from the separation logic context. When we "destruct" an existential `∃ x : nat, P x` (*i.e.*, when we use the existential elimination rule), we add the witness `x` to the Coq context. We later refer to it when executing the `iExists (y+1)` tactic (*i.e.*, when we use the existential introduction rule).

[7] The parameter Σ describes the kind of ghost state available in Iris. It is an important but technical detail that can safely be ignored for the purpose of this thesis. An interested reader is directed to [Jun+18b, Section 4.7].

```
----------------------------------------------------------*
(∀ x, P x -∗ Q (x + 1)) -∗ (∃ x : nat, P x) -∗ ∃ x, Q x
```

(a) Before executing any tactics.

```
y : nat
--------------------------------------
"H1" : ∀ x : nat, P x -∗ Q (x + 1)
"H2" : P y
--------------------------------------*
∃ x : nat, Q x
```

(b) After executing `iIntros "H1 H2". iDestruct "H1" as (y) "H1".`

```
y : nat
--------------------------------------
"H1" : ∀ x : nat, P x -∗ Q (x + 1)
"H2" : P y
--------------------------------------*
Q (y + 1)
```

(c) After executing `iExists (y+1)`.

Figure 2.11: Interactive proof of lemma $\texttt{example}_2$ in IPM.

Symbolic execution tactics. In addition to tactics like `iIntros` and `iApply`, IPM provides tactic for symbolic execution in weakest preconditions. Let us demonstrate their usage on an example. Consider the following formalization of Proposition 2.2:

```
Lemma example_wp (l1 l2 : loc) (v1 v2 w1 : val) :
  l1 ↦ v1 ∗ l2 ↦ v2 -∗
  WP (#l1 ← w1;; !#l2) {{ v, ⌜v = v2⌝ }}.
Proof.
  iIntros "[H1 H2]". wp_store. wp_load. eauto.
Qed.
```

Here, we use Iris's notion $\ulcorner\varphi\urcorner$ `: iProp Σ` to embed Coq propositions φ `: Prop` into Iris, although on paper we take the equality predicate to be primitive. The `#l` embeds locations `l : loc` into the type of expressions, although on paper we ignore it as it is usually clear from context.

The intermediate results of running the `example_wp` proof script can be seen in Figure 2.12. The first tactic `iIntros "[H1 H2]"` introduces the hypothesis $\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2$ into the context, and splits it into two hypothesis $\ell_1 \mapsto v_1$ and $\ell_2 \mapsto v_2$. The next two tactics `wp_store` and `wp_load` apply the rules WP-STORE and WP-LOAD, taking care of the evaluation context (using WP-BIND internally) and pure reductions (using WP-PURE).

```
l1, l2 : loc
v1, v2, w1 : val
--------------------------------------
"H1" : l1 ↦ v1
"H2" : l2 ↦ v2
--------------------------------------*
WP #l1 ← w1;; ! #l2 {{ v, ⌜v = v2⌝ }}
```

(a) After executing `iIntros "[H1 H2]"`.

```
l1, l2 : loc
v1, v2, w1 : val
--------------------------------------
"H1" : l1 ↦ w1
"H2" : l2 ↦ v2
--------------------------------------*
WP ! #l2 {{ v, ⌜v = v2⌝ }}
```

(b) After executing `wp_store`.

```
l1, l2 : loc
v1, v2, w1 : val
--------------------------------------
"H1" : l1 ↦ w1
"H2" : l2 ↦ v2
--------------------------------------*
⌜v2 = v2⌝
```

(c) After executing `wp_load`.

Figure 2.12: Interactive proof of lemma `example_wp` in IPM.

As we have seen, the Iris Coq formalization makes it easy to carry out separation logic reasoning with the rigor of machine-checked proofs but without the hassle of all the bookkeeping that is commonly associated with machine-checked proofs. For all the logics in this thesis we develop tactics similar to the tactics used for weakest-precondition proofs. We will not return to this topic in details, except for Section 4.8, where we describe the symbolic execution tactics for ReLoC.

2.6 Defining custom logics in Iris

As we have mentioned, Iris is a separation logic framework that can be used with different program logics and different programming languages. We finish this chapter by describing different ways one could define custom logics on top of Iris.

Program logics for different programming languages. The program logic part of Iris (the weakest precondition calculus in Section 2.2.1) can be instantiated with languages other than HeapLang. This approach has been taken in, *e.g.*, the RustBelt project [Jun+18a] and in the RefinedC refinement type system [Sam+20b].

Alternatively, one can use another programming language by shallowly embedding it in HeapLang. This approach has been taken in *e.g.*, [MJP19] and in Chapter 3. In other chapters of this thesis we work with the standard HeapLang that we have presented in this chapter.

Custom logical connectives. In addition to using the program logic part of Iris with different languages, we need to extend Iris by defining different logical connectives and possibly obtaining a different program logic altogether. New logical connectives can be defined at several layers in the Iris architecture. The layers are as follows:

- The Bunched Implication logic interface, provided by MoSeL [Kre+18]. It provides basic facilities for logics which include the Bunched Implication fragment. Logics defined at this level include iGPS [Kai+17] and Iron [Biz+19].
- The Iris base logic, which defined the standard separation logic connectives ($-\!\!*$, $*$) and modalities ($\triangleright$, $\Box$), as interpreted in the Iris model. At this layer Iris's invariants and ghost state mechanisms are defined [Kre+17]. In Chapter 5 we define a program logic at the level of Iris base logic. Another example of a logic defined at this level include the If-Convergent calculus from [Tim+18].
- The Iris program logic for HeapLang (the weakest precondition calculus) is defined on top of the Iris base logic. On top of the Iris program logic for HeapLang we can define additional logical connectives, as done in Chapters 3 and 4. Other examples of logics defined at this level include the context-local weakest precondition calculus from [TB19] and the Actris logic for verifying message-passing programs [HBK20].

3

λMC: a logic for non-determinsim in C expressions

3.1 Introduction

The ISO C standard [ISO12]—the official specification of the C language—leaves many parts of the language semantics either *unspecified* (*e.g.*, the order of evaluation of expressions), or *undefined* (*e.g.*, dereferencing a NULL pointer or integer overflow). In case of undefined behavior a program may do literally anything, *e.g.*, it may crash, or it may produce an arbitrary result and side-effects. Therefore, to establish the correctness of a C program, one needs to ensure that the program has no undefined behavior for *all* possible choices of non-determinism due to unspecified behavior.

In this paper we focus on the undefined and unspecified behaviors related to C's expression semantics, which have been ignored by most existing verification tools, but are crucial for establishing the correctness of realistic C programs. The C standard does not require subexpressions to be evaluated in a specific order (*e.g.*, from left to right), but rather allows them to be evaluated in *any* order. Moreover, an expression has undefined behavior when there is a conflicting write-write or read-write access to the same location between two *sequence points* [ISO12, 6.5p2] (so called "sequence point violation"). Sequence points occur *e.g.*, at the end of a full expression (`;`), before and after each function call, and after the first operand of a conditional expression (`- ? - : -`) has been evaluated [ISO12, Annex C]. Let us illustrate this by means of the following example:

```
int main() {
  int x; int y = (x = 3) + (x = 4);
  printf("%d␣%d\n", x, y);
}
```

Due to the unspecified evaluation order, one would naively expect this program to print either "`3 7`" or "`4 7`", depending on which assignment to `x` was evaluated first. But this program exhibits undefined behavior due to a sequence point violation: there are two conflicting writes to the variable `x`. Indeed, when compiled with GCC (version 8.2.0), the program in fact prints "`4 8`", which does not correspond to the expected results of any of the evaluation orders.

One may expect that these programs can be easily ruled out statically using some form of static analysis, but this is not the case. Contrary to the simple program above, one can access the values of arbitrary pointers, making it impossible to statically establish the absence of write-write or read-write conflicts. Besides, one should not merely establish the absence of undefined behavior due to conflicting accesses to the same locations, but one should also establish that there are no other forms of undefined behavior (*e.g.*, that no NULL pointers are dereferenced) for *any evaluation order*.

To deal with this issue, Krebbers [Kre14; Kre15] developed a program logic based on Concurrent Separation Logic (CSL) [OHe07] for establishing the absence of undefined behavior in C programs in the presence of non-determinism. To get an impression of how his logic works, let us consider the rule for the addition operator:

$$\frac{\{P_1\}\ \mathrm{e}_1\ \{\Psi_1\} \qquad \{P_2\}\ \mathrm{e}_2\ \{\Psi_2\} \qquad \forall \mathrm{v}_1\ \mathrm{v}_2.\ \Psi_1\ \mathrm{v}_1 * \Psi_2\ \mathrm{v}_2 \vdash \Phi\ (\mathrm{v}_1 + \mathrm{v}_2)}{\{P_1 * P_2\}\ \mathrm{e}_1 + \mathrm{e}_2\ \{\Phi\}}$$

This rule is much like the rule for parallel composition in CSL—the precondition should be separated into two parts P_1 and P_2 describing the resources needed for proving the Hoare triples of both operands. Crucially, since P_1 and P_2 describe disjoint resources as expressed by the *separating conjunction* $*$, it is guaranteed that e_1 and e_2 do not interfere with each other, and hence cannot cause sequence point violations. The purpose of the rule's last premise is to ensure that for all possible return values v_1 and v_2, the postconditions Ψ_1 and Ψ_2 of both operands can be combined into the postcondition Φ of the whole expression.

Krebbers's logic [Kre14; Kre15] has some limitations that impact its usability:

- The rules are not algorithmic, and hence it is not clear how they could be implemented as part of an automated or interactive tool.
- It is difficult to extend the logic with new features. Soundness was proven with respect to a monolithic and ad-hoc model of separation logic.

In this paper we address both of these problems.

We present a new algorithm for symbolic execution in separation logic. Contrary to ordinary symbolic execution in separation logic [BCO05], our symbolic executor takes an expression and a precondition as its input, and computes not only the postcondition, but also simultaneously computes a *frame* that describes the resources that have *not* been used to prove the postcondition. The frame is used to infer the pre- and postconditions of adjacent subexpressions. For example, in $\mathrm{e}_1 + \mathrm{e}_2$, we use the frame of e_1 to symbolically execute e_2.

In order to enable semi-automated reasoning about C programs, we integrate our symbolic executor into a *verification condition generator (vcgen)*. Our vcgen does not merely turn programs into proof goals, but constructs the proof goals only as long as it can discharge goals automatically using our symbolic executor. When an attempt to use the symbolic executor fails, our vcgen will return a new goal, from which the vcgen can be called back again after the user helped out. This approach is useful when integrated into an interactive theorem prover.

We prove soundness of the symbolic executor and verification condition generator with respect to a refined version of the separation logic by Krebbers [Kre14;

Kre15]. Our new logic has been developed on top of the Iris framework [Jun+15; Jun+16; Kre+17; Jun+18b], and thereby inherits all advanced features of Iris (like its expressive support for ghost state and invariants), without having to model these explicitly. To make our new logic better suited for proving the correctness of the symbolic executor and verification condition generator, our new logic comes with a weakest precondition connective instead of Hoare triples as in Krebbers's original logic.

To streamline the soundness proof of our new program logic, we give a new *monadic definitional translation* of a subset of C relevant for non-determinism and sequence points into an ML-style functional language with concurrency. Contrary to the direct style operational semantics for a subset of C by Krebbers [Kre14; Kre15], our approach leads to a semantics that is both easier to understand, and easier to extend with additional language features.

We have mechanized our whole development in the Coq interactive theorem prover. The symbolic executor and verification condition generator are defined as computable functions in Coq, and have been integrated into tactics in the Iris Proof Mode/MoSeL framework [KTB17; Kre+18]. To obtain end-to-end correctness, we mechanized the proofs of soundness of our symbolic executor and verification condition generator with respect to our new separation logic and new monadic definitional semantics for a subset of C. The Coq development is available at [FGK19b].

Contributions. We describe an approach to semi-automatically prove the absence of undefined behavior in a given C program for *any* evaluation order. While doing so, we make the following contributions:

- We define λMC: a small C-style language with a semantics by a monadic translation into an ML-style functional language with concurrency (Section 3.2);
- We present a separation logic with weakest preconditions for λMC based on the separation logic for non-determinism in C by Krebbers [Kre14; Kre15] (Section 3.3);
- We prove soundness of our separation logic with weakest preconditions by giving a modular model using the Iris framework [Jun+15; Jun+16; Kre+17; Jun+18b] (Section 3.4);
- We present a new symbolic executor that not only computes the postcondition of a C expression, but also a *frame*, used to determine how resources should be distributed among subexpressions (Section 3.5);
- On top of our symbolic executor, we define a verification condition generator that enables semi-automated proofs using an interactive theorem prover (Section 3.6);
- We demonstrate that our approach can be implemented and proved sound using Coq for a superset of the λMC language considered in this paper (Section 3.7).

3.2 λMC: A monadic definitional semantics of C

In this section we describe a small C-style language called λMC, which features non-determinism in expressions. We define its semantics by translation into a ML-style functional language with concurrency called HeapLang.

We briefly describe the λMC source language (Section 3.2.1) and the HeapLang target language (Section 3.2.2) of the translation. Then we describe the translation scheme itself (Section 3.2.3). We explain in several steps how to exploit concurrency and monadic programming to give a concise and clear definitional semantics.

3.2.1 The source language λMC

The syntax of our source language called λMC is as follows:

$$
\begin{aligned}
v \in \mathsf{val} &::= z \mid \mathtt{f} \mid l \mid \mathtt{NULL} \mid (v_1, v_2) \mid () && (z \in \mathbb{Z},\ l \in \mathsf{Loc}) \\
e \in \mathsf{expr} &::= v \mid \mathtt{x} \mid (e_1, e_2) \mid e.1 \mid e.2 \mid e_1 \circledcirc e_2 \mid && (\circledcirc \in \{+, -, \ldots\}) \\
&\quad \mathtt{x} \leftarrow e_1; e_2 \mid \mathtt{if}(e_1)\{e_2\}\{e_3\} \mid \mathtt{while}(e_1)\{e_2\} \mid e_1(e_2) \mid \\
&\quad \mathtt{alloc}(e) \mid \mathtt{*}e \mid e_1 \mathtt{=} e_2 \mid \mathtt{free}(e)
\end{aligned}
$$

The values include integers, `NULL` pointers, concrete locations l, function pointers `f`, structs with two fields (tuples), and the unit value `()` (for functions without return value). There is a global list of function definitions, where each definition is of the form `f(x){e}`. Most of the expression constructs resemble standard C notation, with some exceptions. We do not differentiate between expressions and statements to keep our language uniform. As such, if-then-else and sequencing constructs are not duplicated for both expressions and statements. Moreover, we do not differentiate between *lvalues* and *rvalues* [ISO12, p. 6.3.2.1]. Hence, there is no address operator `&`, and, similarly to ML, the load (`*e`) and assignment ($e_1 = e_2$) operators take a reference as their first argument.

The *sequenced bind* operator $\mathtt{x} \leftarrow e_1; e_2$ generalizes the normal sequencing operator $e_1; e_2$ of C by binding the result of e_1 to the variable `x` in e_2. As such, $\mathtt{x} \leftarrow e_1; e_2$ can be thought of as the declaration of an immutable local variable `x`. We omit mutable local variables for now, but these can be easily added as an extension to our method, as shown in Section 3.7. We write $e_1; e_2$ for a sequenced bind $\_ \leftarrow e_1; e_2$ in which we do not care about the return value of e_1.

To focus on the key topics of the paper—non-determinism and the sequence point restriction—we take a minimalistic approach and omit most other features of C. Notably, we omit non-local control (return, break, continue, and goto). Our memory model is simplified; it only supports structs with two fields (tuples), but no arrays, unions, or machine integers. In Section 3.7 we show that some of these features (arrays, pointer arithmetic, and mutable local variables) can be incorporated.

3.2.2 The target language HeapLang

The target language of our definitional semantics of λMC is concurrent ML-style programming language HeapLang (defined in more details in Section 2.1) with

locks/mutexes (as in Section 2.4.2) and parallel composition (as in Section 2.4.1). The syntax of HeapLang is as follows:

$$v \in Val ::= z \mid \textbf{true} \mid \textbf{false} \mid \text{rec } f\ x = e \mid \ell \mid () \mid \ldots \qquad (z \in \mathbb{Z}, \ell \in Loc)$$
$$e \in Expr ::= v \mid x \mid e_1\ e_2 \mid \text{ref}(e) \mid\ !e \mid e_1 \leftarrow e_2 \mid \textbf{assert}(e) \mid$$
$$e_1 \parallel e_2 \mid \text{newlock} \mid \text{acquire} \mid \text{release} \mid \ldots$$

The language contains some concurrency primitives that we will use to model non-determinism in λMC. Those primitives are (||), newlock, acquire, and release. The first primitive is the parallel composition operator, which executes expressions e_1 and e_2 in parallel, and returns a tuple of their results.

The expression newlock () creates a new mutex. If *lk* is a mutex that was created this way, then acquire *lk* tries to acquire it and blocks until no other thread is using *lk*. An acquired mutex can be released using release *lk*.

In addition, we add an operation **assert**(*e*) that reduces to () if *e* reduces to **true**, and gets stuck otherwise. The definition of the **assert** operation and the associated proof rule will be presented in Section 3.4.1.

3.2.3 The monadic definitional semantics of λMC

We now give the semantics of λMC by translation into HeapLang. The translation is carried out in several stages, each iteration implementing and illustrating a specific aspect of C. First, we model non-determinism in expressions by concurrency, parallelizing execution of subexpressions (step 1). After that, we add checks for sequence point violations in the translation of the assignment and dereferencing operations (step 2). Finally, we add function calls and demonstrate how the translation can be simplified using a monadic notation (step 3).

Step 1: Non-determinism via parallel composition.

We model the unspecified evaluation order in binary expressions like $\mathrm{e}_1 + \mathrm{e}_2$ and $\mathrm{e}_1 = \mathrm{e}_2$ by executing the subexpressions in parallel using the (||) operator:

$$[\![\mathrm{e}_1 + \mathrm{e}_2]\!] \triangleq \text{let } (v_1, v_2) = [\![\mathrm{e}_1]\!] \parallel [\![\mathrm{e}_2]\!] \text{ in } v_1 + v_2$$
$$[\![\mathrm{e}_1 = \mathrm{e}_2]\!] \triangleq \text{let } (v_1, v_2) = [\![\mathrm{e}_1]\!] \parallel [\![\mathrm{e}_2]\!] \text{ in}$$
$$\text{match } v_1 \text{ with}$$
$$\mid \textbf{None} \to \textbf{assert}(\textbf{false}) \quad \texttt{(* NULL pointer *)}$$
$$\mid \textbf{Some}\ l \to \text{match } !l \text{ with}$$
$$\mid \textbf{None} \to \textbf{assert}(\textbf{false}) \quad \texttt{(* Use after free *)}$$
$$\mid \textbf{Some}\ \_ \to l \leftarrow \textbf{Some}\ v_2;\ v_2$$

Since our memory model is simple, the value interpretation is straightforward:

$$[\![\mathrm{z}]\!]_\mathcal{V} \triangleq \mathrm{z} \quad (\text{if } \mathrm{z} \in \mathbb{Z}) \qquad [\![\mathtt{NULL}]\!]_\mathcal{V} \triangleq \textbf{None}$$
$$[\![(\mathrm{v}_1, \mathrm{v}_2)]\!]_\mathcal{V} \triangleq ([\![\mathrm{v}_1]\!]_\mathcal{V}, [\![\mathrm{v}_2]\!]_\mathcal{V}) \qquad [\![()]\!]_\mathcal{V} \triangleq () \qquad [\![1]\!]_\mathcal{V} \triangleq \textbf{Some}\ 1$$

The only interesting case is the translation of locations. Since there is no concept of a `NULL` pointer in HeapLang, we use the option type to distinguish `NULL` pointers from concrete locations (1). The interpretation of assignments thus contains a pattern match to check that no `NULL` pointers are dereferenced. A similar check is performed in the interpretation of the load operation (`*e`). Moreover, each location contains an option to distinguish freed from active locations.

Step 2: Sequence points.

So far we have not accounted for undefined behavior due to sequence point violations. For instance, the program `(x = 3) + (x = 4)` gets translated into a HeapLang expression that updates the value of the location `x` non-deterministically to either 3 or 4, and returns 7. However, in C, the behavior of this program is *undefined*, as it exhibits a sequence point violation: there is a write conflict for the location `x`.

To give a semantics for sequence point violations, we follow the approach by Norrish [Nor98], Ellison and Rosu [ER12], and Krebbers [Kre14; Kre15]. We keep track of a set of locations that have been written to since the last sequence point. We refer to this set as the *environment* of our translation, and represent it using a global variable *env* of the type `mset` *Loc*. Because our target language HeapLang is concurrent, all updates to the environment *env* must be executed *atomically*, *i.e.*, inside a critical section. We enforce this behavior by employing a global mutex *lk*. The interpretation of assignments $e_1 = e_2$ now becomes:

$$
\begin{aligned}
[\![e_1 = e_2]\!] \triangleq\ & \mathsf{let}\ (v_1, v_2) = [\![e_1]\!] \parallel [\![e_2]\!]\ \mathsf{in} \\
& \quad \mathsf{acquire}\ lk; \\
& \quad \mathsf{match}\ v_1\ \mathsf{with} \\
& \quad \mid \mathsf{None} \rightarrow \mathsf{assert}(\mathsf{false}) \qquad \texttt{(* NULL pointer *)} \\
& \quad \mid \mathsf{Some}\ l \rightarrow \\
& \qquad \mathsf{assert}(\neg \mathsf{mset\_member}\ l\ env);\ \texttt{(* Seq. point violation *)} \\
& \qquad \mathsf{match}\ !l\ \mathsf{with} \\
& \qquad \mid \mathsf{None} \rightarrow \mathsf{assert}(\mathsf{false}) \quad \texttt{(* Use after free *)} \\
& \qquad \mid \mathsf{Some}\ \_ \rightarrow \mathsf{mset\_add}\ l\ env;\ l \leftarrow \mathsf{Some}\ v_2; \\
& \quad \mathsf{release}\ lk;\ v_2
\end{aligned}
$$

Whenever we assign to (or read from) a location l, we check if the location l is not already present in the environment *env*. If the location l is present, then it was already written to since the last sequence point. Hence, accessing the location constitutes undefined behavior (see the **assert** in the interpretation of assignments above). In the interpretation of assignments, we furthermore insert the location l into the environment *env*.

In order to make sure that one can access a variable again after a sequence point, we define the *sequenced bind* operator $x \leftarrow e_1; e_2$ as follows:

$$[\![x \leftarrow e_1; e_2]\!] \triangleq \mathsf{let}\ x = [\![e_1]\!]\ \mathsf{in}\ \mathsf{acquire}\ lk;\ \mathsf{mset\_clear}\ env;\ \mathsf{release}\ lk;\ [\![e_2]\!]$$

$$\texttt{ret}\ e \triangleq \lambda\_\ \_.\ e$$
$$e_1 \parallel^{\mathsf{m}} e_2 \triangleq \lambda env\ lk.\ (e_1\ env\ lk) \parallel (e_2\ env\ lk)$$
$$x \xleftarrow{\mathsf{m}} e_1;\ e_2 \triangleq \lambda env\ lk.\ \mathsf{let}\ x = e_1\ env\ lk\ \mathsf{in}\ e_2\ env\ lk$$
$$\mathsf{atomic\_env}\ e \triangleq \lambda env\ lk.\ \mathsf{acquire}\ lk; \mathsf{let}\ a = e\ env\ \mathsf{in}\ \mathsf{release}\ lk;\ a$$
$$\mathsf{atomic}\ e \triangleq \lambda env\ lk.\ \mathsf{acquire}\ lk; \mathsf{let}\ a = e\ env\ (\mathsf{newlock}\ ())\ \mathsf{in}\ \mathsf{release}\ lk;\ a$$
$$\mathsf{run}(e) \triangleq e\ (\mathsf{mset\_create}\ ())\ (\mathsf{newlock}\ ())$$

Figure 3.1: The monadic combinators.

After we finished executing the expression e_1, we clear the environment *env*, so that all locations are accessible in e_2 again.

Step 3: Non-interleaved function calls.

As the final step, we present the correct translation scheme for function calls. Unlike the other expressions, function calls are not interleaved during the execution of subexpressions [ISO12, 6.5.2.2p10]. For instance, in the program `f() + g()` the possible orders of execution are: either all the instructions in `f()` followed by all the instructions in `g()`, or all the instructions in `g()` followed by all the instructions in `f()`.

To model this, we execute each function call *atomically*. In the previous step we used a global mutex for guarding the access to the environment. We could use that mutex for function calls too. However, reusing a single mutex for entering each critical section would not work because a body of a function may contain invocations of other functions. To that extent, we use multiple mutexes to reflect the hierarchical structure of function calls.

To handle multiple mutexes, each C expression is interpreted as a HeapLang function that receives a mutex and returns its result. That is, each C expression is modeled by a monadic expression in the *reader monad* $M(A) \triangleq \texttt{mset}\ Loc \to \texttt{mutex} \to A$. For consistency's sake, we now also use the monad to thread through the reference to the environment (`mset` *Loc*), instead of using a global variable *env* as we did in the previous step.

We use a small set of monadic combinators, shown in Figure 3.1, to build the translation in a more abstract way.[1] The return and bind operators are standard for the reader monad. The parallel operator runs two monadic expressions concurrently, propagating the environment and the mutex. The atomic combinator invokes a monadic expression with a fresh mutex. The atomic_env combinator atomically executes its body with the current environment as an argument. The run function

[1] Essentially, we are using the reader monad to model reentrant atomic blocks in the absence of reentrant locks.

$$
\begin{aligned}
[\![\mathrm{e}_1 + \mathrm{e}_2]\!] &\triangleq (v_1, v_2) \xleftarrow{\mathsf{m}} ([\![\mathrm{e}_1]\!] \,||^{\mathsf{m}}\, [\![\mathrm{e}_2]\!]);\ \texttt{ret}\ (v_1 + v_2) \\
[\![*\mathrm{e}]\!] &\triangleq v \xleftarrow{\mathsf{m}} [\![\mathrm{e}]\!]; \\
&\quad \mathsf{atomic\_env}\ (\lambda env. \\
&\quad\quad \mathsf{match}\ v\ \mathsf{with} \\
&\quad\quad\ |\ \mathsf{None} \to \mathsf{assert(false)} \quad \texttt{(* NULL pointer *)} \\
&\quad\quad\ |\ \mathsf{Some}\ l \to \\
&\quad\quad\quad \texttt{(* Check for a seq. point violation *)} \\
&\quad\quad\quad \mathsf{assert}(\neg \mathsf{mset\_member}\ l\ env); \\
&\quad\quad\quad \mathsf{match}\ !l\ \mathsf{with} \\
&\quad\quad\quad\ |\ \mathsf{None} \to \mathsf{assert(false)} \quad \texttt{(* Use after free *)} \\
&\quad\quad\quad\ |\ \mathsf{Some}\ w \to \texttt{ret}\ w) \\
[\![\mathrm{e}_1 = \mathrm{e}_2]\!] &\triangleq (v_1, v_2) \xleftarrow{\mathsf{m}} ([\![\mathrm{e}_1]\!] \,||^{\mathsf{m}}\, [\![\mathrm{e}_2]\!]); \\
&\quad \mathsf{atomic\_env}\ (\lambda env. \\
&\quad\quad \mathsf{match}\ v_1\ \mathsf{with} \\
&\quad\quad\ |\ \mathsf{None} \to \mathsf{assert(false)} \quad \texttt{(* NULL pointer *)} \\
&\quad\quad\ |\ \mathsf{Some}\ l \to \\
&\quad\quad\quad \texttt{(* Check for a seq. point violation *)} \\
&\quad\quad\quad \mathsf{assert}(\neg \mathsf{mset\_member}\ l\ env); \\
&\quad\quad\quad \mathsf{match}\ !l\ \mathsf{with} \\
&\quad\quad\quad\ |\ \mathsf{None} \to \mathsf{assert(false)} \quad \texttt{(* Use after free *)} \\
&\quad\quad\quad\ |\ \mathsf{Some}\ \_ \to \mathsf{mset\_add}\ l\ env;\ l \leftarrow \mathsf{Some}\ v_2;\ \texttt{ret}\ v_2) \\
[\![\mathrm{x} \leftarrow \mathrm{e}_1; \mathrm{e}_2]\!] &\triangleq x \xleftarrow{\mathsf{m}} [\![\mathrm{e}_1]\!]; \\
&\quad \_ \xleftarrow{\mathsf{m}} (\mathsf{atomic\_env}\ \ \mathsf{mset\_clear}); \\
&\quad [\![\mathrm{e}_2]\!] \\
[\![\texttt{if}(\mathrm{e}_1)\{\mathrm{e}_2\}\{\mathrm{e}_3\}]\!] &\triangleq x \xleftarrow{\mathsf{m}} [\![\mathrm{e}_1]\!]; \\
&\quad \_ \xleftarrow{\mathsf{m}} (\mathsf{atomic\_env}\ \ \mathsf{mset\_clear}); \\
&\quad \mathsf{if}\ x\ \mathsf{then}\ [\![\mathrm{e}_2]\!]\ \mathsf{else}\ [\![\mathrm{e}_3]\!] \\
[\![\mathrm{e}_1(\mathrm{e}_2)]\!] &\triangleq (f, a) \xleftarrow{\mathsf{m}} ([\![\mathrm{e}_1]\!] \,||^{\mathsf{m}}\, [\![\mathrm{e}_2]\!]); \\
&\quad \mathsf{atomic}\ (\mathsf{atomic\_env}\ \ \mathsf{mset\_clear};\ f\ a) \\
[\![\mathrm{f}(\mathrm{x})\{\mathrm{e}\}]\!] &\triangleq \mathsf{let\ rec}\ \mathrm{f}\ \mathrm{x} = v \xleftarrow{\mathsf{m}} [\![\mathrm{e}]\!];\ \_ \xleftarrow{\mathsf{m}} (\mathsf{atomic\_env}\ \ \mathsf{mset\_clear});\ \ \texttt{ret}\ v
\end{aligned}
$$

Figure 3.2: Selected clauses from the monadic definitional semantics.

executes the monadic computation by instantiating it with a fresh mutex and a new environment.

Selected clauses for the translation are presented in Figure 3.2. The translation of the binary operations remains virtually unchanged, except for the usage of monadic parallel composition instead of the standard one. The translation for the assignment and the sequenced bind uses the atomic_env combinator for querying and updating the environment. We also have to adapt our translation of values, by wrapping it in $\texttt{ret} : [\![\mathsf{v}]\!] \triangleq \texttt{ret}\ [\![\mathsf{v}]\!]_{\mathcal{V}}$.

A global function definition `f(x){e}` is translated as a top level let-binding. A function call is then just an atomically executed function invocation in HeapLang, modulo the fact that the function pointer and the arguments are computed in parallel. In addition, sequence points occur at the beginning of each function call and at the end of each function body [ISO12, Annex C], and we reflect that in our translation by clearing the environment at appropriate places.

Our semantics by translation can easily be extended to cover other features of C, *e.g.*, a more advanced memory model (see Section 3.7). However the fragment presented here already illustrates the challenges that non-determinism and sequence point violations pose for verification. In the next section we describe a logic for reasoning about the semantics by translation given in this section.

3.3 Separation logic with weakest preconditions for λMC

In this section we present a separation logic with weakest precondition propositions for reasoning about λMC programs. The logic tackles the main features of our semantics—non-determinism in expressions evaluation and sequence point violations. We will discuss the high-level rules of the logic pertaining to C connectives by going through a series of small examples.

The logic presented here is similar to the separation logic by Krebbers [Kre14], but it is given in a weakest precondition style, and moreover, it is constructed *synthetically* on top of the separation logic framework Iris [Jun+15; Jun+16; Kre+17; Jun+18b], whereas the logic by Krebbers [Kre14] is interpreted directly in a bespoke model.

The following grammar defines the formulas of the logic:

$$\begin{aligned} P, Q \in \mathit{iProp} ::= {} & \mathsf{True} \mid \mathsf{False} \mid \forall x.\, P \mid \exists x.\, P \mid \mathsf{v}_1 = \mathsf{v}_2 \mid 1 \overset{q}{\hookrightarrow}_{\xi} v \qquad (q \in (0,1],\ \xi \in \{L, U\}) \\ & \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \mathbb{U} P \mid \mathsf{cwp}\ e\ \{\Phi\} \mid \ldots \end{aligned}$$

Most of the connectives are commonplace in separation logic, with the exception of the modified points-to connective, which we describe in this section.

The weakest precondition connective cwp e {Φ} states that the program e is safe (the program has defined behavior), and if e terminates to a value v, then v satisfies the predicate Φ. We write cwp e {v. Φ v} for cwp e {λv. Φ v}. As is common, Hoare triples {*P*} e {Φ} are syntactic sugar for $P \vdash \mathsf{cwp}\ e\ \{\Phi\}$.

Contrary to the paper by Krebbers [Kre14], we use weakest preconditions instead of Hoare triples throughout this paper. There are several reasons for doing so:

1. We do not have to manipulate the preconditions explicitly, *e.g.*, by applying the consequence rule to the precondition.

$$\textsc{cwp-value}\ \frac{\Phi\ \mathsf{v}}{\mathsf{cwp}\ \mathsf{v}\ \{\Phi\}} \qquad \textsc{cwp-wand}\ \frac{\mathsf{cwp}\ \mathsf{e}\ \{\Phi\} \qquad (\forall \mathsf{v}.\ \Phi\ \mathsf{v} \mathrel{-\!\!*} \Psi\ \mathsf{v})}{\mathsf{cwp}\ \mathsf{e}\ \{\Psi\}} \qquad \textsc{cwp-seq}\ \frac{\mathsf{cwp}\ \mathsf{e}_1\ \{\mathsf{v}.\ \mathbb{U}(\mathsf{cwp}\ \mathsf{e}_2[\mathsf{v}/\mathsf{x}]\ \{\Phi\})\}}{\mathsf{cwp}\ (\mathsf{x} \leftarrow \mathsf{e}_1; \mathsf{e}_2)\ \{\Phi\}}$$

$$\textsc{cwp-bin-op}\ \frac{\mathsf{cwp}\ \mathsf{e}_1\ \{\Psi_1\} \qquad \mathsf{cwp}\ \mathsf{e}_2\ \{\Psi_2\} \qquad (\forall \mathsf{w}_1 \mathsf{w}_2.\ \Psi_1\ \mathsf{w}_1 * \Psi_2\ \mathsf{w}_2 \mathrel{-\!\!*} \Phi(\mathsf{w}_1\ [\![\circledcirc]\!]\ \mathsf{w}_2))}{\mathsf{cwp}\ (\mathsf{e}_1 \circledcirc \mathsf{e}_2)\ \{\Phi\}}$$

$$\textsc{cwp-load}\ \frac{\mathsf{cwp}\ \mathsf{e}\ \left\{1.\ \exists \mathsf{w}\ q.\ 1 \overset{q}{\hookrightarrow}_U \mathsf{w} * (1 \overset{q}{\hookrightarrow}_U \mathsf{w} \mathrel{-\!\!*} \Phi\ \mathsf{w})\right\}}{\mathsf{cwp}\ (*\mathsf{e})\ \{\Phi\}} \qquad \textsc{cwp-alloc}\ \frac{\mathsf{cwp}\ \mathsf{e}\ \{\mathsf{v}.\ \forall 1.\ 1 \hookrightarrow_U \mathsf{v} \mathrel{-\!\!*} \Phi\ 1\}}{\mathsf{cwp}\ \mathtt{alloc}(\mathsf{e})\ \{\Phi\}}$$

$$\textsc{cwp-store}\ \frac{\begin{array}{c}\mathsf{cwp}\ \mathsf{e}_1\ \{\Psi_1\} \qquad \mathsf{cwp}\ \mathsf{e}_2\ \{\Psi_2\} \\ (\forall 1\,\mathsf{w}.\ \Psi_1\ 1 * \Psi_2\ \mathsf{w} \mathrel{-\!\!*} \exists \mathsf{v}.\ 1 \hookrightarrow_U \mathsf{v} * (1 \hookrightarrow_L \mathsf{w} \mathrel{-\!\!*} \Phi\ \mathsf{w}))\end{array}}{\mathsf{cwp}\ (\mathsf{e}_1 = \mathsf{e}_2)\ \{\Phi\}} \qquad \textsc{cwp-free}\ \frac{\mathsf{cwp}\ \mathsf{e}\ \{1.\ \exists \mathsf{v}.\ 1 \hookrightarrow_U \mathsf{v} * \Phi\ ()\}}{\mathsf{cwp}\ \mathtt{free}(\mathsf{e})\ \{\Phi\}}$$

$$\textsc{cmapsto-split}\ 1 \overset{q_1}{\hookrightarrow}_{\xi_1} \mathsf{v} * 1 \overset{q_2}{\hookrightarrow}_{\xi_2} \mathsf{v} \dashv\vdash 1 \overset{q_1+q_2}{\hookrightarrow}_{\xi_1 \vee \xi_2} \mathsf{v} \qquad \textsc{cmapsto-values-agree}\ \frac{1 \overset{q_1}{\hookrightarrow}_{\xi_1} \mathsf{v}_1 \qquad 1 \overset{q_2}{\hookrightarrow}_{\xi_2} \mathsf{v}_2}{\mathsf{v}_1 = \mathsf{v}_2} \qquad \textsc{u-unlock}\ \frac{1 \overset{q}{\hookrightarrow}_L \mathsf{v}}{\mathbb{U}(1 \overset{q}{\hookrightarrow}_U \mathsf{v})}$$

$$\textsc{u-mono}\ \frac{P \mathrel{-\!\!*} Q}{\mathbb{U}P \mathrel{-\!\!*} \mathbb{U}Q} \qquad \textsc{u-intro}\ \frac{P}{\mathbb{U}P} \qquad \textsc{u-sep}\ \frac{\mathbb{U}P * \mathbb{U}Q}{\mathbb{U}(P * Q)}$$

Figure 3.3: Selected rules.

2. The soundness of our symbolic executor (Theorem 3.3) can be stated more concisely using weakest precondition propositions.
3. It is more convenient to integrate weakest preconditions into the Iris Proof Mode/MoSeL framework in Coq that we use for our implementation (Section 3.7).

A selection of rules is presented in Figure 3.3. Each inference rule $\frac{P_1 \ \ldots \ P_n}{Q}$ in this paper should be read as the entailment $P_1 * \ldots * P_n \vdash Q$. We now explain and motivate the rules of our logic.

3.3.1 Non-determinism

In the introduction (Section 3.1) we have already shown the rule for addition from Krebbers's logic [Kre14], which was written using Hoare triples. Using weakest

preconditions, the corresponding rule (CWP-BIN-OP) is:

$$\frac{\mathsf{cwp}\ \mathsf{e}_1\ \{\Psi_1\} \qquad \mathsf{cwp}\ \mathsf{e}_2\ \{\Psi_2\} \qquad (\forall \mathsf{w}_1\,\mathsf{w}_2.\ \Psi_1\ \mathsf{w}_1 * \Psi_2\ \mathsf{w}_2 \mathrel{-\!\!*} \Phi(\mathsf{w}_1\ [\![\circledcirc]\!]\ \mathsf{w}_2))}{\mathsf{cwp}\ (\mathsf{e}_1 \circledcirc \mathsf{e}_2)\ \{\Phi\}}$$

This rule closely resembles the usual rule for parallel composition in ordinary concurrent separation logic [OHe07]. This should not be surprising, as we have given a definitional semantics to binary operators using the parallel composition operator. It is important to note that the premises CWP-BIN-OP are combined using the *separating conjunction* $*$. This ensures that the weakest preconditions $\mathsf{cwp}\ \mathsf{e}_1\ \{\Psi_1\}$ and $\mathsf{cwp}\ \mathsf{e}_2\ \{\Psi_2\}$ for the subexpressions e_1 and e_2 are verified with respect to disjoint resources. As such they do not interfere with each other, and can be evaluated in parallel without causing sequence point violations.

To see how one can use the rule CWP-BIN-OP, let us verify $P \vdash \mathsf{cwp}\ (\mathsf{e}_1 + \mathsf{e}_2)\ \{\Phi\}$. That is, we want to show that $(\mathsf{e}_1 + \mathsf{e}_2)$ satisfies the postcondition Φ assuming the precondition P. This goal can be proven by separating the precondition P into disjoint parts $P_1 * P_2 * R \dashv\vdash P$. Then using CWP-BIN-OP the goal can be reduced to proving $P_i \vdash \mathsf{cwp}\ \mathsf{e}_i\ \{\Psi_i\}$ for $i \in \{0,1\}$, and $R * \Psi_1\ \mathsf{w}_1 * \Psi_2\ \mathsf{w}_2 \vdash \Phi\ (\mathsf{w}_1\ [\![\circledcirc]\!]\ \mathsf{w}_2)$ for any return values w_i of the expressions e_i.

3.3.2 Fractional permissions

Separation logic includes the *points-to connective* $\mathsf{l} \hookrightarrow \mathsf{v}$, which asserts unique ownership of a location l with value v. This connective is used to specify the behavior of stateful operations, which becomes apparent in the following proposed rule for load:

$$\frac{\mathsf{cwp}\ \mathsf{e}\ \{\mathsf{l}.\ \exists \mathsf{w}.\ \mathsf{l} \hookrightarrow \mathsf{w} * (\mathsf{l} \hookrightarrow \mathsf{w} \mathrel{-\!\!*} \Phi\ \mathsf{w})\}}{\mathsf{cwp}\ (*\mathsf{e})\ \{\Phi\}}$$

In order to verify $*\mathsf{e}$ we first make sure that e evaluates to a location l, and then we need to provide the points-to connective $\mathsf{l} \hookrightarrow \mathsf{w}$ for some value stored at the location. This rule, together with CWP-VALUE, allows for verification of simple programs like $\mathsf{l} \hookrightarrow \mathsf{v} \vdash \mathsf{cwp}\ (*\mathsf{l})\ \{\mathsf{w}.\ \mathsf{w} = \mathsf{v} * \mathsf{l} \hookrightarrow \mathsf{v}\}$.

However, the rule above is too weak. Suppose that we wish to verify the program $*\mathsf{l} + *\mathsf{l}$ from the precondition $\mathsf{l} \hookrightarrow \mathsf{v}$. According to CWP-BIN-OP, we have to separate the proposition $\mathsf{l} \hookrightarrow \mathsf{v}$ into two disjoint parts, each used to verify the load operation. In order to enable sharing of points-to connectives we use *fractional permissions* [Boy03; Bor+05]. In separation logic with fractional permissions each points-to connective is annotated with a fraction $q \in (0,1]$, and the resources can be split in accordance with those fractions:

$$\mathsf{l} \overset{q_1+q_2}{\hookrightarrow} \mathsf{v} \dashv\vdash \mathsf{l} \overset{q_1}{\hookrightarrow} \mathsf{v} * \mathsf{l} \overset{q_2}{\hookrightarrow} \mathsf{v}.$$

A connective $\mathsf{l} \overset{1}{\hookrightarrow} \mathsf{v}$ provides a unique ownership of the location, and we refer to it as a *write permission*. A points-to connective with $q \leq 1$ provides shared ownership of the location, referred to as a *read permission*. By convention, we write $\mathsf{l} \hookrightarrow \mathsf{v}$ to denote the write permission $\mathsf{l} \overset{1}{\hookrightarrow} \mathsf{v}$.

With fractional permissions at hand, we can relax the proposed load rule, by allowing to dereference a location even if we only have a read permission:

$$\frac{\mathsf{cwp}\ \mathsf{e}\ \left\{1.\ \exists \mathsf{w}\, q.\ 1 \overset{q}{\hookrightarrow} \mathsf{w} * (1 \overset{q}{\hookrightarrow} \mathsf{w} \mathrel{-\!\!*} \Phi\ \mathsf{w})\right\}}{\mathsf{cwp}\ (*\mathsf{e})\ \{\Phi\}}$$

This corresponds to the intuition that multiple subexpressions can safely dereference the same location, but not write to them.

Using the rule above we can verify $1 \hookrightarrow 1 \vdash \mathsf{cwp}\ (*1 + *1)\ \{\mathsf{v}.\ \mathsf{v} = 2 * 1 \hookrightarrow 1\}$ by splitting the assumption into $1 \overset{0.5}{\hookrightarrow} 1 * 1 \overset{0.5}{\hookrightarrow} 1$ and first applying CWP-BIN-OP with Ψ_1 and Ψ_2 being $\lambda \mathsf{v}.\ (\mathsf{v} = 1) * 1 \overset{0.5}{\hookrightarrow} 1$. Then we apply CWP-LOAD on both subgoals. After that, we can use CMAPSTO-SPLIT to prove the remaining formula:

$$(\mathsf{v}_1 = 1) * 1 \overset{0.5}{\hookrightarrow} 1 * (\mathsf{v}_2 = 1) * 1 \overset{0.5}{\hookrightarrow} 1 \ \vdash\ (\mathsf{v}_1 + \mathsf{v}_2 = 2) * 1 \hookrightarrow 1.$$

3.3.3 The assignment operator

The second main operation that accesses the heap is the assignment operator $\mathsf{e}_1 = \mathsf{e}_2$. The arguments on the both sides of the assignment are evaluated in parallel, and a points-to connective is required to perform an update to the heap. A naive version of the assignment rule can be obtained by combining the binary operation rule and the load rule:

$$\frac{\mathsf{cwp}\ \mathsf{e}_1\ \{\Psi_1\} \qquad \mathsf{cwp}\ \mathsf{e}_2\ \{\Psi_2\} \qquad (\forall 1\, \mathsf{w}.\ \Psi_1\ 1 * \Psi_2\ \mathsf{w} \mathrel{-\!\!*} \exists \mathsf{v}.\ 1 \hookrightarrow \mathsf{v} * (1 \hookrightarrow \mathsf{w} \mathrel{-\!\!*} \Phi\ \mathsf{w}))}{\mathsf{cwp}\ (\mathsf{e}_1 = \mathsf{e}_2)\ \{\Phi\}}$$

The write permission $1 \hookrightarrow \mathsf{v}$ can be obtained by combining the resources of both sides of the assignment. This allows us to verify programs like $1 = *1 + *1$.

However, the rule above is unsound, because it fails to account for sequence point violations. We could use the rule above to prove safety of undefined programs, *e.g.*, the program $1 = (1 = 3)$.

To account for sequence point violations we decorate the points-to connectives $1 \overset{q}{\hookrightarrow}_{\xi} \mathsf{v}$ with *access levels* $\xi \in \{L, U\}$. These have the following semantics: we can read from and write to a location that is unlocked (U), and the location becomes locked (L) once someone writes to it. Proposition $1 \overset{q}{\hookrightarrow}_U \mathsf{v}$ (resp. $1 \overset{q}{\hookrightarrow}_L \mathsf{v}$) asserts ownership of the unlocked (resp. locked) location 1. We refer to such propositions as *lockable points-to connectives*. Using lockable points-to connectives we can formulate the correct assignment rule:

$$\frac{\mathsf{cwp}\ \mathsf{e}_1\ \{\Psi_1\} \qquad \mathsf{cwp}\ \mathsf{e}_2\ \{\Psi_2\} \qquad (\forall 1\, \mathsf{w}.\ \Psi_1\ 1 * \Psi_2\ \mathsf{w} \mathrel{-\!\!*} \exists \mathsf{v}.\ 1 \hookrightarrow_U \mathsf{v} * (1 \hookrightarrow_L \mathsf{w} \mathrel{-\!\!*} \Phi\ \mathsf{w}))}{\mathsf{cwp}\ (\mathsf{e}_1 = \mathsf{e}_2)\ \{\Phi\}}$$

The set $\{L, U\}$ has a lattice structure with $L \leq U$, and the levels can be combined with a join operation, see CMAPSTO-SPLIT. By convention, $1 \overset{q}{\hookrightarrow} \mathsf{v}$ denotes $1 \overset{q}{\hookrightarrow}_U \mathsf{v}$.

3.3.4 The unlocking modality

As locations become locked after using the assignment rule, we wish to unlock them in order to perform further heap operations. For instance, in the expression $1 = 4; *1$ the location 1 becomes unlocked after the sequence point ";" between the store and the dereferencing operations. To reflect this in the logic, we use the rule CWP-SEQ which features the *unlocking modality* $\mathbb{U}$ (which is called the unlocking assertion in [Kre14, Definition 5.6]):

$$\frac{\mathsf{cwp}\ \mathsf{e}_1\ \{\_.\ \mathbb{U}(\mathsf{cwp}\ \mathsf{e}_2\ \{\Phi\})\}}{\mathsf{cwp}\ (\mathsf{e}_1; \mathsf{e}_2)\ \{\Phi\}}$$

Intuitively, $\mathbb{U}P$ states that P holds, after unlocking all locations. The rules of $\mathbb{U}$ in Figure 3.3 allow one to turn a sequent

$$(P_1 * \ldots * P_m) * (\mathsf{l}_1 \hookrightarrow_L \mathsf{v}_1 * \ldots * \mathsf{l}_m \hookrightarrow_L \mathsf{v}_m) \vdash \mathbb{U} Q$$

into

$$(P_1 * \ldots * P_m) * (\mathsf{l}_1 \hookrightarrow_U \mathsf{v}_1 * \ldots * \mathsf{l}_m \hookrightarrow_U \mathsf{v}_m) \vdash Q.$$

This is done by applying either U-UNLOCK or U-INTRO to each premise; then collecting all premises into one formula under $\mathbb{U}$ by U-SEP; and finally, applying U-MONO to the whole sequent.

3.3.5 A detailed example

To further illustrate the interplay of all the rules, let us specify and verify the following C function:

```
int f (int *r, int *l) {
  *r = (*r + *r) + (*l = *l + 1);
  return *r;
}
```

It corresponds to the λMC term $\mathsf{r} = ((*\mathsf{r} + *\mathsf{r}) + (\mathsf{l} = *\mathsf{l} + 1)); *\mathsf{r}$, which we denote here by e. We want to verify that e is correct w.r.t. to the following specification:

$$\mathsf{r} \hookrightarrow \mathsf{v} * \mathsf{l} \hookrightarrow \mathsf{w} \vdash \mathsf{cwp}\ \mathsf{e}\ \{\mathsf{u}.\ \mathsf{u} = (2\mathsf{v} + \mathsf{w} + 1) * \mathsf{r} \hookrightarrow (2\mathsf{v} + \mathsf{w} + 1) * \mathsf{l} \hookrightarrow (\mathsf{w} + 1)\}$$

We start proving this goal by applying CWP-SEQ, followed by CWP-WAND with

$$\Psi(\mathsf{u}) \triangleq (\mathsf{u} = (2\mathsf{v} + \mathsf{w} + 1)) * \mathsf{r} \hookrightarrow_L (2\mathsf{v} + \mathsf{w} + 1) * \mathsf{l} \hookrightarrow_L (\mathsf{w} + 1).$$

We then get two branches in the proof tree. For the second one (the implication), we use the rules for the $\mathbb{U}$ modality and CWP-LOAD. It remains to show the goal $\mathsf{cwp}\ \mathsf{r} = (*\mathsf{r} + *\mathsf{r}) + (\mathsf{l} = *\mathsf{l} + 1)\ \{\Psi\}$. For that we use the assignment rule CWP-STORE. The left hand side of assignment is trivially evaluated to r (using CWP-VALUE), so we focus on the right hand side instead. For the right hand side we apply the rule CWP-BIN-OP twice, distributing the resources from the precondition in such a way that

- we use $\mathtt{r} \overset{0.5}{\hookrightarrow} \mathtt{v}$ (twice) to verify $*\mathtt{r}$;
- we use $1 \hookrightarrow \mathtt{w}$ to verify $(1 = *1 + 1)$.

Those propositions are then combined and updated to $\mathtt{r} \hookrightarrow \mathtt{v} * 1 \hookrightarrow_L \mathtt{w} + 1$. The return value of the whole expression is $2\mathtt{v} + \mathtt{w} + 1$, as expected. Finally, the assignment rule updates the hypothesis to $\mathtt{r} \hookrightarrow_L (2\mathtt{v} + \mathtt{w} + 1) * 1 \hookrightarrow_L \mathtt{w} + 1$, which allows us to conclude the proof.

Of course, writing out a proof like this is a tedious endeavor, as we are forced at almost each intermediate step to come up with the division of the hypothesis among the new goals, as well as the auxiliary predicates Ψ_i. We will see in Section 3.6 that this process can be largely automated, handling both non-determinism and sequence points in most of the ordinary cases.

3.4 Soundness of weakest preconditions for λMC

In this section we prove adequacy of the separation logic with weakest preconditions for λMC as presented in Section 3.3. We do this by giving a model using the Iris framework that is structured in a similar way as the translation that we gave in Section 3.2. This translation consisted of three layers: the target HeapLang language, the monadic combinators, and the λMC operations themselves. In the model, each corresponding layer abstracts from the details of the previous layer, in such a way that we never have to break the abstraction of a layer. At the end, putting all of this together, we get the following adequacy statement:

Theorem 3.1 (Adequacy of Weakest Preconditions)**.** If cwp e {Φ} is derivable, then e has no undefined behavior for any evaluation order. In other words, run(e) does not assert false.

The proof of the adequacy theorem closely follows the layered structure, by combining the correctness of the monadic run combinator with adequacy of HeapLang in Iris [Jun+18b, Theorem 6]. The rest of this section is organized as:

1. Because our translation targets HeapLang, we start by recalling the separation logic with weakest preconditions, for HeapLang part of Iris (Section 3.4.1).
2. On top of the logic for HeapLang, we define a notion of weakest preconditions mwp e {Φ} for expressions e built from our monadic combinators (Section 3.4.2).
3. Next, we define the lockable points-to connective $\ell \overset{q}{\hookrightarrow}_\xi v$ using Iris's machinery for custom ghost state (Section 3.4.3).
4. Finally, we define weakest preconditions for λMC by combining the weakest preconditions for monadic expressions with our translation scheme (Section 3.4.5).

3.4.1 Weakest preconditions for HeapLang

We recall (Section 2.2.1) the most essential Iris connectives for reasoning about HeapLang programs: wp e {Φ} and $\ell \mapsto v$, which are the HeapLang weakest precondition proposition and the HeapLang points-to connective, respectively. An example rule

$$\textsc{wp-load}\ \frac{\ell \mapsto v \qquad (\ell \mapsto v \mathrel{-\!\!*} \Phi\ v)}{\mathsf{wp}\ !\ell\ \{\Phi\}} \qquad \textsc{wp-store}\ \frac{\ell \mapsto v \qquad (\ell \mapsto w \mathrel{-\!\!*} \Phi\ ())}{\mathsf{wp}\ \ell \leftarrow w\ \{\Phi\}}$$

$$\textsc{wp-bind}\ \frac{\mathsf{wp}\ e\ \{v.\ \mathsf{wp}\ K[\,v\,]\ \{\Phi\}\}}{\mathsf{wp}\ K[\,e\,]\ \{\Phi\}} \qquad \textsc{wp-assert}\ \frac{\mathsf{wp}\ e\ \{v.\ v = \mathbf{true} * \Phi\ ()\}}{\mathsf{wp}\ \mathbf{assert}(e)\ \{\Phi\}}$$

$$R * (\forall \gamma\ lk.\ \mathsf{is\_lock}(\gamma, lk, R) \mathrel{-\!\!*} \Phi\ lk) \vdash \mathsf{wp}\ \mathsf{newlock}\ ()\ \{\Phi\}$$
$$\mathsf{is\_lock}(\gamma, lk, R) * (R * \mathsf{locked}(\gamma) \mathrel{-\!\!*} \Phi\ ()) \vdash \mathsf{wp}\ \mathsf{acquire}\ lk\ \{\Phi\}$$
$$\mathsf{is\_lock}(\gamma, lk, R) * R * \mathsf{locked}(\gamma) * \Phi\ () \vdash \mathsf{wp}\ \mathsf{release}\ lk\ \{\Phi\}$$
$$\mathsf{is\_lock}(\gamma, lk, R) * \mathsf{is\_lock}(\gamma, lk, R) \dashv\vdash \mathsf{is\_lock}(\gamma, lk, R) \qquad (\textsc{islock-dupl})$$

Figure 3.4: Selected wp rules.

is the store rule for HeapLang, shown in Figure 3.4. The rule requires a points-to connective $\ell \mapsto v$, and the user receives the updated points-to connective $\ell \mapsto w$ back for proving $\Phi\ ()$. Note that the rule is formulated for a concrete location ℓ and a value w, instead of arbitrary expressions. This does not limit the expressive power; since the evaluation order in HeapLang is deterministic,[2] arbitrary expressions can be handled using the WP-BIND rule. Using this rule, one can bind an expression e in an arbitrary evaluation context K. We can thus use the WP-BIND rule twice to derive a more general store rule for HeapLang:

$$\frac{\mathsf{wp}\ e_2\ \{w.\ \mathsf{wp}\ e_1\ \{\ell.\ (\exists v.\ \ell \mapsto v) * (\ell \mapsto w \mathrel{-\!\!*} \Phi\ ())\}\}}{\mathsf{wp}\ (e_1 \leftarrow e_2)\ \{\Phi\}}$$

To verify the monadic combinators and the translation of λMC operations in the upcoming sections Sections 3.4.2 and 3.4.5, we need the specifications for all the functions that we use, including those on mutable sets and mutexes. The rules for mutable sets are the same as in Section 2.2.5, and thus omitted. They involve the usual abstract predicate $\mathsf{is\_mset}(s, X)$ stating that the reference s represents a set with contents X.

The rules for mutexes (the same as in Section 2.4.2) are presented in Figure 3.4. When a new mutex is created, a user gets access to a proposition $\mathsf{is\_lock}(\gamma, lk, R)$, which states that the value lk is a mutex containing the resources R. This proposition can be duplicated freely (ISLOCK-DUPL). A thread can acquire the mutex and receive the resources contained in it. In addition, the thread receives a token $\mathsf{locked}(\gamma)$ meaning that it has entered the critical section. When a thread leaves the critical section and releases the mutex, it has to give up both the token and the resources R.

[2] And right-to-left, although our monadic translation does not rely on that.

$$\textsc{mwp-ret}\ \frac{\mathsf{wp}\ e\ \{\Phi\}}{\mathsf{mwp}\ (\mathtt{ret}\ e)\ \{\Phi\}} \qquad \textsc{mwp-bind}\ \frac{\mathsf{mwp}\ e_1\ \{v.\ \mathsf{mwp}\ e_2[v/x]\ \{\Phi\}\}}{\mathsf{mwp}\ (x \xleftarrow{\mathsf{m}} e_1;\ e_2)\ \{\Phi\}} \qquad \textsc{mwp-pure}\ \frac{e \to_{\mathsf{pure}} e' \qquad \mathsf{mwp}\ K[e']\ \{\Phi\}}{\mathsf{mwp}\ K[e]\ \{\Phi\}}$$

$$\textsc{mwp-wand}\ \frac{\mathsf{mwp}\ e\ \{\Phi\} \qquad (\forall v.\ \Phi(v) \mathrel{-\!\!*} \Psi(v))}{\mathsf{mwp}\ e\ \{\Psi\}}$$

$$\textsc{mwp-par}\ \frac{\mathsf{mwp}\ e_1\ \{\Psi_1\} \qquad \mathsf{mwp}\ e_2\ \{\Psi_2\} \qquad (\forall w_1 w_2.\ \Psi_1\ w_1 * \Psi_2\ w_2 \mathrel{-\!\!*} \Phi\ (w_1, w_2))}{\mathsf{mwp}\ (e_1 \parallel^{\mathsf{m}} e_2)\ \{\Phi\}}$$

$$\textsc{mwp-atomic-env}\ \frac{\forall env.\ \mathsf{env\_inv}(env) \mathrel{-\!\!*} \mathsf{wp}\ (v\ env)\ \{w.\ \mathsf{env\_inv}(env) * \Phi\ w\}}{\mathsf{mwp}\ (\mathsf{atomic\_env}\ v)\ \{\Phi\}} \qquad \textsc{mwp-run}\ \frac{\mathsf{mwp}\ v\ \{\Phi\}}{\mathsf{wp\ run}\ v\ \{\Phi\}}$$

Figure 3.5: Selected monadic mwp rules.

The rule WP-ASSERT is new (it was not described in Chapter 2), but it can be easily derived given the following definition of **assert**:

$$\mathbf{assert}(e) \triangleq \mathtt{if}\ e\ \mathtt{then}\ ()\ \mathtt{else}\ \pi_1(0)$$

The idea behind this definition is that $\pi_1(0)$ is a stuck expression that can not be evaluated. Thus, in order for **assert**(e) not to get stuck, e has to evaluate to **true**.

3.4.2 Weakest preconditions for monadic expressions

As a next step, we define a weakest precondition proposition mwp $e\ \{\Phi\}$ for a monadic expression e. The definition is constructed in the ambient logic, and it encapsulates the monadic operations in a separate layer. Due to that, we are able to carry out proofs of high-level specifications without breaking the abstraction (Section 3.4.5). The specifications for selected monadic operations in terms of mwp are presented in Figure 3.5. We define the weakest precondition for a monadic expression e as follows:

$$\mathsf{mwp}\ e\ \{\Phi\} \triangleq \mathsf{wp}\ e \left\{ \begin{array}{c} g.\ \forall \gamma\ env\ lk.\ \mathsf{is\_lock}(\gamma, lk, \mathsf{env\_inv}(env)) \mathrel{-\!\!*} \\ \mathsf{wp}\ (g\ env\ lk)\ \{\Phi\} \end{array} \right\}$$

The idea is that we first reduce e to a monadic value g. To perform this reduction we have the outermost wp connective in the definition of mwp. This monadic value is then evaluated with an arbitrary environment and an arbitrary mutex. Note that we universally quantify over any mutex lk to support nested locking in atomic . This definition is parameterized by an *environment invariant* env_inv(env), which describes the resources accessible in the critical sections. We show how to define env_inv in the next subsection.

$$\textsc{heap-alloc}\quad \frac{\ell \mapsto v \qquad \mathsf{full\_heap}(\sigma)}{\Rrightarrow \ell \hookrightarrow_U v * \mathsf{full\_heap}(\sigma[\ell \leftarrow (U, v)])}$$

$$\textsc{heap-upd}\quad \frac{\ell \hookrightarrow_U v \qquad \mathsf{full\_heap}(\sigma)}{\Rrightarrow \sigma(\ell) = (U, v) * \ell \mapsto v * (\forall v'\, \xi'.\, \ell \mapsto v' \Rrightarrow\!\!\!\!\!\!\ast\ \ell \hookrightarrow_{\xi'} v' * \mathsf{full\_heap}(\sigma[\ell \leftarrow (\xi', v')]))}$$

Figure 3.6: Selected rules of the lockable heap construction.

Using this definition we derive the rules for $\mathsf{mwp}\ e\ \{\Phi\}$, as given in Figure 3.5. In a monad, the expression evaluation order is made explicit via the bind operation $x \xleftarrow{\mathsf{m}} e_1;\ e_2$. To that extent, contrary to HeapLang, we no longer have a rule like WP-BIND, which allows to bind an expression in a general evaluation context. Instead, we have the rule MWP-BIND, which reflects that the only evaluation context we have is the monadic bind $x \xleftarrow{\mathsf{m}} [\bullet];\ e$.

3.4.3 Modeling the heap

The monadic rules in Figure 3.5 are expressive enough to derive some of the λMC-level rules, but we are still missing one crucial part: handling of the heap. In order to do that, we need to define lockable points-to connectives $1 \overset{q}{\hookrightarrow}_\xi \mathsf{v}$ in such a way that they are linked to the HeapLang points-to connectives $\ell \mapsto v$.

The key idea is the following. The environment invariant env_inv of monadic weakest preconditions will track *all* HeapLang points-to connectives $\ell \mapsto v$ that have ever been allocated at the λMC level. Via Iris ghost state, we then connect this knowledge to the lockable points-to connectives $1 \overset{q}{\hookrightarrow}_\xi \mathsf{v}$. We refer to the construction that allows us to carry this out as the *lockable heap*. Note that the description of lockable heap is fairly technical and requires an understanding of the ghost state mechanism in Iris.

A lockable heap is a map $\sigma : \mathit{Loc} \xrightarrow{\mathsf{fin}} \{L, U\} \times \mathit{Val}$ that keeps track of the access levels and values associated with the locations. The connective $\mathsf{full\_heap}(\sigma)$ asserts the ownership of all the locations present in the domain of σ. Specifically, it asserts $\ell \mapsto v$ for each $[\ell \leftarrow (\xi, v)] \in \sigma$. The connective $\ell \overset{q}{\hookrightarrow}_\xi v$ then states that $[\ell \leftarrow (\xi, v)]$ is part of the global lockable heap, and it asserts this with the fractional permission q. We treat the lockable heap as an opaque abstraction, whose exact implementation via Iris ghost state is described in the Coq formalization [FGK19b]. The main interface for the locking heap are the rules in Figure 3.6. The rule HEAP-ALLOC states that we can turn a HeapLang points-to connective $\ell \mapsto v$ into $\ell \hookrightarrow_\xi v$ by changing the lockable heap σ accordingly. The rule HEAP-UPD states that given $\ell \hookrightarrow_\xi v$, we can temporarily get a HeapLang points-to connective $\ell \mapsto v$ out of the locking heap and update its value.

The environment invariant env_inv(*env*) in the definition of mwp ties the contents of the lockable heap to the contents of the environment *env*:

$$\mathsf{env\_inv}(env) \triangleq \exists \sigma\, X.\ \mathsf{is\_mset}(env, X) * \mathsf{full\_heap}(\sigma) * (\forall \ell \in X. \exists v.\ \sigma(\ell) = (L, v))$$

The first conjunct states that $X : \wp^{\mathsf{fin}}(Loc)$ is a set of locked locations, according to the environment *env*. The second conjunct asserts ownership of the global lockable heap σ. Finally, the last conjunct states that the contents of *env* agrees with the lockable heap: every location that is in X is locked according to σ.

3.4.4 The unlocking modality

The unlocking modality is defined in the logic as:

$$\mathbb{U}P \triangleq \exists S.\ (\ \mathop{\Large *}_{(\mathsf{l},\mathsf{v},q)\in S}\ \mathsf{l} \overset{q}{\hookrightarrow}_L \mathsf{v}) * ((\ \mathop{\Large *}_{(\mathsf{l},\mathsf{v},q)\in S}\ \mathsf{l} \overset{q}{\hookrightarrow}_U \mathsf{v}) \mathrel{-\!\!*} P)$$

Here S is a finite multiset of tuples containing locations, values, and fractions. The update modality accumulates the locked locations, waiting for them to be unlocked at a sequence point.

3.4.5 Deriving the λMC rules

To model weakest preconditions for λMC (Figure 3.3) we compose the construction we have just defined with the translation of Section 3.2 $\mathsf{cwp}\ \mathsf{e}\ \{\Phi\} \triangleq \mathsf{mwp}\ [\![\mathsf{e}]\!]\ \{\Phi'\}$. Here, Φ' is the obvious lifting of Φ from λMC values to HeapLang values. Using the rules from Figures 3.5 and 3.6 we derive the high-level λMC rules without unfolding the definition of the monadic mwp.

Example 3.2. Consider the rule CWP-STORE for assignments $\mathsf{e}_1 = \mathsf{e}_2$, stated in terms of $\mathsf{mwp}\ e\ \{\Phi\}$

$$\frac{\mathsf{mwp}\ [\![\mathsf{e}_1]\!]\ \{\Psi_1\} \qquad \mathsf{mwp}\ [\![\mathsf{e}_2]\!]\ \{\Psi_2\} \qquad (\forall \mathsf{l}\, \mathsf{w}.\ \Psi_1\ \mathsf{l} * \Psi_2\ \mathsf{w} \mathrel{-\!\!*} \exists \mathsf{v}.\ \mathsf{l} \hookrightarrow_U \mathsf{v} * (\mathsf{l} \hookrightarrow_L \mathsf{w} \mathrel{-\!\!*} \Phi\ \mathsf{w}))}{\mathsf{mwp}\ [\![\mathsf{e}_1 = \mathsf{e}_2]\!]\ \{\Phi\}}$$

Using MWP-BIND and MWP-PAR, the soundness of CWP-STORE can be reduced to verifying the assignment with e_1 being l, e_2 being v', under the assumption $\mathsf{l} \hookrightarrow_U \mathsf{v}$. We use MWP-ATOMIC-ENV to turn our goal into a HeapLang weakest precondition proposition and to gain access an environment *env*, and to the proposition env_inv(*env*), from which we extract the lockable heap σ. We then use HEAP-UPD to get access to the underlying HeapLang location and obtain that l is not locked according to σ. Due to the environment invariant, we obtain that l is not in *env*, which allows us to prove the **assert** for sequence point violation in the interpretation of the assignment. Finally, we perform the physical update of the location.

3.5 A symbolic executor for λMC

In order to turn our program logic into an automated procedure, it is important to have rules for weakest preconditions that have an algorithmic form. However, the rules for binary operators in our separation logic for λMC do not have such a form. Take for example the rule CWP-BIN-OP for binary operators $\mathsf{e}_1 \circledcirc \mathsf{e}_2$. This rule cannot be applied in an algorithmic manner. To use the rule one should supply the postconditions for e_1 and e_2, and frame the resources from the context into two disjoint parts. This is generally impossible to do automatically.

To address this problem, we first describe how the rules for binary operators can be transformed into algorithmic rules by exploiting the notion of *symbolic execution* [BCO05] (Section 3.5.1). We then show how to implement these algorithmic rules as part of an automated symbolic execution procedure (Section 3.5.2).

3.5.1 Rules for symbolic execution

We say that we can *symbolically execute* an expression e using a *precondition* P, if we can find a *symbolic execution tuple* (w, Q, R) consisting of a *return value* w, a *postcondition* Q, and a *frame* R satisfying:

$$P \vdash \mathsf{cwp}\ \mathsf{e}\ \{\mathsf{v}.\ \mathsf{v} = \mathsf{w} * Q\} * R$$

This specification is much like that of ordinary symbolic execution in separation logic [BCO05], but there is important difference. Apart from computing the postcondition Q and the return value w, there is also the frame R, which describes the resources that are *not used* for proving e. For instance, if the precondition P is $P' * \mathsf{l} \overset{q}{\hookrightarrow} \mathsf{w}$ and e is a load operation $*\mathsf{l}$, then we can symbolically execute e with the postcondition Q being $\mathsf{l} \overset{q/2}{\hookrightarrow} \mathsf{w}$, and the frame R being $P' * \mathsf{l} \overset{q/2}{\hookrightarrow} \mathsf{w}$. Clearly, P' is not needed for proving the load, so it can be moved into the frame. More interestingly, since loading the contents of l requires a read permission $\mathsf{l} \overset{p}{\hookrightarrow} \mathsf{w}$, with $p \in (0, 1]$, we can split the hypothesis $\mathsf{l} \overset{q}{\hookrightarrow} \mathsf{w}$ into two halves and move one into the frame. Below we will see why that matters.

If we can symbolically execute one of the operands of a binary expression $\mathsf{e}_1 \circledcirc \mathsf{e}_2$, say e_1 in P, and find a symbolic execution tuple (w_1, Q, R), then we can use the following admissible rule:

$$\frac{R \vdash \mathsf{cwp}\ \mathsf{e}_2\ \{\mathsf{w}_2.\ Q \mathrel{-\!\!*} \Phi\ (\mathsf{w}_1\ [\![\circledcirc]\!]\ \mathsf{w}_2)\}}{P \vdash \mathsf{cwp}\ (\mathsf{e}_1 \circledcirc \mathsf{e}_2)\ \{\Phi\}}$$

This rule has a much more algorithmic flavor than the rule CWP-BIN-OP. Applying the above rule now boils down to finding such a tuple (w, Q, R), instead of having to infer postconditions for both operands, as we need to do to apply CWP-BIN-OP.

For instance, given an expression $(*\mathsf{l}) \circledcirc \mathsf{e}_2$ and a precondition $P' * \mathsf{l} \overset{q}{\hookrightarrow} \mathsf{v}$, we can

derive the following rule:

$$\frac{P' * 1 \overset{q/2}{\hookrightarrow} \mathsf{v} \vdash \mathsf{cwp}\ \mathsf{e}_2 \left\{ \mathsf{w}_2.\ 1 \overset{q/2}{\hookrightarrow} \mathsf{v} \mathrel{-\!\!*} \Phi\ (\mathsf{v}\ [\![\circledcirc]\!]\ \mathsf{w}_2) \right\}}{P' * 1 \overset{q}{\hookrightarrow} \mathsf{v} \vdash \mathsf{cwp}\ (*1 \circledcirc \mathsf{e}_2)\ \{\Phi\}}$$

This rule matches the intuition that only a fraction of the permission $1 \overset{q}{\hookrightarrow} \mathsf{v}$ is needed to prove a load $*1$, so that the remaining half of the permission can be used to prove the correctness of e_2 (which may contain other loads of 1).

3.5.2 An algorithm for symbolic execution

For an arbitrary expression e and a proposition P, it is unlikely that one can find such a symbolic execution tuple (w, Q, R) automatically. However, for a certain class of C expressions that appear in actual programs we can compute a choice of such a tuple. To illustrate our approach, we will define such an algorithm for a small subset $\overline{\mathsf{expr}}$ of C expressions described by the following grammar:

$$\bar{\mathsf{e}} \in \overline{\mathsf{expr}} ::= \mathsf{v} \mid *\bar{\mathsf{e}} \mid \bar{\mathsf{e}}_1 = \bar{\mathsf{e}}_2 \mid \bar{\mathsf{e}}_1 \circledcirc \bar{\mathsf{e}}_2.$$

We keep this subset small to ease presentation. In Section 3.7 we explain how to extend the algorithm to cover the sequenced bind operator $\mathsf{x} \leftarrow \bar{\mathsf{e}}_1; \bar{\mathsf{e}}_2$.

Moreover, to implement symbolic execution, we cannot manipulate arbitrary separation logic propositions. We thus restrict to *symbolic heaps* ($m \in \mathsf{sheap}$), which are defined as finite partial functions $\mathsf{Loc} \overset{\mathsf{fin}}{\rightharpoonup} (\{L, U\} \times (0, 1] \times \mathsf{val})$ representing a collection of points-to propositions:

$$[\![m]\!] \triangleq \mathop{\Large *}_{\substack{l \in \mathsf{dom}(m) \\ m(l) = (\xi, \mathsf{q}, \mathsf{v})}} l \overset{q}{\hookrightarrow}_{\xi} \mathsf{v}.$$

We use the following operations on symbolic heaps:

- $m[l \mapsto (\xi, q, \mathsf{v})]$ sets the entry $m(l)$ to (ξ, q, v);
- $m \setminus \{l \mapsto \_\}$ removes the entry $m(l)$ from m;
- $m_1 \sqcup m_2$ merges the symbolic heaps m_1 and m_2 in such a way that for each $l \in \mathsf{dom}(m_1) \cup \mathsf{dom}(m_2)$, we have:

$$(m_1 \sqcup m_2)(l) = \begin{cases} m_i(l) & \text{if } l \in \mathsf{dom}(m_i) \text{ and } l \notin \mathsf{dom}(m_j) \\ (\xi \vee \xi', q + q', \mathsf{v}) & \text{if } m_1(l) = (\xi, q, \mathsf{v}) \text{ and } m_2(l) = (\xi', q', \_). \end{cases}$$

With this representation of propositions, we define the symbolic execution algorithm as a partial function $\mathsf{forward} : (\mathsf{sheap} \times \mathsf{expr}) \to (\mathsf{val} \times \mathsf{sheap} \times \mathsf{sheap})$, which satisfies the specification stated in Section 3.5.1, *i.e.*, for which the following holds:

Theorem 3.3. Given an expression e and an symbolic heap m, if $\mathsf{forward}(m, \mathsf{e})$ returns a tuple (w, m_1^o, m_1), then $[\![m]\!] \vdash \mathsf{cwp}\ \mathsf{e}\ \{\mathsf{v}.\ \mathsf{v} = \mathsf{w} * [\![m_1^o]\!]\} * [\![m_1]\!]$.

$$\mathsf{forward}(m, \mathsf{v}) \triangleq (\mathsf{v}, \varnothing, m)$$

$$\mathsf{forward}(m, \mathsf{e}_1 \circledcirc \mathsf{e}_2) \triangleq (\mathsf{v}_1 \mathbin{[\![\circledcirc]\!]} \mathsf{v}_2, m_1^o \sqcup m_2^o, m_2)$$
$$\textbf{where} \quad (\mathsf{v}_1, m_1^o, m_1) = \mathsf{forward}(m, \mathsf{e}_1)$$
$$(\mathsf{v}_2, m_2^o, m_2) = \mathsf{forward}(m_1, \mathsf{e}_2)$$

$$\mathsf{forward}(m, *\mathsf{e}_1) \triangleq (\mathsf{w}, m_2^o \sqcup \{1 \mapsto (U, q, \mathsf{w})\}, m_2)$$
$$\textbf{where} \quad (1, m_1^o, m_1) = \mathsf{forward}(m, \mathsf{e}_1) \qquad \text{provided } 1 \in \mathsf{Loc}$$
$$(m_2, m_2^o, q, \mathsf{w}) = \mathsf{delete\_frac\_2}(1, m_1, m_1^o)$$

$$\mathsf{forward}(m, \mathsf{e}_1 = \mathsf{e}_2) \triangleq (\mathsf{v}_2, m_3^o \sqcup \{1 \mapsto (L, 1, \mathsf{v}_2)\}, m_3)$$
$$\textbf{where} \quad (1, m_1^o, m_1) = \mathsf{forward}(m, \mathsf{e}_1) \qquad \text{provided } 1 \in \mathsf{Loc}$$
$$(\mathsf{v}_2, m_2^o, m_2) = \mathsf{forward}(m_1, \mathsf{e}_2)$$
$$(m_3, m_3^o) = \mathsf{delete\_full\_2}(1, m_2, m_1^o \sqcup m_2^o)$$

$$\mathsf{forward}(m, \mathsf{e}) \triangleq \bot \qquad \text{if } \mathsf{e} \notin \overline{\mathsf{expr}}$$

Auxiliary functions:

$$\mathsf{delete\_frac\_2}(1, m_1, m_2) \triangleq \begin{cases} (m_1\,[1 \mapsto (U, q/2, \mathsf{v})], m_2, q/2, \mathsf{v}) & \text{if } m_1(1) = (U, q, \mathsf{v}) \\ (m_1, m_2\,[1 \mapsto (U, q/2, \mathsf{v})], q/2, \mathsf{v}) & \text{if } m_1(1) \neq (U, \_, \_),\ m_2(1) = (U, q, \mathsf{v}) \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{delete\_full\_2}(1, m_1, m_2) \triangleq (m_1 \setminus \{1 \mapsto \_\}, m_2 \setminus \{1 \mapsto \_\})$$
$$\textbf{where} \quad (U, 1, \_) = (m_1 \sqcup m_2)(1)$$

Figure 3.7: The definition of the symbolic executor.

The definition of the algorithm is shown in Figure 3.7. Given a tuple (m, e), a call to $\mathsf{forward}(m, \mathsf{e})$ either returns a tuple (v, m^o, m') or fails, which either happens when $\mathsf{e} \notin \overline{\mathsf{expr}}$ or when one of intermediate steps of computation fails. In the latter cases, we write $\mathsf{forward}(m, \mathsf{e}) = \bot$. The algorithm proceeds by case analysis on the expression e. In each case, the expected output is described by the equation $\mathsf{forward}(m, \mathsf{e}) = (\mathsf{v}, m^o, m')$. The results of the intermediate computations appear on separate lines under the clause "**where** …". If one of the corresponding equations does not hold, *e.g.*, a recursive call fails, then the failure is propagated. Let us now explain the case for the assignment operator.

If e is an assignment operator $\mathsf{e}_1 = \mathsf{e}_2$, we first evaluate e_1 and then e_2. Fixing the order of symbolic execution from left to right does not compromise the non-determinism underlying the C semantics of binary operators. Indeed, when $\mathsf{forward}(m, \mathsf{e}_1) = (\mathsf{v}_1, m_1^o, m_1)$, we evaluate the expression e_2, using the frame m_1, *i.e.*, only the resources of m that remain after the execution of e_1. When

$\mathsf{forward}(m, \mathsf{e}_1) = (1, m_1^o, m_1)$, with $1 \in \mathsf{Loc}$, and $\mathsf{forward}(m_1, \mathsf{e}_2) = (\mathsf{v}_2, m_2^o, m_2)$, the function delete_full_2$(1, m_2, m_1^o \sqcup m_2^o)$ checks whether $(m_2 \sqcup m_1^o \sqcup m_2^o)(1)$ contains the write permission $1 \hookrightarrow_U -$. If this holds, it removes the location 1, so that the write permission is now consumed. Finally, we merge $\{1 \mapsto (L, 1, \mathsf{v}_2)\}$ with the output heap m_3^o, so that after assignment, the write permission $1 \hookrightarrow_L \mathsf{v}_2$ is given back in a locked state.

3.6 A verification condition generator for λMC

To establish correctness of programs, we need to prove goals $P \vdash \mathsf{cwp}\ \mathsf{e}\ \{\Phi\}$. To prove such a goal, one has to repeatedly apply the rules for weakest preconditions, intertwined with logical reasoning. In this section we will automate this process for λMC by means of a *verification condition generator* (vcgen).

As a first attempt to define a vcgen, one could try to recurse over the expression e and apply the rules in Figure 3.3 eagerly. This would turn the goal into a separation logic proposition that subsequently should be solved. However, as we pointed out in Section 3.5.1, the resulting separation logic proposition will be very difficult to prove—either interactively or automatically—due to the existentially quantified postconditions that appear because of uses of the rules for binary operators (*e.g.*, CWP-BIN-OP). We then proposed alternative rules that avoid the need for existential quantifiers. These rules look like:

$$\frac{R \vdash \mathsf{cwp}\ \mathsf{e}_2\ \{\mathsf{v}_2.\ Q \mathrel{-\!\!*} \Phi\ (\mathsf{v}_1\ [\![\circledcirc]\!]\ \mathsf{v}_2)\}}{P \vdash \mathsf{cwp}\ (\mathsf{e}_1 \circledcirc \mathsf{e}_2)\ \{\Phi\}}$$

To use this rule, the crux is to symbolically execute e_1 with precondition P into a symbolic execution triple (v_1, Q, R), which we alluded could be automatically computed by means of the symbolic executor if $\mathsf{e}_1 \in \overline{\mathsf{expr}}$ (Section 3.5.2).

We can only use the symbolic executor if P is of the shape $[\![m]\!]$ for a symbolic heap m. However, in actual program verification, the precondition P is hardly ever of that shape. In addition to a series of points-to connectives (as described by a symbolic heap), we may have arbitrary propositions of separation logic, such as pure facts, abstract predicates, nested Hoare triples, Iris ghost state, *etc.* These propositions may be needed to prove intermediate verification conditions, *e.g.*, for function calls. As such, to effectively apply the above rule, we need to separate our precondition P into two parts: a symbolic heap $[\![m]\!]$ and a remainder P'. Assuming $\mathsf{forward}(m, \mathsf{e}_1) = (\mathsf{v}_1, m_1^o, m_1)$, we may then use the following rule:

$$\frac{P' * [\![m_1]\!] \vdash \mathsf{cwp}\ \mathsf{e}_2\ \{\mathsf{v}_2.\ [\![m_1^o]\!] \mathrel{-\!\!*} \Phi\ (\mathsf{v}_1\ [\![\circledcirc]\!]\ \mathsf{v}_2)\}}{P' * [\![m]\!] \vdash \mathsf{cwp}\ (\mathsf{e}_1 \circledcirc \mathsf{e}_2)\ \{\Phi\}}$$

It is important to notice that by applying this rule, the remainder P' remains in our precondition as is, but the symbolic heap is changed from $[\![m]\!]$ into $[\![m_1]\!]$, *i.e.*, into the frame that we obtained by symbolically executing e_1.

Having applied the above rule, we could proceed recursively on the structure of the expression. For example, if e_2 is an assignment operator $\mathsf{e}_1' = \mathsf{e}_2'$, we could use the

following rule:

$$\frac{P' * [\![m_2]\!] \vdash \mathsf{cwp}\ \mathsf{e}_2' \{\mathsf{v}_2'.\ [\![m_2^o]\!] \mathrel{-\!\!*} \exists \mathsf{w}.\ 1 \hookrightarrow_U \mathsf{w} * (1 \hookrightarrow_L \mathsf{v}_2' \mathrel{-\!\!*} \Phi'\ \mathsf{v}_2')\}}{P' * [\![m_1]\!] \vdash \mathsf{cwp}\ (\mathsf{e}_1' = \mathsf{e}_2') \{\Phi'\}}$$

Under the proviso $\mathsf{forward}(m_1, \mathsf{e}_1') = (1, m_2^o, m_2)$ with $1 \in \mathsf{Loc}$. Again, we see that the remainder P' is kept, while the symbolic heap is updated—the precondition is thus again in the right shape to apply a similar rule for e_2'.

It should come as no surprise that we can automate this process, by applying rules, such as the ones we have given above, recursively, and threading through symbolic heaps. Formally, we do this by defining the vcgen as a total function: $\mathsf{vcg} : (\mathsf{sheap} \times \mathsf{expr} \times (\mathsf{sheap} \to \mathsf{val} \to \mathit{iProp})) \to \mathit{iProp}$ where iProp is the type of propositions of our logic. The definition of vcg is given in Figure 3.8. Before explaining the details, let us state its correctness theorem:

Theorem 3.4. Given an expression e, a symbolic heap m, and a postcondition Φ, the following statement holds:

$$\frac{P' \vdash \mathsf{vcg}(m, \mathsf{e}, \lambda m'\ \mathsf{v}.\ [\![m']\!] \mathrel{-\!\!*} \Phi\ \mathsf{v})}{P' * [\![m]\!] \vdash \mathsf{cwp}\ \mathsf{e}\ \{\Phi\}}$$

This theorem reflects the general shape of the rules we previously described. We start off with a goal $P' * [\![m]\!] \vdash \mathsf{cwp}\ \mathsf{e}\ \{\Phi\}$, and after using the vcgen, we should prove that the generated goal follows from P'. It is important to note that the continuation in the vcgen is not only parameterized by the return value, but also by a symbolic heap corresponding to the resources that remain. To get these resources back, the vcgen is initiated with the continuation $\lambda m'\ \mathsf{v}.\ [\![m']\!] \mathrel{-\!\!*} \Phi\ \mathsf{v}$.

Most clauses of the definition of the vcgen (Figure 3.8) follow the approach we described so far. For unary expressions like load we generate a condition that corresponds to the weakest precondition rule. For binary expressions, we symbolically execute either operand, and proceed recursively in the other. There are a number of important bells and whistles that we will discuss now.

3.6.1 Sequencing

In the case of sequenced binds $\mathsf{x} \leftarrow \mathsf{e}_1; \mathsf{e}_2$, we recursively compute the verification condition for e_1 with the continuation:

$$\lambda m'\ \mathsf{v}.\ \mathbb{U}\ (\mathsf{vcg}(\mathsf{unlock}(m'), \mathsf{e}_2[\mathsf{v}/\mathsf{x}], \mathcal{K})).$$

Due to a sequence point, all locations modified by e_1 will be in the unlocked state after it is finished executing. Therefore, in the recursive call to e_2 we unlock all locations in the symbolic heap (*c.f.* $\mathsf{unlock}(m')$), and we include a $\mathbb{U}$ modality in the continuation. The $\mathbb{U}$ modality is crucial so that the resources that are not given to the vcgen (the remainder P' in Theorem 3.4) can also be unlocked.

$$\mathsf{vcg}(m, \mathsf{v}, \mathcal{K}) \triangleq \mathcal{K}\ m\ \mathsf{v}$$

$$\mathsf{vcg}(m, \mathsf{e}_1 \circledcirc \mathsf{e}_2, \mathcal{K}) \triangleq$$
$$\begin{cases} \mathsf{vcg}(m_2, \mathsf{e}_2, \lambda m'\, \mathsf{v}_2.\ \mathcal{K}\ (m' \sqcup m^o)\ (\mathsf{v}_1 \circledcirc \mathsf{v}_2)) & \text{if } \mathsf{forward}(m, \mathsf{e}_1) = (\mathsf{v}_1, m^o, m_2) \\ \mathsf{vcg}(m_1, \mathsf{e}_1, \lambda m'\, \mathsf{v}_1.\ \mathcal{K}\ (m' \sqcup m^o)\ (\mathsf{v}_1 \circledcirc \mathsf{v}_2)) & \text{if } \mathsf{forward}(m, \mathsf{e}_1) = \bot \text{ and } \mathsf{forward}(m, \mathsf{e}_2) = (\mathsf{v}_2, m^o, m_1) \\ [\![m]\!] \mathrel{-\!\!*} \mathsf{cwp}\ (\mathsf{e}_1 \circledcirc \mathsf{e}_2)\ \{\mathcal{K}_{\mathrm{ret}}\} & \text{otherwise} \end{cases}$$

$$\mathsf{vcg}(m, *\mathsf{e}, \mathcal{K}) \triangleq \mathsf{vcg}(m, \mathsf{e}, \mathcal{K}')$$
$$\textbf{with } \mathcal{K}' \triangleq \lambda m\, \mathsf{l}.\ \begin{cases} \mathcal{K}\ m\ \mathsf{w} & \text{if } \mathsf{l} \in \mathsf{Loc} \text{ and } m(\mathsf{l}) = (U, q, \mathsf{w}) \\ [\![m]\!] \mathrel{-\!\!*} \exists \mathsf{w}\ q.\ \mathsf{l} \overset{q}{\hookrightarrow}_U \mathsf{w} * (\mathsf{l} \overset{q}{\hookrightarrow}_U \mathsf{w} \mathrel{-\!\!*} \mathcal{K}_{\mathrm{ret}}\ \mathsf{w}) & \text{otherwise} \end{cases}$$

$$\mathsf{vcg}(m, \mathsf{e}_1 = \mathsf{e}_2, \mathcal{K}) \triangleq$$
$$\begin{cases} \mathsf{vcg}(m_2, \mathsf{e}_2, \lambda m'\, \mathsf{v}.\ \mathcal{K}'\ (m' \sqcup m^o)(1, \mathsf{v})) & \text{if } \mathsf{forward}(m, \mathsf{e}_1) = (1, m^o, m_2) \\ \mathsf{vcg}(m_1, \mathsf{e}_1, \lambda m'\, 1.\ \mathcal{K}'\ (m' \sqcup m^o)(1, \mathsf{v})) & \text{if } \mathsf{forward}(m, \mathsf{e}_1) = \bot \text{ and } \mathsf{forward}(m, \mathsf{e}_2) = (\mathsf{v}, m^o, m_1) \\ [\![m]\!] \mathrel{-\!\!*} \mathsf{cwp}\ (\mathsf{e}_1 = \mathsf{e}_2)\ \{\mathcal{K}_{\mathrm{ret}}\} & \text{otherwise} \end{cases}$$
$$\textbf{with } \mathcal{K}' \triangleq \lambda m\, (1, \mathsf{v}).\ \begin{cases} \mathcal{K}\ (m' \sqcup \{1 \mapsto (L, 1, \mathsf{v})\})\ \mathsf{v} & \text{if } 1 \in \mathsf{Loc} \text{ and } \mathsf{delete\_full}(1, m) = m' \\ [\![m]\!] \mathrel{-\!\!*} \exists \mathsf{w}.\ 1 \hookrightarrow_U \mathsf{w} * (1 \hookrightarrow_L \mathsf{v} \mathrel{-\!\!*} \mathcal{K}_{\mathrm{ret}}\ \mathsf{v}) & \text{otherwise} \end{cases}$$

$$\mathsf{vcg}(m, \mathsf{x} \leftarrow \mathsf{e}_1; \mathsf{e}_2, \mathcal{K}) \triangleq \mathsf{vcg}(m, \mathsf{e}_1, \lambda m'\, \mathsf{v}.\ \mathbb{U}\ (\mathsf{vcg}(\mathsf{unlock}(m'), \mathsf{e}_2[\mathsf{v}/\mathsf{x}], \mathcal{K})))$$

Auxiliary functions:

$$\mathcal{K}_{\mathrm{ret}} : \mathsf{val} \to \mathit{iProp} \triangleq \lambda \mathsf{w}.\ (\exists m'.\ [\![m']\!] * \mathcal{K}\ m'\ \mathsf{w}) \qquad \mathsf{unlock}(m) \triangleq \bigsqcup_{\substack{1 \in \mathrm{dom}(m) \\ m(1) = (\_, \mathsf{q}, \mathsf{v})}} \{1 \mapsto (U, q, \mathsf{v})\}$$

Figure 3.8: Selected cases of the verification condition generator.

3.6.2 Handling failure

In the case of binary operators $\mathsf{e}_1 \circledcirc \mathsf{e}_2$, it could be that the symbolic executor fails on both e_1 and e_2, because neither of the arguments were of the right shape (*i.e.*, not an element of $\overline{\mathsf{expr}}$), or the required resources were not present in the symbolic heap. In this case the vcgen generates the goal of the form $[\![m]\!] \mathrel{-\!\!*} \mathsf{cwp}\ (\mathsf{e}_1 \circledcirc \mathsf{e}_2)\ \{\mathcal{K}_{\mathrm{ret}}\}$ where $\mathcal{K}_{\mathrm{ret}} \triangleq \lambda \mathsf{w}.\ \exists m'.\ [\![m']\!] * \mathcal{K}\ m'\ \mathsf{w}$. What appears here is that the current symbolic heap $[\![m]\!]$ is given back to the user, which they can use to prove the weakest precondition

of $e_1 \circledcirc e_2$ by hand. Through the postcondition $\exists m'.\ [\![m']\!] * \mathcal{K}\ m'\ \mathtt{w}$ the user can resume the vcgen, by choosing a new symbolic heap m' and invoking the continuation $\mathcal{K}\ m'\ \mathtt{w}$.

For assignments $\mathrm{e}_1 = \mathrm{e}_2$ we have a similar situation. Symbolic execution of both e_1 and e_2 may fail, and then we generate a goal similar to the one for binary operators. If the location l that we wish to assign to is not in the symbolic heap, we use the continuation $[\![m]\!] \mathrel{-\!\!*} \exists \mathtt{w}.\ \mathrm{l} \hookrightarrow_U \mathtt{w} * (\mathrm{l} \hookrightarrow_L \mathtt{v} \mathrel{-\!\!*} \mathcal{K}_{\mathrm{ret}}\ \mathtt{v})$. As before, the user gets back the current symbolic heap $[\![m]\!]$, and could resume the vcgen through the postcondition $\mathcal{K}_{\mathrm{ret}}\ \mathtt{v}$ by picking a new symbolic heap.

3.7 Discussion

In this section we discuss some extensions of the λMC language, the logic, and the vcgen that we have implemented in the Coq formalization.

3.7.1 Extensions of the language

The memory model that we have presented in this paper was purposely oversimplified. In the Coq formalization, the memory model for λMC additionally supports mutable local variables, arrays, and pointer arithmetic. Adding support for these features was relatively easy and required only local changes to the definitional semantics and the separation logic.

For implementing mutable local variables, we tag each location with a Boolean that keeps track of whether it is an allocated or a local variable. Using this tag we disallow the `free(-)` operation to deallocated local variables.

Our extended memory model is block/offset-based like CompCert's memory model [LB08]. Pointers are not simply represented as locations, but as pairs (ℓ, i), where ℓ is a HeapLang reference to a memory block containing a list of values, and i is an offset into that block. The points-to connectives of our separation logic then correspondingly range over block/offset-based pointers.

3.7.2 Symbolic execution of sequence points

We have adapted our forward algorithm to handle sequenced bind operators $\mathtt{x} \leftarrow \mathrm{e}_1; \mathrm{e}_2$. The subtlety lies in supporting nested sequenced binds. For example, in an expression $(\mathtt{x} \leftarrow \mathrm{e}_1; \mathrm{e}_2) + \mathrm{e}_3$ the postcondition of e_1 can be used (along with the frame) for the symbolic execution of e_2, but it cannot be used for the symbolic execution of e_3. In order to solve this, our forward algorithm takes a *stack* of symbolic heaps as an input, and returns a *stack* of symbolic heaps (of the same length) as a frame. All the cases shown in Figure 3.7 are easily adapted w.r.t. this modification, and the following definition captures the case for the sequence point bind:

$$
\begin{array}{l}
\mathsf{forward}(\vec{m}, \mathtt{x} \leftarrow \mathrm{e}_1; \mathrm{e}_2) \triangleq (\mathtt{v}_2, m_2^o \sqcup m', \vec{m}_2) \\
\quad \textbf{where} \quad (\mathtt{v}_1, m_1^o, \vec{m}_1) \;=\; \mathsf{forward}(\vec{m}, \mathrm{e}_1) \\
\qquad\qquad\quad (\mathtt{v}_2, m_2^o, m' :: \vec{m}_2) \;=\; \mathsf{forward}(\mathsf{unlock}(m_1^o) :: \vec{m}_1, \mathrm{e}_2[\mathtt{v}_1/\mathtt{x}])
\end{array}
$$

To accommodate for this change, we need to update the auxiliary operations delete_frac_2 and delete_full_2 to operate on stacks of symbolic heaps. Finally, at the top level, the verification condition generator invokes forward with a stack containing a single symbolic heap.

3.7.3 Shared resource invariants

As in Krebbers's logic [Kre14], the rules for binary operators in Figure 3.3 require the resources to be separated into disjoint parts for the subexpressions. If both sides of a binary operator are function calls, then they can only share read permissions despite that both function calls are executed atomically. Following Krebbers, we address this limitation by adding a shared resource invariant R to our weakest preconditions and add the following rules:

$$\textsc{cwp-add-resource}\quad \frac{R_1 \qquad \mathsf{cwp}_{R_1 * R_2}\ \mathsf{e}\ \{\mathsf{v}.\ R_1 \mathrel{-\!\!*} \Phi\ \mathsf{v}\}}{\mathsf{cwp}_{R_2}\ \mathsf{e}\ \{\Phi\}}$$

$$\textsc{cwp-call}\quad \frac{\mathsf{f(x)\{e\}\ defined} \qquad R \mathrel{-\!\!*} \mathbb{U}(\mathsf{cwp}_{\mathsf{True}}\ \mathsf{e[x/v]}\ \{\mathsf{w}.\ R * \Phi\ \mathsf{w}\})}{\mathsf{cwp}_R\ \mathsf{f(v)}\ \{\Phi\}}$$

$$\textsc{wpm-add-resource}\quad \frac{R_1 \qquad \mathsf{mwp}_{R_1 * R_2}\ e\ \{v.\ R_1 \mathrel{-\!\!*} \Phi\ v\}}{\mathsf{cwp}_{R_2}\ e\ \{\Phi\}}$$

$$\textsc{wpm-atomic}\quad \frac{R \mathrel{-\!\!*} \exists Q.\ Q * \mathsf{mwp}_Q\ \big(v\ ()\big)\ \{w.\ Q \mathrel{-\!\!*} R * \Phi(w)\}}{\mathsf{mwp}_R\ \mathsf{atomic}\ v\ \{\Phi\}}$$

$$\textsc{wpm-atomic-env}\quad \frac{\forall env.\ \mathsf{env\_inv}(env) * R \mathrel{-\!\!*} \mathsf{wp}\ (v\ env)\ \{w.\ \mathsf{env\_inv}(env) * R * \Phi\ w\}}{\mathsf{mwp}_R\ \mathsf{atomic\_env}\ v\ \{\Phi\}}$$

To temporarily transfer resources into the invariant, one can use the rule CWP-ADD-RESOURCE. Because function calls are not interleaved, one can use the rule CWP-CALL to gain access to the shared resource invariant for the duration of the function call. The new cwp rules are derived from the new mwp rules WPM-ADD-RESOURCE, WPM-ATOMIC, and WPM-ATOMIC-ENV.

Our handling of shared resource invariants generalizes the treatment by Krebbers: using custom ghost state in Iris we can endow the resource invariant with a protocol. This allows us to verify examples that were previously impossible [Kre14]:

```
int f(int *p, int y) { return (*p = y); }
int main() { int x; f(&x, 3) + f(&x, 4); return x; }
```

Krebbers could only prove that `main` returns 0, 3 or 4, whereas we can prove it returns 3 or 4 by combining resource invariants with Iris's ghost state. We can do this by establishing a protocol on the location x, stating that the value of x cannot become 0 once it has been set to 3 or 4.

To support shared resource invariants in $\mathsf{cwp}_R\ \mathsf{e}\ \{\Phi\}$, we had to develop and prove a novel expressive specification for locks/mutexes. For the reasons of space we do not present the specification here, and refer the reader to the Coq formalization.

3.7.4 Implementation in Coq

In the Coq development [FGK19b] we have:

- Defined λMC with the extensions described above, as well as the monadic combinators, as a shallow embedding on top of Iris's HeapLang [Jun+18b; Iri20].
- Modeled the separation logic for λMC and the monadic combinators as a shallow embedding on top of the Iris's program logic for HeapLang.
- Implemented the symbolic executor and vcgen as computable Coq functions, and proved their soundness w.r.t. our separation logic.
- Turned the verification condition generator into a tactic that integrates into the Iris Proof Mode/MoSeL framework [KTB17; Kre+18].

This last point allowed us to leverage the existing machinery for separation logic proofs in Coq. Firstly, we get basic building blocks for implementing the vcgen tactic for free. Secondly, when the vcgen is unable to solve the goal, one can use the Iris Proof Mode/MoSeL tactics to help out in a convenient manner.

To implement the symbolic executor and vcgen, we had to reify the terms and values of λMC. To see why reification is needed, consider the data type for symbolic heaps, which uses locations as keys. In proofs, those locations appear as universally quantified variables. To compute using these, for example when performing operations on symbolic heaps, we need to reify them into some symbolic representation. We have implemented the reification mechanism using type classes, following Spitters and van der Weegen [SV11].

With all the mechanics in place, our vcgen is able to aid us significantly. Consider the following program that copies the contents of one array into another:

```
int arraycopy(int *p, int *q, int n) {
  int pend = p + n;
  while (p < pend) { *(p++) = *(q++); }
}
```

We proved $\{p \mapsto \vec{x} * q \mapsto \vec{y} * (|\vec{x}| = |\vec{y}| = n)\}$ `arraycopy(p,q,n)` $\{p \mapsto \vec{y} * q \mapsto \vec{y}\}$ in 11 lines of Coq code. The vcgen can automatically process the program up until the while loop. At that point, the user has to manually perform an induction on the array, providing a suitable induction hypothesis. The vcgen is then able to discharge the base case automatically. In the inductive case, it will automatically process the program until the next iteration of the while loop, where the user has to apply the induction hypothesis.

3.8 Related work

3.8.1 C semantics

There has been a considerable body of work on formal semantics for the C language, including several large projects that aimed to formalize substantial subsets of C [Nor98; Ler09; ER12; HER15; Mem+16; Kre15], and projects that focused on

specific aspects like its memory model [Mem+16; Kre13; Kre16; LB08; Coh+09b; Kan+15; Cao+18; Mem+19], weak memory concurrency [Bat+11; Lah+17; NMS16], non-local control flow [KW13], verified compilation [Ler09; Ste+15], *etc.*

The focus of this paper—non-determinism in C expressions—has been treated formally a number of times, notably by Norrish [Nor98], Ellison and Rosu [ER12], Krebbers [Kre16], and Memarian *et al.* [Mem+16]. The first three have in common that they model the sequence point restriction by keeping track of the locations that have been written to. The treatment of sequence points in our definitional semantics is closely inspired by the work of Ellison and Rosu [ER12], which resembles closely what is in the C standard. Krebbers [Kre16] used a more restrictive version of the semantics by Ellison and Rosu—he assigned undefined behavior in some corner cases to ease the soundness theorem of his logic. We directly proved soundness of the logic w.r.t. the more faithful model by Ellison and Rosu.

Memarian *et al.* [Mem+16] give a semantics to C by elaboration into a language they call Core. Unspecified evaluation order in Core is modeled using an unseq operation, which is similar to our ‖ operation. Compared to our translation, Core is much closer to C (it has function calls, memory operations, *etc.* as primitives, while we model them with monadic combinators), and supports concurrency.

3.8.2 Reasoning tools and program logics for C

Apart from formalizing the semantics of C, there have been many efforts to create reasoning tools for the C language in one way or another. There are standalone tools, like VeriFast [JSP10], VCC [Coh+09a], and the Jessie plugin of Frama-C [MM11], and there are tools built on top of general purpose proof assistants like VST [App14; Cao+18] in Coq, or AutoCorres [Gre+14] in Isabelle/HOL. Although, admittedly, all of these tools cover larger subsets of C than we do, as far as we know, they all ignore non-determinism in expressions.

There are, however, a few exceptions. Norrish proved confluence for a certain class of C expressions [Nor99]. Such a confluence result may be used to justify proofs in a tool that does not have an underlying non-deterministic semantics.

Another exception is the separation logic for non-determinism in C by Krebbers [Kre14]. Our work is inspired by his, but there are several notable differences:

- We have proved soundness with respect to a definitional semantics for a subset of C. We believe that this approach is more modular, since the semantics can be specified at a higher level of abstraction.
- We have built our logic on top of the Iris framework. This makes the development more modular (since we can use all the features as well as the Coq infrastructure of Iris) and more expressive (as shown in Section 3.7).
- Reasoning in Krebbers's logic directly was infeasible. There was no automation like our vcgen, so one had to subdivide resources between subexpressions manually all the time. Also, there was not even tactical support for carrying out proofs manually. Our logic is redesigned to get such support from the Iris Proof Mode/MoSeL framework.

To handle missing features of C as part of our vcgen, we plan to explore approaches by other verification projects in proof assistants. A notable example of such a project is VST, which supports machine arithmetic [DA13] and data types like structs and unions [Cao+18] as part of its tactics for symbolic execution.

3.8.3 Separation logic and symbolic execution

In their seminal work, Berdine *et al.* [BCO05] demonstrate the application of symbolic execution to automated reasoning in separation logic. In their setting, frame inference is used to perform symbolic execution of function calls. The frame has to be computed when the call site has more resources than needed to invoke a function. In our setting we compute frames for subexpressions, which, unlike functions, do not have predefined specifications. Due to that, we have to perform frame inference simultaneously with symbolic execution. The symbolic execution algorithm of Berdine *et al.* can handle inductive predicates, and can be extended with shape analysis [DOY06]. We do not support such features, and leave them to future work.

Caper [Din+17] is a tool for automated reasoning in concurrent separation logic, and it also deals with non-determinism, although the nature of non-determinism in Caper is different. Non-determinism in Caper arises due to branching on unknown conditionals and due to multiple possible ways to apply ghost state related rules (rules pertaining to abstract regions and guards). The former cause is tackled by considering sets of symbolic execution traces, and the latter is resolved by employing heuristics based on bi-abduction [Cal+11]. Applications of abductive reasoning to our approach to symbolic execution are left for future work.

Recently, Bannister *et al.* [BHK18; BH18] proposed a new separation logic connective for performing forwards reasoning whilst avoiding frame inference. This approach, however, is aimed at sequential deterministic programs, focusing on a notion of partial correctness that allows for failed executions. Another approach to verification of sequential stateful programs is based on characteristic formulae [Cha11]. A stateful program is transformed into a higher-order logic predicate, implicitly encoding the frame rule. The resulting formula is then proved by a user in Coq.

When implementing a vcgen in a proof assistant (see *e.g.*, [Mal14; Cao+18]) it is common to let the vcgen return a new goal when it gets stuck, from which the user can help out and call back the vcgen. The novelty of our work is that this approach is applied to operations that are called in parallel.

4 ReLoC: a logic for proving contextual refinements

4.1 Introduction

A fundamental question in computer science is *when two programs are equivalent*? The "golden standard" of program equivalence is *contextual equivalence*, stated directly in terms of the operational semantics. Intuitively, expressions e and e' are contextually equivalent if no well-typed client can distinguish them, which formally means that for all well-typed contexts $\mathcal{C}$, the expression $\mathcal{C}[\,e\,]$ has same observable behaviors as $\mathcal{C}[\,e'\,]$. Contextual equivalence can be further decomposed into *contextual refinement*. An expression e *contextually refines* e' if, for all contexts $\mathcal{C}$, if $\mathcal{C}[\,e\,]$ has some observable behavior, then so does $\mathcal{C}[\,e'\,]$. Expressions e and e' are contextually equivalent iff e contextually refines e' and *vice versa*.

Contextual refinement and contextual equivalence have many applications in computer science. One such application is to specify programs in terms of other programs. For example, one can specify an implementation of a program module (say, a map) that internally uses an efficient but complicated data structure (say, a balanced search tree) by stating that it refines an implementation that internally uses an inefficient but easy to understand data structure (say, an unordered list). In the context of a typed language that supports data abstraction, a specification of a program module in terms of refinement shows that clients of the program module cannot depend on the internal representation of data. This can be seen as an instance of the *representation independence* principle [Rey74; Mit86].

In the context of concurrency, contextual refinement is often used to specify a fine-grained concurrent program module by stating that it contextually refines a coarse-grained version. This is similar to showing that a fine-grained program module is *linearizable* [HW90; Fil+10], *i.e.*, each fine-grained operation appears to take place instantaneously. A simple example is the specification of a fine-grained concurrent counter by a coarse-grained one, see Figure 4.1 for the code. The increment operation of the fine-grained version, $\mathsf{counter}_i$, takes an "optimistic" lock-free approach to incrementing the value using a compare-and-set operation inside a loop. If the value of the counter has been changed (for instance, by some other thread), then the fine-grained counter reattempts the increment from the beginning. The increment operation of the coarse-grained version, $\mathsf{counter}_s$, is performed inside a critical

Fine-grained grained version (*i.e.*, the implementation):

$$\begin{aligned}
\mathsf{read} &\triangleq \lambda c.\ !c \\
\mathsf{inc}_i &\triangleq \mathsf{rec}\ \mathit{inc}\ c = \mathsf{let}\ n = !c\ \mathsf{in} \\
&\qquad\qquad \mathsf{if}\ \mathsf{CAS}(c, n, 1+n)\ \mathsf{then}\ n\ \mathsf{else}\ \mathit{inc}\ c \\
\mathsf{counter}_i &\triangleq \mathsf{let}\ c = \mathsf{ref}(0)\ \mathsf{in}\ ((\lambda().\ \mathsf{read}\ c), (\lambda().\ \mathsf{inc}_i\ c))
\end{aligned}$$

Coarse-grained version (*i.e.*, the specification):

$$\begin{aligned}
\mathsf{inc}_s &\triangleq \lambda c\, l.\ \mathsf{acquire}\ l;\ \mathsf{let}\ n = !c\ \mathsf{in}\ c \leftarrow (1+n);\ \mathsf{release}\ l;\ n \\
\mathsf{counter}_s &\triangleq \mathsf{let}\ l = \mathsf{newlock}\ ()\ \mathsf{in}\ \mathsf{let}\ c = \mathsf{ref}(0)\ \mathsf{in}\ ((\lambda().\ \mathsf{read}\ c), (\lambda().\ \mathsf{inc}_s\ c\ l))
\end{aligned}$$

Figure 4.1: A fine-grained and coarse-grained concurrent counter. (Note that the read operation is shared by both.)

section guarded by a lock. We can state the desired refinement as follows:

$$\mathsf{counter}_i \precsim_{ctx} \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

Due to the instrumentation of the coarse-grained version with locks, this refinement expresses that each operation of the fine-grained version takes place instantaneously. We will use the counter as a simple running example throughout the paper.

Another application of contextual refinement and contextual equivalence is to state algebraic properties of program constructs. For example, let us consider the non-deterministic choice operator $e_1 \oplus e_2$, which non-deterministically executes the expression e_1 or e_2. Using contextual equivalence, we can state that this operator is commutative ($e_1 \oplus e_2 \simeq_{ctx} e_2 \oplus e_1$), associative ($e_1 \oplus (e_2 \oplus e_3) \simeq_{ctx} (e_1 \oplus e_2) \oplus e_3$), and that sequential composition distributes over the operator ($(e_1 \oplus e_2); e_3 \simeq_{ctx} (e_1; e_3) \oplus (e_2; e_3)$).

Proving contextual refinement and contextual equivalence

Contextual refinement $e \precsim_{ctx} e' : \tau$ (and contextual equivalence $e \simeq_{ctx} e' : \tau$) are very strong notions because they relate the expressions e and e' in *any* well-typed context $\mathcal{C}$ with a hole of type τ. As a consequence, proving contextual refinement and equivalence directly is challenging—one has to consider arbitrary contexts $\mathcal{C}$, which are only known to be well-typed. Contextual refinement and equivalence are therefore typically proved indirectly using approaches based on bisimulations (*e.g.*, [Gor99; Pit00; KW06; SP07]) or logical relations (*e.g.*, [Pit05; Ahm06; DAB09; BST12; Tur+13]). In the present paper we focus on approaches based on logical relations because they scale well to increasingly rich programming languages with features such as impredicative polymorphism, recursive types, higher-order state, and fine-grained concurrency.

In the approaches based on logical relations, the key is a notion of *logical refinement*, notation $e \precsim e' : \tau$. Logical refinement is defined by structural recursion over the type τ, rather than by quantification over all contexts. The soundness theorem of logical relations states that logical refinement implies contextual refinement, *i.e.*, that $e \precsim e' : \tau$ implies $e \precsim_{ctx} e' : \tau$. As a result, proving contextual refinement can be reduced to proving logical refinement, which is generally much easier.

Unfortunately, it is difficult to construct a suitable notion of logical refinement when considering language features like recursive types and higher-order state. In the presence of (general) recursive types, no structurally-recursive definition over the type exists, and in the presence of higher-order references, one needs some notion of *recursively-defined worlds* [Bir+11]. The technique of *step-indexing* [AAV02; Ahm04] has been used to stratify the definitions using recursion over a natural number, called the *step-index*, which corresponds to the number of computation steps performed by the program.

Step-indexing has shown to be very effective by a large body of work on step-indexed logical relations, *e.g.*, [NDR11; HD11; BST12; ÇPG16; RG18]. However, definitions and proofs are intricate because step-indices appear practically everywhere—they even appear in definitions and proofs related to language features (say, products or sums) for which step-indexing is orthogonal. Dreyer *et al.*thus proposed the "logical approach" to logical relations [DAB09; Dre+10] to hide step-indices by abstracting and internalizing them in a modal logic using the *later* modality ($\triangleright$) [App+07]. Turon *et al.* [Tur+13; TDB13] further developed the logical approach by using separation logic [ORY01; OHe07; Bro07] to abstract over program states and to handle (fine-grained) concurrency.

More recently, Krebbers *et al.* [KTB17] and Timany [Tim18] defined a logical relation for program refinement based on the work of Turon *et al.*in the state-of-the-art higher-order concurrent separation logic Iris [Jun+18b; Jun+15; Jun+16; Kre+17]. Iris supports impredicative invariants [SB14] and used-defined ghost state, which can be used to streamline the definition of the logical relation, and to carry out proofs of challenging program refinements. The meta theory of Iris is mechanized in the Coq proof assistant, and Iris comes equipped with a *proof mode* [KTB17; Kre+18]—an extensive set of Coq tactics for separation logic proofs—which allowed them to mechanize all their results in Coq.

Problem statement and key idea

To prove refinements of complicated program modules in a scalable fashion, it is important to decompose refinement proofs into smaller refinements that can be proved in isolation. As a simple example, let us consider the refinement of the fine-grained and coarse-grained concurrent counter from Figure 4.1:

$$\mathsf{counter}_i \precsim_{ctx} \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

We wish to decompose the proof of this refinement into refinements for the read and increment operations. Naively, one might consider proving contextual refinements for these operations. Unfortunately, such contextual refinements do not hold—they

only hold *conditionally* under the assumption that the internal state in both of the implementations is related (including the state of the lock used by the coarse-grained version).

Instead of performing composition at the level of contextual refinement, our key idea is to perform composition at the level of logical refinement. By generalizing logical refinement to become an internal (*i.e.*, first-class) notion in (the Iris) separation logic, we can use the connectives of separation logic to express conditional refinements. Logical refinements for the operations of the concurrent counter are as follows:

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} (\lambda().\ \mathsf{read}\ c_i) \precsim (\lambda().\ \mathsf{read}\ c_s) : \mathsf{unit} \to \mathsf{int}$$
$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} (\lambda().\ \mathsf{inc}_i\ c_i) \precsim (\lambda().\ \mathsf{inc}_s\ c_s\ lk) : \mathsf{unit} \to \mathsf{int}.$$

We use the *magic wand* ($-\!\!*$, also known as *separating implication*) to make these refinements conditional under the invariant I_{cnt} (expressed using Iris's invariant connective $\boxed{I}$), which is defined as $I_{\mathsf{cnt}} \triangleq \exists n \in \mathbb{N}.\ c_i \mapsto_{\mathsf{i}} n * c_s \mapsto_{\mathsf{s}} n * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false})$. The invariant I_{cnt} intuitively expresses that in between function calls, the values of both counters are equal, and the lock (used in the coarse-grained implementation) is in unlocked state. With logical refinements for the individual operations at hand, we can compose them into the logical refinement $\mathsf{counter}_i \precsim \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int})$, which using soundness gives us the desired contextual refinement $\mathsf{counter}_i \precsim_{ctx} \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int})$.

Treating logical refinement as an internal notion in separation logic succinctly distinguishes our work from prior work. In prior work on refinements for rich languages, *e.g.*, the aforementioned work by Turon *et al.* [Tur+13; TDB13], Krebbers *et al.* [KTB17], and Timany [Tim18], logical refinement is an external notion (*i.e.*, a proposition in ordinary mathematics, rather than in separation logic), which means that one cannot concisely state refinements that are conditional on the program state. To state and prove such refinements, one needs to unfold the definition of the logical refinement into the model.

Apart from being able to decompose refinement proofs, internalizing logical refinement gives us a number of other tangible benefits. First, it allows us to develop type-directed structural rules and symbolic execution rules for proving logical refinements. Our symbolic execution rules closely resemble the typical rules for symbolic execution in separation logic, but come in two forms: for the program on the left-hand side and right-hand side of the refinement, making it possible to write concise proofs.

Second, by internalizing logical refinement we can state logical refinements that apply to the situation when the expression on the one side of the refinement contains a program subject to specification, while the expression on the other side is arbitrary. We call such specifications *relational specifications*. Relational specifications take the ability to decompose refinement proofs one step further. As a simple example, let us consider the example from Figure 4.1, where we proved that a fine-grained concurrent counter refines a coarse-grained version. This refinement is insufficient if we want to prove that a program module that *uses internally* the fine-grained counter (say, a ticket lock) refines another module that does not use the coarse-grained counter

(say, a spin lock). However, we can instead formulate a *relational specification* for the program module that is proven just once, and derive different logical (and thus by soundness, contextual) refinements from it.

A key challenge in stating relational specifications for operations is to concisely capture that they behave as-if they were atomic, *i.e.*, they appear to take place instantaneously. There has been a long line of work on *logically atomic specifications* to reason about atomicity in the context of Hoare-style logics [JP11; SBP13; RDG14; Jun+15; Jun+20]. We show that such logically atomic specifications generalize to the relational case, and call them *logically atomic relational specifications*. Concretely, we introduce relational specification patterns based on da Rocha Pinto *et al.*'s TaDA-style [RDG14] and Svendsen *et al.*'s HOCAP-style [SBP13] logically atomic specifications.

The ReLoC logic

Based on the previously described key ideas, we develop a relational separation logic called **ReLoC**. ReLoC is built on top of Iris, allowing the user to leverage the features of Iris such as invariants, (higher-order) ghost state, and prophecy variables. Invariants and ghost state state are powerful mechanisms that support reasoning about concurrent programs through used-defined protocols. Prophecy variables [AL91; Jun+20] allow for speculative reasoning about the future state of concurrent programs. In Iris they come in the form of ghost variables whose value can be referenced before they are specified, thus allowing one to "prophesize" their potential value. We show how these features can be used in ReLoC to prove challenging refinements.

We have implemented ReLoC as a shallow embedding on top of Iris in Coq [KTB17; Kre+18]. In addition to mechanizing all meta-theoretic results of ReLoC, like its soundness theorem, we have implemented new tactics that support mechanized interactive reasoning about program refinements in ReLoC in a practical and modular way. To our knowledge, ReLoC is the first fully mechanized relational logic enabling reasoning about contextual refinements of programs in a fine-grained concurrent higher-order imperative programming language. The mechanization can be found at [FKB21a].

Contributions and structure of the paper

- We present a relational logic **ReLoC** for reasoning about contextual refinements of fine-grained concurrent higher-order imperative programs. We present our target programming language (Section 4.2), an overview of ReLoC (Section 4.3), and a detailed description of its type-directed structural rules and symbolic execution rules (Section 4.4).
- We introduce relational specification patterns based on TaDA [RDG14] and HOCAP-style [SBP13] logically atomic specifications (Section 4.5).
- We show how to integrate *prophecy variables* into ReLoC, thereby enabling speculative reasoning in proofs of program refinements (Section 4.6).
- We describe the logical relations model of ReLoC in Iris (Section 4.7).

- We describe the mechanization of ReLoC in Coq, and explain how we support mechanized interactive reasoning in ReLoC in a practical and modular way (Section 4.8).

We discuss further related work in Section 4.9 and conclude in Section 4.10.

In addition to the case studies presented in this paper, we have also verified a collection of refinements of concurrent programs from the literature. We give a brief overview of these examples in Section 4.10.1; and the proofs can be found in the accompanying Coq sources.

Differences with the conference version of this paper

In the conference version of this paper [FKB18] we described the first version of ReLoC. This paper extends the conference paper in two ways. First, we introduce ReLoC Reloaded (in this paper referred to as just ReLoC), which has several new features, especially in terms its Coq mechanization. Second, we have expanded the presentation of, as well as the material covered by, the paper significantly. Concretely, ReLoC Reloaded has the following new features compared to its original version:

- ReLoC Reloaded's primitive refinement judgment $e \precsim e' : \tau$ is defined for closed expressions (*i.e.*, without free variables), and the version for open expressions (*i.e.*, with free variables) is a derived notion (see Definition 4.5).
- ReLoC Reloaded's underlying programming language is HeapLang—the default language of Iris's Coq mechanization. By having a tight integration of ReLoC with Iris's Coq ecosystem we managed to reuse more Coq code and integrate novel Iris features.
- One such feature that we have integrated into ReLoC Reloaded is the support for prophecy variables (Section 4.6), which was recently added to Iris [Jun+20].

Compared to the conference paper, we have significantly expanded Sections 4.2, 4.4 and 4.8, and added Sections 4.6, 4.7 and 4.10.2, which are completely new. We have extended Section 4.5 with HOCAP-style specifications, which we put into action by verifying a refinement between a ticket lock and a spin lock in Section 4.5.5.

4.2 The programming language

We consider a typed version of HeapLang, the default language that is shipped with Iris's Coq development [Iri20]. HeapLang is a call-by-value λ-calculus, with higher-order references, fork-based unstructured concurrency, and atomic operations for fine-grained concurrency. We equip the untyped version of HeapLang (which was given in Chapter 2) with System-F-style types and some corresponding expressions. The syntax is shown in Figure 4.2. We let α range over a countable infinite set *TVar* of type variables, which can be bound by the universal type $\forall\alpha.\tau$, existential type $\exists\alpha.\tau$, and recursive type $\mu\alpha.\tau$. We omit the usual Boolean and arithmetic operations such as addition, multiplication, equality, negation.

Most of the operations are the same as in Chapter 2, so we only discuss the newly added operations. The new operations are type abstraction $\Lambda.e$, type application $e\langle\rangle$;

$$
\begin{array}{rll}
\tau \in \mathit{Type} ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 & \text{(STLC)} \\
& \mid \alpha \mid \forall \alpha.\tau \mid \exists \alpha.\tau & \text{(Polymorphic and existential types)} \\
& \mid \mu\alpha.\tau & \text{(Recursive types)} \\
& \mid \mathsf{ref}\ \tau & \text{(Reference types)} \\
v \in \mathit{Val} ::= & i \mid \ell \mid \mathbf{true} \mid \mathbf{false} \mid (v_1, v_2) \mid \mathbf{inl}\ (v) \mid \mathbf{inr}\ (v) \mid \mathsf{rec}\ f\ x = e & i \in \mathbb{Z}, \ell \in \mathit{Loc} \\
& \mid \Lambda.e & \text{(Type abstraction)} \\
& \mid \mathsf{pack}\ v \mid \mathsf{fold}\ v & \\
e \in \mathit{Expr} ::= & x \mid v \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid (e_1, e_2) \mid \pi_i(e) \mid \mathbf{inl}\ (e) \mid \mathbf{inr}\ (e) & i \in \{1,2\} \\
& \mid (\mathsf{match}\ e\ \mathsf{with}\ \mathbf{inl}\ (x) \to e_1 \mid \mathbf{inr}\ (x) \to e_2) \mid e_1(e_2) & \\
& \mid e\langle\rangle & \text{(Type application)} \\
& \mid \mathsf{pack}\ e \mid \mathsf{unpack}\ e_1\ \mathsf{in}\ x.\ e_2 & \text{(Existential types operations)} \\
& \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e & \text{(Recursive types operations)} \\
& \mid \mathsf{ref}(e) \mid\ !e \mid e_1 \leftarrow e_2 \mid \mathsf{CAS}(e_1, e_2, e_3) & \text{(Reference types operations)} \\
& \mid \mathsf{fork}\ \{e\} \mid \ldots &
\end{array}
$$

Figure 4.2: The syntax of the extended HeapLang language.

the pack/unpack constructs for packing/unpacking existential types; the fold/unfold constructs for iso-recursive types. Type level lambdas and type applications do not contain type annotations, following *e.g.*, [Ahm06].

Syntactic sugar. We use syntactic sugar to define non-recursive functions, let-bindings, and sequential composition. We let $(\lambda x.\ e) \triangleq (\mathsf{rec}\ \_\ x = e)$ and $(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \triangleq ((\lambda x.\ e_2)\ e_1)$ and $(e_1; e_2) \triangleq (\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2)$. The underscore _ denotes an anonymous binder, *i.e.*, a fresh variable that is unused in the body of the binding expression.

Type system. Typing judgments take the form $\Xi \mid \Gamma \vdash e : \tau$, where Γ is a context assigning types to program variables, and Ξ is a context of type variables. The inference rules for the typing judgments are standard; a selection of representative rules is given in Figure 4.3. The typing rule for the compare-and-set (CAS) operation has a side-condition unboxed(τ), which ensures that a compare-and-set can only be performed on word-sized data types, *i.e.*, the unit, Boolean, integer, and reference type.

Operational semantics. Recall from Chapter 2 that the operational semantics consists of three reduction relations: pure head reduction $\to_{\mathsf{pure}}$, head reductions $\to_{\mathsf{h}}$, thread-local reductions $\to_{\mathsf{t}}$, and thread-pool reduction $\to_{\mathsf{tp}}$.

Pure head reductions are head reductions that do not modify the state. They are the same as in Chapter 2, with the exception of reductions for newly added constructs,

Selected typing rules:

$$\textsc{var-typed}\ \frac{\Gamma(x) = \tau}{\Xi \mid \Gamma \vdash x : \tau} \qquad \textsc{proj-typed}\ \frac{\Xi \mid \Gamma \vdash e : \tau_1 \times \tau_2 \qquad i \in \{1,2\}}{\Xi \mid \Gamma \vdash \pi_i(e) : \tau_i} \qquad \textsc{rec-typed}\ \frac{\Xi \mid x{:}\tau_1, f{:}\tau_1 \to \tau_2, \Gamma \vdash e : \tau_2}{\Xi \mid \Gamma \vdash \mathsf{rec}\ f\ x = e : \tau_1 \to \tau_2}$$

$$\textsc{tlam-typed}\ \frac{\Xi, \alpha \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \Lambda.e : \forall \alpha.\tau} \qquad \textsc{tapp-typed}\ \frac{\Xi \mid \Gamma \vdash e : \forall \alpha.\tau \qquad \Xi \vdash \tau'}{\Xi \mid \Gamma \vdash e\langle\rangle : \tau[\tau'/\alpha]} \qquad \textsc{tpack-typed}\ \frac{\Xi \mid \Gamma \vdash e : \tau[\tau'/\alpha]}{\Xi \mid \Gamma \vdash \mathsf{pack}\ e : \exists \alpha.\tau}$$

$$\textsc{tunpack-typed}\ \frac{\Xi \mid \Gamma \vdash e_1 : \exists \alpha.\tau_1 \qquad \alpha, \Xi \mid x{:}\tau_1, \Gamma \vdash e_2 : \tau_2 \qquad \alpha \text{ is not free in } \Gamma \text{ or } \tau_2}{\Xi \mid \Gamma \vdash \mathsf{unpack}\ e_1\ \mathsf{in}\ x.\ e_2 : \tau_2}$$

$$\textsc{fold-typed}\ \frac{\Xi \mid \Gamma \vdash e : \tau[\mu\tau/\alpha]}{\Xi \mid \Gamma \vdash \mathsf{fold}\ e : \mu\alpha.\tau} \qquad \textsc{unfold-typed}\ \frac{\Xi \mid \Gamma \vdash e : \mu\alpha.\tau}{\Xi \mid \Gamma \vdash \mathsf{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]} \qquad \textsc{alloc-typed}\ \frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathsf{ref}(e) : \mathsf{ref}\ \tau}$$

$$\textsc{load-typed}\ \frac{\Xi \mid \Gamma \vdash e : \mathsf{ref}\ \tau}{\Xi \mid \Gamma \vdash\ !e : \tau} \qquad \textsc{store-typed}\ \frac{\Xi \mid \Gamma \vdash e_1 : \mathsf{ref}\ \tau \qquad \Xi \mid \Gamma \vdash e_2 : \tau}{\Xi \mid \Gamma \vdash e_1 \leftarrow e_2 : \mathsf{unit}}$$

$$\textsc{cas-typed}\ \frac{\Xi \mid \Gamma \vdash e_1 : \mathsf{ref}\ \tau \quad \Xi \mid \Gamma \vdash e_2 : \tau \quad \Xi \mid \Gamma \vdash e_3 : \tau \quad \mathrm{unboxed}(\tau)}{\Xi \mid \Gamma \vdash \mathsf{CAS}(e_1, e_2, e_3) : \mathsf{bool}} \qquad \textsc{fork-typed}\ \frac{\Xi \mid \Gamma \vdash e : \mathsf{unit}}{\Xi \mid \Gamma \vdash \mathsf{fork}\ \{e\} : \mathsf{unit}}$$

New pure reductions $e_1 \to_{\mathsf{pure}} e_2$:

$$(\Lambda.e)\langle\rangle \to_{\mathsf{pure}} e \qquad \mathsf{unpack}\ (\mathsf{pack}\ v)\ \mathsf{in}\ x.\ e \to_{\mathsf{pure}} e[v/x] \qquad \mathsf{unfold}\ (\mathsf{fold}\ v) \to_{\mathsf{pure}} v$$

Figure 4.3: The type system and operational semantics of HeapLang.

as shown in Figure 4.3. Head reductions are lifted to thread-local reductions using call-by-value evaluation contexts $K \in ECtx$. The evaluation contexts are the same as evaluations context from Chapter 2, but extended to cover newly added operations:

$$\begin{aligned} K \in ECtx ::= {} & [\bullet] \mid e_1(K) \mid K(v_2) \mid e_1 \leftarrow K \mid K \leftarrow v_2 \\ & \mid K\langle\rangle \mid \mathsf{pack}\ K \mid \mathsf{unpack}\ K\ \mathsf{in}\ x.\ e_2 \mid \mathsf{fold}\ K \mid \mathsf{unfold}\ K \mid \ldots \end{aligned}$$

Thread-pool reductions $\to_{\mathsf{tp}}$ are defined on configurations $\rho = (\vec{e}, \sigma)$ consisting of a state σ (a finite partial map from locations to values) and a thread-pool $\vec{e}$ (a list of expressions corresponding to the threads) by interleaving, *i.e.*, by picking a thread and executing it, thread-locally, for one step.

Contextual refinement. The notion of contextual refinement that we use is standard (see, *e.g.*, [Pit05] or [Har16, Chapters 46 & 47]). It formalizes the situation when the set of observations that can be made about the first program is a subset of observations that can be made about the second program. An observation about a program are made using a *program context* $\mathcal{C}$, which is a program with a hole:

$$\mathcal{C} \in Ctx ::= \square \mid \mathsf{rec}\ f\ x = \mathcal{C} \mid \mathcal{C}(e_2) \mid e_1(\mathcal{C}) \mid \Lambda.\mathcal{C} \mid \mathcal{C}\langle\rangle \mid \ldots$$

In contrast with evaluation contexts K, program contexts may contain a hole under binders.

Since we are in a typed setting, we consider only *typed contexts*. A program context is well-typed, denoted as $\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\Xi' \mid \Gamma' \vdash \tau')$, if for any term $\Xi \mid \Gamma \vdash t : \tau$ we have $\Xi' \mid \Gamma' \vdash \mathcal{C}[\,t\,] : \tau'$. The typing relation on contexts is standard, and can be derived from the typing rules in Figure 4.3.

We then define contextual refinement as follows. An expression e_1 *contextually refines* an expression e_2 at type τ, denoted as $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$, if no well-typed *program context* $\mathcal{C}$ resulting in a closed program can distinguish the two:

$$\begin{aligned}\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau \triangleq \forall \tau'\, (\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\emptyset \mid \emptyset \vdash \tau'))\ v\ \vec{e_f}\ \sigma.\\ (\mathcal{C}[\,e_1\,], \emptyset) \to^*_{\mathsf{tp}} (v :: \vec{e_f}, \sigma) \implies\\ \exists v'\ \vec{e_f'}\ \sigma'.\ (\mathcal{C}[\,e_2\,], \emptyset) \to^*_{\mathsf{tp}} (v' :: \vec{e_f'}, \sigma').\end{aligned}$$

Contextual equivalence $\Xi \mid \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$ is defined as the symmetric closure of contextual refinement, *i.e.*, $(\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau) \wedge (\Xi \mid \Gamma \vdash e_2 \precsim_{ctx} e_1 : \tau)$.

Note that contextual refinement only takes termination into account, and does not require the resulting values v and v' to be equal. Demanding the equality on the resulting values would make contextual refinement too strong. For example, the terms $(\lambda x.\, x + 1)$ and $(\lambda x.\, 1 + x)$ of function type would not be deemed contextually equivalent, because they terminate to syntactically different values in the empty program context.

There are, however, equivalent formulations of contextual refinement which equate the resulting values v and v'. In order to do that, it is necessary to restrict the typed context $\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\emptyset \mid \emptyset \vdash \tau')$ to those for which τ' is a directly observable type, like Booleans or integers. For example, we could have used the following equivalent[1] definition (a variation of **true**-adequate contextual equivalence from [Pit05, Exercise 7.5.10]):

$$\begin{aligned}\forall (\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\emptyset \mid \emptyset \vdash \mathsf{bool}))\ \vec{e_f}\ \sigma.\\ (\mathcal{C}[\,e_1\,], \emptyset) \to^*_{\mathsf{tp}} (\mathbf{true} :: \vec{e_f}, \sigma) \implies \exists \vec{e_f'}\ \sigma'.\ (\mathcal{C}[\,e_2\,], \emptyset) \to^*_{\mathsf{tp}} (\mathbf{true} :: \vec{e_f'}, \sigma').\end{aligned}$$

4.3 A tour of ReLoC

This section gives a tour of ReLoC by demonstrating its key logical connectives and proof rules. We first describe ReLoC's grammar, soundness statement, and rule format

[1] Proving that this definition is equivalent to the one presented earlier is not a very complicated, albeit laborious, task. See the Coq mechanization for the formal proof.

(Section 4.3.1). After that, we put ReLoC to action by proving contextual refinements of two program modules. The first is a bit module, which demonstrates ReLoC's type-directed structural rules and symbolic execution rules for reasoning about pure programs (Section 4.3.3). The second is the concurrent counter module from Section 4.1, which involves reasoning about internal state and concurrency. Specifically we demonstrate how ReLoC is used to reason about stateful programs using symbolic execution (Section 4.3.4.1), concurrency using invariants (Section 4.3.4.2), and recursive functions and loops using Löb induction (Section 4.3.4.3).

4.3.1 Grammar and soundness

ReLoC is based on higher-order intuitionistic separation logic, and the grammar of its propositions is:

$$\begin{aligned} P, Q \in \mathit{iProp} ::= {} & \mathsf{True} \mid \mathsf{False} \mid \forall x.\, P \mid \exists x.\, P \mid P \wedge Q \mid P \vee Q \mid P \implies Q \\ & \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto_{\mathsf{i}} v \mid \ell \mapsto_{\mathsf{s}} v \mid (\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau) \\ & \mid [\![\tau]\!]_\Delta(v_1, v_2) \mid \boxed{P}^{\mathcal{N}} \mid \triangleright P \mid \Box P \mid {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \mid \ldots \end{aligned}$$

ReLoC is an extension of Iris and therefore includes all connectives of Iris, in particular, the *later* modality $\triangleright$, *persistence* modality $\Box$, *update* modality ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$, and *invariant assertion* $\boxed{P}^{\mathcal{N}}$. We introduce these connectives in passing throughout this section. Some of these connectives are annotated by *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$ and *invariant names* $\mathcal{N} \in \mathit{InvName}$, which are needed for bookkeeping related to Iris's invariant mechanism. Until we introduce invariants in Section 4.3.4.2, we will omit these annotations. Similarly, we will ignore the later modality $\triangleright$ until we explain it in Section 4.3.4.3.

An essential difference to vanilla Iris is that ReLoC has internal (or first-class) *refinement judgments* $\Delta \models e_1 \precsim e_2 : \tau$, which should be read as "the expression e_1 refines the expression e_2 at type τ". Just like contextual refinement, the refinement judgment in ReLoC is indexed by a type τ. The judgment contains an environment Δ which assigns *interpretations* to type variables. These interpretations are given by an Iris relation of type $\mathit{Val} \times \mathit{Val} \to \mathit{iProp}$. One such kind of relation, the *value interpretation* relation $[\![\tau]\!]_\Delta(-,-) : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$ (for each syntactic type τ of HeapLang) will be discussed in Section 4.4. We elide the contexts Δ in refinement judgments whenever they are empty.

The intuitive meaning of $\Delta \models e_1 \precsim e_2 : \tau$ is that e_1 is safe, and all of its behaviors can be simulated by e_2. It is a simulation in the sense that any execution step of e_1 can be matched by a (possibly empty) sequence of execution steps of e_2. Borrowing the terminology from languages with non-determinism, we think of e_1 as being *demonic* and e_2 as being *angelic*. That is, the non-deterministic choices of e_1 (*e.g.*, scheduling of forked-off threads) are selected by an external demon; whereas for the non-deterministic choices of e_2, an angle blesses the person proving the refinement with an ability to select a choice themselves.

Since we often use refinement judgments to specify programs, we refer to the left-hand side e_1 as the *implementation*, and to right-hand side e_2 as the *specification*.

The intuitive meaning is formally reflected by the soundness theorem w.r.t. contextual refinement.

Theorem 4.1 (Soundness)**.** If the refinement judgment $\emptyset \models e_1 \precsim e_2 : \tau$ is derivable in ReLoC, then $\emptyset \mid \emptyset \vdash e_1 \precsim_{ctx} e_2 : \tau$.

In this section we only consider closed programs e_1 and e_2; we will see how ReLoC (and its soundness theorem) generalize to open terms in Section 4.4.4.

Like ordinary separation logic, ReLoC has *heap assertions*. Since ReLoC is relational, these come in two forms: $\ell \mapsto_{\mathsf{i}} v$ and $\ell \mapsto_{\mathsf{s}} v$, which signify ownership of a location ℓ with value v on the implementation and specification side, respectively.

Contrary to earlier work on logical refinements in Iris, *e.g.*, [KTB17; Tim18], refinement judgments $\Delta \models e_1 \precsim e_2 : \tau$ in ReLoC are first-class propositions. As such, we can combine them in arbitrary ways with the other logical connectives, and state conditional refinements. For example, the proposition

$$(\ell_1 \mapsto_{\mathsf{i}} v_1 * \ell_2 \mapsto_{\mathsf{s}} v_2 * \Delta \models e_1' \precsim e_2' : \sigma) \mathrel{-\!\!*} \Delta \models e_1 \precsim e_2 : \tau, \tag{4.1}$$

states that the e_1 refines e_2, under the assumption of another refinement and that certain locations have specified values in the heap. Having conditional refinements is crucial for modularity, as it allows us to formulate and prove refinements of individual methods of a data structure under the assumptions provided by the internal invariant of the data structure. The fact that refinement judgments are first class also plays an important role in the presentation of ReLoC's proof rules.

4.3.2 Derivability and inference rules

As standard in logic, Iris/ReLoC has a derivability relation $P \vdash Q$. We say that Q is derivable if $\mathsf{True} \vdash Q$. In many situations, we use magic wand $\mathrel{-\!\!*}$ instead of the derivability relation $\vdash$, because we have the standard deduction property:

$$P \vdash Q \mathrel{-\!\!*} R \qquad \text{iff} \qquad P * Q \vdash R$$

Most of the inference rules we present can be internalized as ReLoC propositions by a magic wand or a derivability relation between the separating conjunction of the antecedents and the consequent. We thus use the following notations:

$$\frac{P_1 \qquad \cdots \qquad P_n}{Q} \quad \text{is notation for} \quad (P_1 * \cdots * P_n) \mathrel{-\!\!*} Q,$$

$$\frac{P}{Q} \quad \text{is notation for} \quad (P \mathrel{-\!\!*} Q) \wedge (Q \mathrel{-\!\!*} P).$$

For instance, the conditional refinement in Formula (4.1) is presented as the following inference rule:

$$\frac{\ell_1 \mapsto_{\mathsf{i}} v_1 \qquad \ell_2 \mapsto_{\mathsf{s}} v_2 \qquad \Delta \models e_1' \precsim e_2' : \sigma}{\Delta \models e_1 \precsim e_2 : \tau}$$

In rules like this, it is useful to think of premises $\ell_1 \mapsto_{\mathsf{i}} v_1$ and $\ell_2 \mapsto_{\mathsf{s}} v_2$ as side conditions, and of the premise $\Delta \models e_1' \precsim e_2' : \sigma$ as the new goal that you get when you apply the rule. This *backwards-style* reasoning integrates well in the Coq proof assistant; we discuss it more in detail in Section 4.8.

We use the derivability relation $\vdash$ explicitly to state rules that cannot be internalized, *e.g.*, $\dfrac{\vdash P}{\vdash Q}$ states that if P is derivable, then Q is derivable. This is weaker than $\dfrac{P}{Q}$, which denotes that Q can be derived from P, *i.e.*, $P \vdash Q$.

4.3.3 Example: Contextual equivalance of a bit module

We demonstrate the basic usage of ReLoC by using its *type-directed structural* and *symbolic execution* rules to prove contextual equivalence of two implementations of a simple program module (representation independence). The module we consider represents a single bit data structure—it contains an initial value for the bit, an operation for flipping the bit, and an operation for converting the values of the abstract type to Booleans. We use an existential type (*i.e.*, abstract type) to hide the representation type and thus the type of the module:

$$\mathsf{TBit} \triangleq \exists \alpha.\, \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool}).$$

Perhaps the simplest implementation of the bit interface is the one that uses Booleans for the internal state:

$$\mathsf{bitbool} : \mathsf{TBit} \triangleq \mathsf{pack}(\mathbf{true}, (\lambda b.\ \neg b), (\lambda b.\ b)).$$

The second implementation models a bit by a number from the set $\{0, 1\}$:

$$\mathsf{flipnat} : \mathsf{int} \to \mathsf{int} \triangleq \lambda n.\ \mathsf{if}\ (n = 0)\ \mathsf{then}\ 1\ \mathsf{else}\ 0$$
$$\mathsf{bitnat} : \mathsf{TBit} \triangleq \mathsf{pack}(1, \mathsf{flipnat}, (\lambda n.\ n = 1)).$$

Before we explain how the contextual equivalence of these two implementation is formally proved in ReLoC, let us informally discuss why these implementations are equivalent. Note that the underlying types (int and bool) are not isomorphic. This, however, is not going to be a problem, because the underlying types are hidden/existentially abstracted in the module signature. As a consequence of that, a (well-typed) client has to be polymorphic in the type α, and can thus only create and modify values of α through the functions provided by the module. A client that uses the bitnat module can only construct integers 0 and 1 (using the initial value and applying the flip function a number of times). Thus, requiring an isomorphism between the underlying types is too strict—for example, we do not care what Boolean value an integer 7 might correspond to, because the number 7 can never be constructed using the functions provided by bitnat.

This intuitive reasoning signals the key idea behind the *representation independence* principle [Mit86], which states that in order to prove that two modules are equivalent, it suffices to pick a *relation* between the underlying types and demonstrate that all the methods preserve this relation. For this example, a sensible candidate for such a

Value interpretation rules:

$$\textsc{val-var}\ \frac{\Delta(\alpha)(v_1, v_2)}{[\![\alpha]\!]_\Delta(v_1, v_2)} \qquad \textsc{val-unit}\ \frac{v_1 = v_2 = ()}{[\![\mathsf{unit}]\!]_\Delta(v_1, v_2)} \qquad \textsc{val-bool}\ \frac{\exists b \in \mathbb{B}.\ v_1 = v_2 = b}{[\![\mathsf{bool}]\!]_\Delta(v_1, v_2)} \qquad \textsc{val-int}\ \frac{\exists n \in \mathbb{Z}.\ v_1 = v_2 = n}{[\![\mathsf{int}]\!]_\Delta(v_1, v_2)}$$

Type-directed structural rules:

$$\textsc{rel-return}\ \frac{[\![\tau]\!]_\Delta(v_1, v_2)}{\Delta \models v_1 \precsim v_2 : \tau} \qquad \textsc{rel-pair}\ \frac{\Delta \models e_1 \precsim e_2 : \tau \qquad \Delta \models e_1' \precsim e_2' : \sigma}{\Delta \models (e_1, e_1') \precsim (e_2, e_2') : \tau \times \sigma}$$

$$\textsc{rel-pack}\ \frac{\forall v_1, v_2.\ \mathsf{persistent}(R(v_1, v_2)) \qquad [\alpha := R], \Delta \models e_1 \precsim e_2 : \tau}{\Delta \models \mathsf{pack}\ e_1 \precsim \mathsf{pack}\ e_2 : \exists \alpha.\tau}$$

$$\textsc{rel-rec}\ \frac{\Box\Big(\forall v_1, v_2.\ [\![\tau]\!]_\Delta(v_1, v_2) \mathrel{-\!\!*} \Delta \models (\mathsf{rec}\ f_1\ x_1 = e_1)\ v_1 \precsim (\mathsf{rec}\ f_2\ x_2 = e_2)\ v_2 : \sigma\Big)}{\Delta \models (\mathsf{rec}\ f_1\ x_1 = e_1) \precsim (\mathsf{rec}\ f_2\ x_2 = e_2) : \tau \to \sigma}$$

Symbolic execution rules:

$$\textsc{rel-pure-l}\ \frac{e_1 \to_{\mathsf{pure}} e_1' \qquad \triangleright(\Delta \models K[\,e_1'\,] \precsim e_2 : \tau)}{\Delta \models K[\,e_1\,] \precsim e_2 : \tau} \qquad \textsc{rel-pure-r}\ \frac{e_2 \to_{\mathsf{pure}} e_2' \qquad \Delta \models_{\mathcal{E}} e_1 \precsim K[\,e_2'\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,e_2\,] : \tau}$$

$$\textsc{rel-alloc-l}'\ \frac{\forall \ell.\ \ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models K[\,\ell\,] \precsim e_2 : \tau}{\Delta \models K[\,\mathsf{ref}(v)\,] \precsim e_2 : \tau} \qquad \textsc{rel-alloc-r}\ \frac{\forall \ell.\ \ell \mapsto_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,\ell\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{ref}(v)\,] : \tau}$$

$$\textsc{rel-load-l-inv}\ \frac{\boxed{P}^{\mathcal{N}} \qquad \Big(\triangleright P * \mathsf{closeInv}_{\mathcal{N}}(P)\Big) \mathrel{-\!\!*} \exists v.\ \ell \mapsto_{\mathsf{i}} v * \triangleright\Big(\ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models_{\top \setminus \mathcal{N}} K[\,v\,] \precsim e_2 : \tau\Big)}{\Delta \models K[\,!\ell\,] \precsim e_2 : \tau}$$

$$\textsc{rel-load-r}\ \frac{\ell \mapsto_{\mathsf{s}} v \qquad \ell \mapsto_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,v\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,!\ell\,] : \tau} \qquad \textsc{rel-store-r}\ \frac{\ell \mapsto_{\mathsf{s}} - \qquad \ell \mapsto_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,()\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\ell \leftarrow v\,] : \tau}$$

$$\textsc{rel-cas-l-inv}\ \frac{\begin{array}{l}\boxed{P}^{\mathcal{N}} \quad \triangleright P * \mathsf{closeInv}_{\mathcal{N}}(P) \mathrel{-\!\!*} \\ \qquad \exists v.\ \ell \mapsto_{\mathsf{i}} v * \triangleright\left(\begin{array}{l}\Big(v = v_1 * \ell \mapsto_{\mathsf{i}} v_2 \mathrel{-\!\!*} \Delta \models_{\top \setminus \mathcal{N}} K[\,\mathsf{true}\,] \precsim e_2 : \tau\Big) \wedge \\ \Big(v \neq v_1 * \ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models_{\top \setminus \mathcal{N}} K[\,\mathsf{false}\,] \precsim e_2 : \tau\Big)\end{array}\right)\end{array}}{\Delta \models K[\,\mathsf{CAS}(\ell, v_1, v_2)\,] \precsim e_2 : \tau}$$

Figure 4.4: Selected rules of ReLoC.

Invariants rules:

$$\textsc{rel-inv-alloc}\ \frac{\triangleright P \qquad \boxed{P}^{\mathcal{N}} \twoheadrightarrow \Delta \models e_1 \precsim e_2 : \tau}{\Delta \models e_1 \precsim e_2 : \tau} \qquad \textsc{rel-inv-restore}\ \frac{\mathsf{closeInv}_{\mathcal{N}}(P) \qquad \triangleright P \qquad \Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau}{\Delta \models_{\mathcal{E}\setminus\mathcal{N}} e_1 \precsim e_2 : \tau}$$

Figure 4.4: Selected rules of ReLoC (cont.)

relation is $\{(\mathbf{true}, 1), (\mathbf{false}, 0)\}$. Note that our relation does not include any integers other than 0 or 1, because as we previously explained, a well-typed client of bitnat cannot construct other integers. With the relation at hand, the informal proof is as follows. The initial values offered by the modules are related. The flip function preserves this relation. The function that converts "bits" to Booleans sends related values to the same Boolean.

We will now demonstrate how to carry out this argument formally in ReLoC. Specifically, we prove the following refinement using the rules in Figure 4.4:

$$\mathsf{bitbool} \precsim \mathsf{bitnat} : \mathsf{TBit}$$

The other direction can be proved in a similar way, which using soundness (Theorem 4.1), gives us the contextual equivalence $\mathsf{bitbool} \simeq_{ctx} \mathsf{bitnat} : \mathsf{TBit}$.

From a high-level point of view, the proof of this example involves applying ReLoC's type-directed structural rules following the structure of TBit. At the leaves of the proof, we continue with ReLoC's symbolic execution rules to perform computation steps.

Since TBit is an existential type, and both bitbool and bitnat are pack's, we start off by applying the type-directed structural rule REL-PACK. For that we need to pick a relation R, which will be the interpretation for the type variable α, and should link together the underlying representations of bits in bitbool and bitnat. We define the relation R as follows:

$$R(b, n) \triangleq (b = \mathbf{true} \land n = 1) \lor (b = \mathbf{false} \land n = 0).$$

Starting with the initial goal $\mathsf{bitbool} \precsim \mathsf{bitnat} : \mathsf{TBit}$, we apply REL-PACK. As a side-condition, we have to prove that R is *persistent* for any v_1, v_2, written as $\mathsf{persistent}(R(v_1, v_2))$, intuitively meaning that the proposition $R(v_1, v_2)$ does not assert ownership of any resources. We discuss persistent propositions in more detail in Sections 4.3.4.2 and 4.4.2, and for now we just note that the relation R is indeed persistent. After application of the REL-PACK rule the goal becomes:

$$[\alpha := R] \models (\mathbf{true}, (\lambda b.\ \neg b), (\lambda b.\ b)) \precsim (1, \mathsf{flipnat}, (\lambda n.\ n = 1)) : \alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool}).$$

By repeatedly applying the type-directed structural rule REL-PAIR we get three new goals:

1. $[\alpha := R] \models \mathbf{true} \precsim 1 : \alpha$;

2. $[\alpha := R] \models (\lambda b.\, \neg b) \precsim \mathsf{flipnat} : \alpha \to \alpha$;
3. $[\alpha := R] \models (\lambda b.\, b) \precsim (\lambda n.\, n = 1) : \alpha \to \mathsf{bool}$.

For the first goal, we can use the rules REL-RETURN and VAL-VAR, leaving us with the obligation $R(\mathbf{true}, 1)$, which holds by the definition of R.

For the second and the third goal we need to prove refinements of two closures, for which we use the type-directed structural rule REL-REC. Let us look at the third goal in detail. After the application of REL-REC we have to show:

$$\square\Big(\forall v_1, v_2.\, [\![\alpha]\!]_{[\alpha:=R]}(v_1, v_2) \mathrel{-\!\!*} [\alpha := R] \models (\lambda b.\, b)\ v_1 \precsim (\lambda n.\, n = 1)\ v_2 : \mathsf{bool}\Big).$$

The goal is wrapped in Iris's *persistence modality* $\square$, which turns any proposition into a persistent one. Once again, we postpone the details about the persistence modality until Sections 4.3.4.2 and 4.4.2, and only remark that here we are allowed to prove the goal without the $\square$ modality. Using this information, and the rule VAL-VAR we reduce our goal to show:

$$R(v_1, v_2) \mathrel{-\!\!*} [\alpha := R] \models (\lambda b.\, b)\ v_1 \precsim (\lambda n.\, n = 1)\ v_2 : \mathsf{bool},$$

for arbitrary v_1, v_2. We then unfold the definition of R and observe that we need to distinguish two cases: 1. $v_1 = \mathbf{true}$ and $v_2 = 1$; 2. $v_1 = \mathbf{false}$ and $v_2 = 0$. Suppose we are in the first case (the second case is similar). We have to show:

$$[\alpha := R] \models (\lambda b.\, b)\ \mathbf{true} \precsim (\lambda n.\, n = 1)\ 1 : \mathsf{bool}.$$

At this point we apply ReLoC's *symbolic execution* rules: we symbolically reduce both the left-hand and the right-hand side of the refinement. For this we use the rules REL-PURE-L and REL-PURE-R (the later modalities ($\triangleright$) in these rules can be ignored for now, they will be explained in Section 4.3.4.3). These rules perform *pure reductions*, *i.e.*, reductions that do not depend on the heaps. In our case we have a β-reduction on the left-hand side, and a β-reduction and an evaluation of the binary operation (equality testing) on the right-hand side:

$$(\lambda b.\, b)\ \mathbf{true} \to_{\mathsf{pure}} \mathbf{true} \qquad (\lambda n.\, n = 1)\ 1 \to_{\mathsf{pure}} (1 = 1) \to_{\mathsf{pure}} \mathbf{true}.$$

After the repeated application of the said rules we arrive at a goal

$$[\alpha := R] \models \mathbf{true} \precsim \mathbf{true} : \mathsf{bool},$$

which we discharge by REL-RETURN and VAL-BOOL. This completes the proof of the refinement.

4.3.4 Example: Contextual refinement of a concurrent counter

The previous example showcased how ReLoC can be used to show contextual refinement and equivalence of pure program modules. In this subsection we prove contextual refinement of the fine-grained concurrent counter in Figure 4.1 from

$$\textsc{newlock-r}\ \frac{\forall lk.\ \mathsf{isLock_s}(lk, \mathbf{false}) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,lk\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{newlock}\ ()\,] : \tau}$$

$$\textsc{acquire-r}\ \frac{\mathsf{isLock_s}(lk, \mathbf{false}) \qquad \mathsf{isLock_s}(lk, \mathbf{true}) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,()\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{acquire}\ lk\,] : \tau}$$

$$\textsc{release-r}\ \frac{\mathsf{isLock_s}(lk, b) \qquad \mathsf{isLock_s}(lk, \mathbf{false}) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,()\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{release}\ lk\,] : \tau}$$

Figure 4.5: Right-hand side relational specification for locks.

Section 4.1 by showing that it refines the coarse-grained counter. Specifically, we prove the following refinement:

$$\mathsf{counter}_i \precsim_{ctx} \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

Using soundness (Theorem 4.1), this contextual refinement can be reduced to proving the refinement judgment $\mathsf{counter}_i \precsim \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int})$ in ReLoC.

The previous example demonstrated the basic usage of symbolic execution rules of ReLoC. Those symbolic execution rules were confined to the pure fragment of the programming language. In this example we show how to use ReLoC's symbolic execution rules for stateful computations and concurrency primitives. In addition to the type-directed structural rules and symbolic execution rules, the proof will require the usage of *invariants* for linking together the values of the two counters. We will use selected ReLoC rules from Figure 4.4. To symbolically execute the operations on locks that appear in $\mathsf{counter}_s$, we will also make use of the *relational specification* for locks in Figure 4.5. The lock specification is stated in terms of an abstract predicate $\mathsf{isLock_s}(lk, \mathbf{false})$ (resp., $\mathsf{isLock_s}(lk, \mathbf{true})$) stating that lk is a lock which is unlocked (resp., locked). The relational specification for locks can then be seen as consisting of symbolic execution rules that manipulate that abstract predicate.[2] We will see in Section 4.5.1.1 that these specifications can be proven for a simple spin lock.

4.3.4.1 Symbolic execution

Recall that performing symbolic execution means reducing the left-hand or right-hand side of the refinement according to the computational rules. We have already seen the usage of REL-PURE-L, which allows us to perform pure computations. For this example we also use stateful symbolic execution rules in Figure 4.4. To start with the

[2] Because this specification is for the "angelic" right-hand side, it does not express mutual exclusion as it is common for separation logic specifications. We explain this by contrasting the specification with the one for the left-hand side in Section 4.5.1.2.

refinement proof, we apply the stateful symbolic execution rule REL-ALLOC-L′ to the left-hand side to obtain:

$$c_i \mapsto_{\mathsf{i}} 0 \twoheadrightarrow ((\lambda().\ \mathsf{read}\ c_i), (\lambda().\ \mathsf{inc}_i\ c_i)) \precsim \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

Note that after the application of the rule we gain access to the resource $c_i \mapsto_{\mathsf{i}} 0$ representing the value of the counter on the left-hand side. Subsequently, using the symbolic execution rules REL-PURE-R, REL-ALLOC-R and NEWLOCK-R on the right-hand side the goal becomes:

$$c_i \mapsto_{\mathsf{i}} 0 * c_s \mapsto_{\mathsf{s}} 0 * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false}) \twoheadrightarrow$$
$$((\lambda().\ \mathsf{read}\ c_i), (\lambda().\ \mathsf{inc}_i\ c_i)) \precsim (\lambda().\ \mathsf{read}\ c_s), (\lambda().\ \mathsf{inc}_s\ c_s\ lk)) : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

In addition to gaining the resource $c_s \mapsto_{\mathsf{s}} 0$, representing the value of the right-hand side counter, we get access to the abstract predicate $\mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false})$, which keeps track of the state of the lock lk on the right-hand side.

ReLoC's symbolic execution rules are inspired by the "backwards"-style Hoare rules of [IO01] and the weakest-precondition rules in Iris [Kre+17; Jun+18b].

4.3.4.2 Invariants and persistent propositions

At this point we wish to prove a refinement of two closures. By the rule REL-PAIR it would suffice to prove that both closures refine each other. However, if we were to apply REL-PAIR, we would be forced to split our resources in two: the resources needed for the refinement proof of the read function, and the resources needed for the refinement proof of the increment function. But both of those operations require access to the counter locations $c_i \mapsto_{\mathsf{i}} -$ and $c_s \mapsto_{\mathsf{s}} -$. To circumvent this issue we put said resources in a global *invariant* $\boxed{P}$, which allows P to be shared between different parts of the program (and between different threads). In our running example, we establish the invariant $\boxed{I_{\mathsf{cnt}}}^{\mathcal{N}}$ (using REL-INV-ALLOC), where:

$$I_{\mathsf{cnt}} \triangleq \exists n \in \mathbb{N}.\ c_i \mapsto_{\mathsf{i}} n * c_s \mapsto_{\mathsf{s}} n * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false}).$$

The invariant $\boxed{I_{\mathsf{cnt}}}^{\mathcal{N}}$ not only allows us to share access to c_i and c_s, but also ensures that the values of the respective counters match up. For our invariant we pick any fresh invariant name $\mathcal{N} \in \mathit{InvName}$ (more on the invariant names below).

Invariants $\boxed{P}$ are *persistent*: once established, they will remain valid for the rest of the verification. This differentiates them from *ephemeral* propositions like $\ell \mapsto_{\mathsf{i}} v$ and $\ell \mapsto_{\mathsf{s}} v$, which could be invalidated in the future by actions of the program or proof.

The notion of being persistent is expressed in ReLoC (and Iris) by means of the *persistence* modality $\Box$. The purpose of $\Box P$ is to say that P holds without asserting any ephemeral propositions. The most important rules for the $\Box$ modality are $\Box P = \Box P * \Box P$ and $\Box P \twoheadrightarrow P$, which allow to freely duplicate $\Box P$ and finally get P out. We say that P is *persistent*, written as persistent(P), if $P \vdash \Box P$; otherwise, we say that P is *ephemeral*. To prove $\Box P$, one can only use persistent resources like $\boxed{P}$, and not ephemeral resources like $\ell \mapsto_{\mathsf{i}} v$. We refer to stripping off the persistence modality

in the context of persistent hypotheses as *introducing the* □ *modality*. We make that precise and give rules for the □ modality in Section 4.4.3.

Once the invariant $\boxed{I_{\mathsf{cnt}}}$ for our running example has been established, we can duplicate it, and apply REL-PAIR to obtain two goals:

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} (\lambda().\, \mathsf{read}\ c_i) \precsim (\lambda().\, \mathsf{read}\ c_s) : \mathsf{unit} \to \mathsf{int}$$
$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} (\lambda().\, \mathsf{inc}_i\ c_i) \precsim (\lambda().\, \mathsf{inc}_s\ c_s\ lk) : \mathsf{unit} \to \mathsf{int}.$$

We first describe how to prove the refinement of read. As $\lambda x.\, e$ is syntactic sugar for $\mathsf{rec}\ \_\ x = e$, we can apply REL-REC at the function type $\mathsf{unit} \to \mathsf{int}$ and obtain the new goal:

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} \Box\big(\forall v\, v'.\, [\![\mathsf{unit}]\!]_\Delta(v, v') \mathrel{-\!\!*} (\lambda().\, !c_i)\ v \precsim (\lambda().\, !c_s)\ v' : \mathsf{int}\big).$$

By VAL-UNIT, we obtain that $[\![\mathsf{unit}]\!]_\Delta(v, v')$ implies $v = v' = ()$. Moreover, since $\boxed{I_{\mathsf{cnt}}}$ is our only hypothesis, and it is persistent, we can strip off the □ modality, arriving at the following goal:

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} (\lambda().\, !c_i)\ () \precsim (\lambda().\, !c_s)\ () : \mathsf{int}.$$

Accessing invariants. The fact that invariants are persistent (and thus can be duplicated, *i.e.*, $\boxed{P} = \boxed{P} * \boxed{P}$) comes with a cost—once a proposition P has been turned into an invariant $\boxed{P}$, one is only allowed to access P during a single *atomic* execution step on the left-hand side. This restriction is crucial as the scheduling of threads on the left-hand side is demonic. When proving a refinement, we have to consider all possible interleavings of threads. If we were to be able to access an invariant for the duration of multiple steps, another thread could be scheduled in between, and observe that the invariant was temporarily broken.

Scheduling on the right-hand side, however, is angelic. That is, when proving a refinement, we have the ability to select the choice of scheduling. As a consequence, ReLoC allows us to execute multiple steps on the right-hand side while accessing an invariant.

Let us take a look at the way accessing invariants in ReLoC works. We do so by continuing the proof of our running example (after introducing □ and performing pure symbolic execution steps):

$$\boxed{I_{\mathsf{cnt}}} \mathrel{-\!\!*} !c_i \precsim !c_s : \mathsf{int}.$$

At this point we would like to access the locations c_i and c_s stored in the invariant $\boxed{I_{\mathsf{cnt}}}$. For this we use the rule REL-LOAD-L-INV in Figure 4.4.

This rule is quite a mouthful, so let us first take a look at its shape before going into detail about the mask annotations and later modalities ▷. The essence of REL-LOAD-L-INV is that it provides temporary access to the resources P guarded by the invariant. In addition, it provides the *invariant closing resource* $\mathsf{closeInv}_{\mathcal{N}}(P)$, which can restore the invariant (using the rule REL-INV-RESTORE). The resources P can be used to prove $\ell \mapsto_{\mathsf{i}} v$, which is needed to justify the symbolic execution step on the left. Afterwards, we are left with the goal $\Delta \models_{\top\setminus\mathcal{N}} K[\,v\,] \precsim e_2 : \tau$. We typically do not immediately restore the invariant (using REL-INV-RESTORE), but first use the resources P to perform matching symbolic execution steps on the right.

In our example, by applying REL-LOAD-L-INV, we obtain $c_i \mapsto_{\mathsf{i}} n$ and $c_s \mapsto_{\mathsf{s}} n$ and $\mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false})$, for some $n \in \mathbb{N}$, reducing our goal to $\models_{\top \setminus \mathcal{N}} n \precsim !\, c_s : \mathsf{int}$. We then use REL-LOAD-R to reduce our goal to $\models_{\top \setminus \mathcal{N}} n \precsim n : \mathsf{int}$. Because these steps did not change the heap, REL-INV-RESTORE's premises for closing the invariant are trivially met. The refinement proof is then concluded by applying the structural rules REL-RETURN and VAL-INT.

Let us take a look at the rules REL-LOAD-L-INV and REL-INV-RESTORE in more detail. A crucial aspect of these rules is that they ensure that access to the invariant $\boxed{P}^{\mathcal{N}}$ is *temporary*, *i.e.*, that P is only used during a single symbolic execution step on the left-hand side (but possibly several steps on the right), and that the same invariant cannot be opened twice. This is achieved by tagging each invariant $\boxed{P}^{\mathcal{N}}$ with a name $\mathcal{N} \in \mathit{InvName}$, and by keeping track of which invariants have been accessed. The latter is done in a way similar to Iris—like Iris's Hoare triples $\{P\}\, e\, \{Q\}_{\mathcal{E}}$, our refinement judgments $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ are annotated with a *mask* $\mathcal{E} \subseteq \mathit{InvName}$ of accessible invariants. By default all invariants are accessible, so we write $\Delta \models e_1 \precsim e_2 : \tau$ for $\Delta \models_{\top} e_1 \precsim e_2 : \tau$, where $\top$ is the set of all invariant names.

An invariant namespace is a (non-empty) list of strings or values: $\mathit{InvName} = \mathit{List}(\mathit{String} + \mathit{Val})$. When opening an invariant and removing it from a mask, we coerce an invariant namespace $\mathcal{N}$ into a mask by taking its upwards extension $\mathcal{N}^{\uparrow} = \{\mathcal{N}.x_1 \ldots .x_n \mid n \in \mathbb{N}, x_i \in \mathit{String} + \mathit{Val}\}$. Abusing the notation, we write $\mathcal{E} \setminus \mathcal{N}$ for $\mathcal{E} \setminus \mathcal{N}^{\uparrow}$.

When accessing an invariant, *e.g.*, using REL-LOAD-L-INV or REL-CAS-L-INV, its namespace is removed from the mask annotation of the judgment. The removal of the namespace from the mask guarantees that invariants are only used for a single execution step on the left-hand side. After all, all rules for symbolic execution on the left-hand side require a $\top$ mask, whereas those for the right-hand side allow for an arbitrary mask. The only way of performing a subsequent step on the left-hand side is thus by first restoring the mask to $\top$, which can only be done by restoring the invariants that have been accessed (using the rule REL-INV-RESTORE).

One may wonder why refinement judgments are annotated with a mask instead of a Boolean that indicates if an invariant has been opened. As we will show in Section 4.4, ReLoC allows one to access multiple invariants simultaneously. To avoid *reentrancy*—which means accessing the same invariant twice in a nested fashion—we need to know exactly which invariants are opened.

An additional aspect to note is that invariants $\boxed{P}^{\mathcal{N}}$ in ReLoC (and Iris) are *impredicative* [SB14; Jun+18b]. This means that P is allowed to contain other invariant assertions $\boxed{Q}^{\mathcal{N}'}$ or even refinement judgments $e \precsim t : \tau$. As a consequence, to ensure soundness of the logic, all rules for invariants only provide access to $\triangleright P$, *i.e.*, P "guarded" by the *later* modality $\triangleright$. When invariants are not used impredicatively (*i.e.*, invariants over so called *timeless* propositions, which include connectives of first-order logic and heap assertions), these modalities can be soundly omitted.

4.3.4.3 Later modality and Löb induction

The later modality $\triangleright$ is not only used for resolving the impredicativity issues, but also for handling general recursion. As is custom in logics based on step-indexing [AM01], such as Iris, the later modality $\triangleright$ and Löb induction are used to reason about recursive functions. Specifically, Iris provides the following rules for $\triangleright$:

$$\triangleright\text{-intro}\ \frac{P}{\triangleright P} \qquad \triangleright\text{-mono}\ \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \text{Löb}\ \frac{\triangleright P \vdash P}{\vdash P}$$

In our example, this means that by Löb induction (rule Löb), we may prove $\mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}$, under the assumption of the induction hypothesis $\triangleright(\mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int})$. The induction hypothesis is 'guarded' by a $\triangleright$, and can only be used after we have performed a step of symbolic execution on the left-hand side. That is why the symbolic execution rules for the left-hand side contain the later modality in the premises. Let us see how it works in the example. We use rel-pure-l to arrive at:

$$\triangleright(\mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}) \twoheadrightarrow$$
$$\triangleright(\mathsf{let}\ c = !\,c_i\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(c_i, c, 1+c)\ \mathsf{then}\ c\ \mathsf{else}\ \mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}).$$

By monotonicity (rule $\triangleright$-mono), we can now remove $\triangleright$ both from the induction hypothesis and from the goal. Subsequently, we symbolically execute the load operation using the invariant, just like in the previous section, reaching the goal

$$\mathsf{if}\ \mathsf{CAS}(c_i, n, 1+n)\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}$$

for some $n \in \mathbb{N}$. To symbolically execute the compare-and-set (CAS), we use rel-cas-l-inv. By this rule, we have to consider two outcomes, depending on whether the original value of the counter has changed between the load and compare-and-set operations or not.

1. Suppose that the value of the counter c_i has changed. In that case the compare-and-set operation fails and we are left with

   $$c_i \mapsto_{\mathsf{i}} m * c_s \mapsto_{\mathsf{s}} m * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false}) \twoheadrightarrow$$
   $$\models_{\top\setminus\mathcal{N}} \mathsf{if}\ \mathbf{false}\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}$$

   for some $m \neq n$. Because the symbolic heap has not been changed, we can easily restore the invariant and execute the $\mathsf{if}\ \mathbf{false}\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots$ to obtain $\mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}$, which is exactly our induction hypothesis.
2. If the value has not changed, then the compare-and-set succeeds and we are left with the new goal:

   $$c_i \mapsto_{\mathsf{i}} (1+n) * c_s \mapsto_{\mathsf{s}} n * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false}) \twoheadrightarrow$$
   $$\models_{\top\setminus\mathcal{N}} \mathsf{if}\ \mathbf{true}\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{inc}_i\ c_i \precsim \mathsf{inc}_s\ c_s\ lk : \mathsf{int}.$$

At this point we use the symbolic execution rules REL-STORE-R, REL-LOAD-R and the lock specifications from Figure 4.5 to symbolically execute the right-hand side of the refinement and update the resources to match:

$$c_i \mapsto_{\mathsf{i}} (1+n) * c_s \mapsto_{\mathsf{s}} (1+n) * \mathsf{isLock_s}(lk, \mathbf{false}) \twoheadrightarrow$$
$$\models_{\top \setminus \mathcal{N}} \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ n\ \mathbf{else}\ \mathsf{inc}_i\, c_i \precsim n : \mathsf{int}.$$

We can then restore the invariant and symbolically execute the left-hand side to finish the proof.

Note that the point in the proof when we symbolically execute $\mathsf{inc}_s\ c_s\ lk$ on the right-hand side corresponds to the linearization point of inc_i.

This concludes the proof of the counter refinement. For the purposes of the proof, we have used some derived rules and principles in ReLoC. In the next section we will present an overview of primitive rules—the very core of ReLoC—and show how they can be used to recover the kind of intuitive reasoning we employed in this section.

4.4 A closer look at ReLoC

We now explain some of the more technical details of ReLoC, and show how the principles that we have used in Section 4.3 can be obtained from ReLoC's primitive proof rules. First, we describe how to work with invariants using Iris's update modality $\Rightarrow$ (Section 4.4.1). Then we explain the role and rules of persistent propositions (Section 4.4.2), and go through a selection of ReLoC's primitive proof rules and explain how the symbolic execution and structural rules can be derived from them (Section 4.4.3). Finally, we demonstrate how ReLoC's rules can be used to prove the *fundamental property*: if we can derive a typing judgment $\vdash e : \tau$, then e refines itself, *i.e.*, $e \precsim e : \tau$. To prove the fundamental property, we need to generalize the relational judgment to open terms, and prove the structural rules for open terms as well (Section 4.4.4).

A selection of ReLoC's primitive proof rules are shown in Figure 4.6.

4.4.1 Invariants and the update modality

The rules for invariants in Figure 4.4 in Section 4.3.4.2 are fairly restrictive, *e.g.*, they allow us to open at most one invariant at the same time. Moreover, several of those rules, *e.g.*, REL-LOAD-L-INV and REL-CAS-L-INV, mix together symbolic execution and invariant manipulation. We now present ReLoC's more primitive proof rules, which integrate Iris's flexible mechanism for invariants and ghost state, and which can be used to derive rules such as like REL-LOAD-L-INV and REL-CAS-L-INV.

Invariants and ghost state in Iris are controlled via the *update modality* ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$. The intuition behind ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$ is to express that under the assumption that the invariants in $\mathcal{E}_1$ are accessible initially, one can obtain P, and end up in the situation where the invariants in $\mathcal{E}_2$ are accessible. Thus, for showing P we can open the invariants from $\mathcal{E}_1$ and have to restore the invariants from $\mathcal{E}_2$ (the invariants from $\mathcal{E}_1 \setminus \mathcal{E}_2$ may remain open). Furthermore, this modality allows one to perform changes to Iris's

Value interpretation rules:

$$\textsc{val-var}\ \frac{\Delta(\alpha)(v_1, v_2)}{[\![\alpha]\!]_\Delta(v_1, v_2)} \qquad \textsc{val-unit}\ \frac{v_1 = v_2 = ()}{[\![\mathsf{unit}]\!]_\Delta(v_1, v_2)} \qquad \textsc{val-bool}\ \frac{\exists b \in \mathbb{B}.\ v_1 = v_2 = b}{[\![\mathsf{bool}]\!]_\Delta(v_1, v_2)} \qquad \textsc{val-int}\ \frac{\exists n \in \mathbb{Z}.\ v_1 = v_2 = n}{[\![\mathsf{int}]\!]_\Delta(v_1, v_2)}$$

$$\textsc{val-prod}\ \frac{\exists v_1, v_2, w_1, w_2.\ (v = (v_1, v_2)) * (w = (w_1, w_2)) * [\![\tau]\!]_\Delta(v_1, w_1) * [\![\sigma]\!]_\Delta(v_2, w_2)}{[\![\tau \times \sigma]\!]_\Delta(v, w)}$$

$$\textsc{val-arr}\ \frac{\Box(\forall w_1\, w_2.\ [\![\tau]\!]_\Delta(w_1, w_2) \mathrel{-\!\!*} \Delta \models v_1\ w_1 \precsim v_2\ w_2 : \sigma)}{[\![\tau \to \sigma]\!]_\Delta(v_1, v_2)}$$

Monadic rules:

$$\textsc{rel-return}\ \frac{[\![\tau]\!]_\Delta(v_1, v_2)}{\Delta \models v_1 \precsim v_2 : \tau} \qquad \textsc{rel-bind}\ \frac{\Delta \models e_1 \precsim e_2 : \tau \qquad \forall v_1\, v_2.\ [\![\tau]\!]_\Delta(v_1, v_2) \mathrel{-\!\!*} \Delta \models K_1[\,v_1\,] \precsim K_2[\,v_2\,] : \sigma}{\Delta \models K_1[\,e_1\,] \precsim K_2[\,e_2\,] : \sigma}$$

Type-directed structural rules:

$$\textsc{rel-fork}\ \frac{\Delta \models e_1 \precsim e_2 : \mathsf{unit}}{\Delta \models \mathsf{fork}\ \{e_1\} \precsim \mathsf{fork}\ \{e_2\} : \mathsf{unit}}$$

Symbolic execution rules:

$$\textsc{rel-load-l}\ \frac{{}^{\top}\!\Rightarrow^{\mathcal{E}} \left(\exists v.\ \ell \mapsto_{\mathsf{i}} v * \triangleright \left(\ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,v\,] \precsim e_2 : \tau\right)\right)}{\Delta \models K[\,!\ell\,] \precsim e_2 : \tau}$$

$$\textsc{rel-store-l}\ \frac{{}^{\top}\!\Rightarrow^{\mathcal{E}} \left(\ell \mapsto_{\mathsf{i}} - \mathrel{-\!\!*} \triangleright \left(\ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,()\,] \precsim e_2 : \tau\right)\right)}{\Delta \models K[\,\ell \leftarrow v\,] \precsim e_2 : \tau}$$

$$\textsc{rel-cas-l}\ \frac{{}^{\top}\!\Rightarrow^{\mathcal{E}} \left(\begin{array}{r}\exists v'.\ \ell \mapsto_{\mathsf{i}} v' * \triangleright \left(v' \neq v_1 \mathrel{-\!\!*} \triangleright (\ell \mapsto_{\mathsf{i}} v' \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,\mathbf{false}\,] \precsim e_2 : \tau)\right) \wedge \\ \triangleright \left(v' = v_1 \mathrel{-\!\!*} \triangleright (\ell \mapsto_{\mathsf{i}} v_2 \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,\mathbf{true}\,] \precsim e_2 : \tau)\right)\end{array}\right)}{\Delta \models K[\,\mathsf{CAS}(\ell, v_1, v_2)\,] \precsim e_2 : \tau}$$

Figure 4.6: Selected primitive rules of ReLoC.

Invariants rules (INV-ALLOC and INV-ACCESS are inherited from Iris):

$$\textsc{rel-upd}\ \frac{{}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} \Delta \vDash_{\mathcal{E}_2} e_1 \precsim e_2 : \tau}{\Delta \vDash_{\mathcal{E}_1} e_2 \precsim e_2 : \tau} \qquad \textsc{inv-alloc}\ \frac{\triangleright P}{\Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}} \qquad \textsc{inv-access}\ \frac{\mathcal{N}^{\uparrow} \subseteq \mathcal{E} \qquad \boxed{P}^{\mathcal{N}}}{{}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\triangleright P \twoheadrightarrow {}^{\mathcal{E}\setminus\mathcal{N}}\Rrightarrow^{\mathcal{E}} \mathsf{True})}$$

Figure 4.6: Selected primitive rules of ReLoC (cont.)

ghost state via *frame preserving updates*; for a description of those we refer the reader to [Jun+18b].

The key rules of the update modality are:

$$\Rrightarrow\text{-INTRO}\ \frac{P}{{}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}} P} \qquad \Rrightarrow\text{-MONO}\ \frac{P \vdash Q}{{}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} Q} \qquad \Rrightarrow\text{-IDEMP}\ \frac{{}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\Rrightarrow^{\mathcal{E}_3} P}{{}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_3} P} \qquad \Rrightarrow\text{-SEP}\ \frac{P * {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} Q}{{}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} (P * Q)}$$

These rules say that the update modality is a monad, which is indexed (due to the masks), and strong (due to rule ⇛-SEP). In ReLoC (and Iris) proofs, we often need to eliminate update modalities in the proof context, which is allowed if the goal is an update modality with corresponding source mask. This is expressed by the following derived rule:

$$\Rrightarrow\text{-ELIM}\ \frac{{}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} P \qquad P \twoheadrightarrow {}^{\mathcal{E}_2}\Rrightarrow^{\mathcal{E}_3} Q}{{}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_3} Q}$$

This rule is derivable from ⇛-MONO, and ⇛-IDEMP.

Before we will describe the rules of the update modality related to invariants, let us describe some syntactic sugar that we inherit from Iris. We write $\Rrightarrow_{\mathcal{E}} P$ for ${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}} P$, and $\Rrightarrow P$ for $\Rrightarrow_{\top} P$, where $\top$ is the set of all invariant names. Moreover, since the update modality is often combined with the magic wand, we write $P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q$ for $P \twoheadrightarrow {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} Q$, and follow the same conventions for omitting masks on $\Rrightarrow$ as used for $\Rrightarrow$.

ReLoC's main rule for interacting with the update modality is REL-UPD. It allows to eliminate an update modality around a refinement judgment. To get an idea of how this rule is used, let us take a look at the primitive rule INV-ALLOC for allocating an invariant. The derived rule REL-INV-ALLOC in Figure 4.4 is a composition of REL-UPD with Iris's rules ⇛-ELIM and INV-ALLOC.

By combining REL-UPD with Iris's rules ⇛-ELIM and INV-ACCESS for accessing invariants, one can turn an invariant $\boxed{P}^{\mathcal{N}}$ into its content P, together with a way of restoring the invariant $\triangleright P \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}$. It is important to notice that by using the combination of these rules, the mask on the refinement judgment changes from $\mathcal{E}$ into $\mathcal{E} \setminus \mathcal{N}$. This prohibits access to the invariant $\mathcal{N}$ until it has been restored—thus preventing reentrancy. Restoring the invariant is done by using the rule REL-UPD with

the premise $\triangleright P \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow\!{}^{\mathcal{E}}$ True. This requires one to give up P, and in turn transforms the mask of the judgment back into $\mathcal{E}$. Note that one can use INV-ACCESS multiple times to open multiple invariants.

Invariants and symbolic execution. Opening invariants through REL-UPD and INV-ACCESS as described above is fairly limited. Once we open an invariant, the mask at the refinement judgment changes from $\top$ into $\top \setminus \mathcal{N}$, which prevents any symbolic execution on the left-hand side. The rules for symbolic execution on that side require the mask to be $\top$. As we discussed in Section 4.3.4.2 already, this restriction to the $\top$ mask on left-hand side rules is crucial. It is unsound to perform multiple symbolic execution steps on the left while an invariant is open. To see why this is the case, consider the following refinement:

$$(\lambda x.\, \mathsf{let}\; n = !\,x \;\mathsf{in}\; x \leftarrow n+1; n) \precsim \mathsf{inc}_i : \mathsf{ref\ int} \to \mathsf{int}$$

This refinement does not hold because the two programs can be distinguished by the context:

$$\mathsf{let}\; c = \mathsf{ref}(0)\; \mathsf{in\ let}\; f = [\bullet]\; \mathsf{in\ fork}\; \{f\ c\}; f\,c.$$

The left-hand side is basically the coarse-grained increment operation inc_s without the lock protection. Thus, the function on the left-hand side does not guarantee thread-safety: the value of the passed reference can change unpredictably if the function is invoked in parallel with itself. By contrast, the inc_i always increments the counter monotonically.

If we were allowed to perform multiple symbolic execution rules on the left-hand side, then we could have proven the above refinement, using an invariant of $\boxed{\exists n.\, c_s \mapsto_{\mathsf{s}} n * c_i \mapsto_{\mathsf{i}} n}$.

In order to support symbolic execution with invariants, ReLoC provides additional rules to simultaneously access an invariant and perform a single atomic symbolic execution step on the left-hand side. Examples of such rules are REL-LOAD-L, REL-STORE-L and REL-CAS-L.

We can now explain the derived rule REL-LOAD-L-INV in terms of the primitive rules. The proposition $\triangleright P \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow\!{}^{\mathcal{E}}$ True is used for closing the invariant $\mathcal{N}$ because it changes the mask from $\mathcal{E} \setminus \mathcal{N}$ to $\mathcal{E}$. Thus $\mathsf{closeInv}_{\mathcal{N}}(P) \triangleq (\triangleright P \;{}^{\top\setminus\mathcal{N}}\!\Rrightarrow\!{}^{\top}$ True). To prove REL-LOAD-L-INV from Figure 4.4, we apply REL-LOAD-L to obtain the goal:

$${}^{\top}\!\Mapsto^{\top\setminus\mathcal{N}}\Big(\exists v.\, \ell \mapsto_{\mathsf{i}} v * \triangleright\Big(\ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \vDash_{\top\setminus\mathcal{N}} K[\,v\,] \precsim e : \tau\Big)\Big).$$

We then use INV-ACCESS and $\Mapsto$-ELIM to get the premise of REL-LOAD-L-INV. In the same way REL-CAS-L-INV can be derived from REL-CAS-L. Finally, the closing rule REL-INV-RESTORE is a consequence of the definition of $\mathsf{closeInv}_{\mathcal{N}}(P)$ and REL-UPD.

Using ReLoC's primitive symbolic execution rules such as REL-LOAD-L, REL-STORE-L and REL-CAS-L one can also derive the following weaker, but perhaps more intuitive, symbolic execution rule:

$$\frac{\ell \mapsto_{\mathsf{i}} v \qquad \triangleright\Big(\ell \mapsto_{\mathsf{i}} w \mathrel{-\!\!*} \Delta \vDash K[\,()\,] \precsim e_2 : \tau\Big)}{\Delta \vDash K[\,\ell \leftarrow w\,] \precsim e_2 : \tau}\;\textsc{rel-store-l}'$$

Since these rules have a $\top$ mask, they can only be used when no invariants have been opened. Recall that by contrast, the symbolic execution rules for the right-hand side, such as REL-LOAD-R, REL-STORE-R in Figure 4.4, which are of a similar shape, can be performed even with invariants open because they allow the mask to be arbitrary.

4.4.2 The persistence modality

Recall from Section 4.3.4.2 that a proposition P is persistent, written as $\mathsf{persistent}(P)$, if $P \vdash \Box P$, where $\Box$ is Iris's *persistence* modality. The $\Box$ modality plays an important role in ReLoC because it makes it possible to express that if two expressions are related, they remain related forever. For example, the persistence modality plays a crucial role in the rule REL-REC in Figure 4.4—it ensures that ephemeral resources (such as heap assertions) are not used for the verification of the closure's body. After all, closures can be invoked arbitrarily many times at different points in time (possibly concurrently), and hence it is impossible to guarantee that ephemeral resources will still be available when the closure is called. For example, without the $\Box$ modality in the premise of REL-REC one would be able to prove the following unsound refinement:

$$\mathsf{let}\ \ell = \mathsf{ref}(0)\ \mathsf{in}\ \lambda().\, \ell \leftarrow 1 + !\ell; !\ell \precsim \lambda().\, 1 \ :\ \mathsf{unit} \to \mathsf{int}.$$

One would use REL-ALLOC-L' to obtain the heap assertion $\ell \mapsto_{\mathsf{i}} 0$, and subsequently use that assertion to verify the body of the closure. Fortunately, the $\Box$ modality in REL-REC prevails—$\ell \mapsto_{\mathsf{i}} 0$ is ephemeral, not persistent, so cannot be moved under a $\Box$.

In Section 4.3.4.2 we gave an idea of the core rules of the persistence modality. Let us now take a look at the rules in more detail:

$$\frac{\Box P * \Box P}{\Box P}\ \Box\text{-DUP} \qquad \frac{\Box P}{P}\ \Box\text{-ELIM} \qquad \frac{P \vdash Q}{\Box P \vdash \Box Q}\ \Box\text{-MONO} \qquad \frac{\Box P}{\Box\Box P}\ \Box\text{-IDEMP} \qquad \frac{\Box P * \Box Q}{\Box(P * Q)}\ \Box\text{-SEP}$$

The rules $\Box$-DUP and $\Box$-ELIM say that the $\Box P$ is duplicable, and one can get P out. The rule $\Box$-IDEMP says that $\Box P$ itself is persistent. The rules $\Box$-ELIM, $\Box$-MONO and $\Box$-IDEMP say that $\Box$ is in fact a co-monad. Finally, $\Box$ commutes with most logical connectives, for example, the separating conjunction, as expressed by $\Box$-SEP.

If we wish to prove $\Box Q$ under the assumptions $P_1, \ldots, P_n$, where each P_i is persistent, then we can introduce the $\Box$ modality and prove Q from $P_1, \ldots, P_n$:

$$\frac{\mathsf{persistent}(P_1) \quad \ldots \quad \mathsf{persistent}(P_n) \quad P_1 * \cdots * P_n \vdash Q}{P_1 * \cdots * P_n \vdash \Box Q}\ \Box\text{-INTRO}$$

This rule is derivable from the definition of $\mathsf{persistent}(-)$, $\Box$-SEP, and $\Box$-MONO.

Note that $\mathsf{persistent}(P)$ is defined through the validity relation $P \vdash \Box P$; *i.e.*, it is a meta-logical notion (in terms of the mechanization, $\mathsf{persistent}(P)$ is a Coq-level predicate, not an Iris-level predicate). As such, the rule above does not fit the description we have given to the inference rules in Section 4.3.1. Rather, it should be seen as a family of inference rules indexed by meta-level propositions $\mathsf{persistent}(P_1), \ldots, \mathsf{persistent}(P_n)$.

4.4.3 Value interpretation and monadic rules

In addition to the refinement judgment $\Delta \models e_1 \precsim e_2 : \tau$, which relates expressions e_1 and e_2, ReLoC provides the value interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$, which relates values v_1 and v_2. The rule REL-RETURN expresses that $[\![\tau]\!]_\Delta(v_1, v_2)$ implies $\Delta \models v_2 \precsim v_2 : \tau$. However, the inverse direction does not hold, $[\![\tau]\!]_\Delta(v_1, v_2)$ is strictly stronger than $\Delta \models v_1 \precsim v_2 : \tau$ as its rules (in Figure 4.6) are bidirectional, whereas those for the expression judgment are unidirectional. The bidirectionality is crucial for the rule REL-REC in Figure 4.4, as it contains $[\![\tau]\!]_\Delta(v_1, v_2)$ in negative position—*i.e.*, as a client of REL-REC one gets $[\![\tau]\!]_\Delta(v_1, v_2)$ as an assumption and hence needs to eliminate it.

We want the value interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$ to be persistent, because our type system is not substructural, *i.e.*, types denote knowledge, but not ownership of data. For example, in typing the expression $e_1 \leftarrow e_2$ with STORE-TYPED, we use the same context Γ for type checking both e_1 and e_2. In order to semantically validate such rules, we want the propositions $[\![\tau]\!]_\Delta(v_1, v_2)$ to be duplicable. To that end, we require all the interpretations in the context Δ to be persistent. That is why the rule REL-PACK in Figure 4.4 has a side-condition $\forall v_1, v_2.\, \mathsf{persistent}(R(v_1, v_2))$.

The value interpretation also appears in the monadic rules REL-RETURN and REL-BIND in Figure 4.6. These rules are used to derive all type-directed structural rules of ReLoC, with the exception of REL-FORK, which is the sole primitive type-directed structural rule. As an example, consider the type-directed structural rule for the first projection π_1.

Lemma 4.2. The following rule is derivable:

$$\frac{\Delta \models e_1 \precsim e_2 : \tau \times \sigma}{\Delta \models \pi_1(e_1) \precsim \pi_1(e_2) : \tau}$$

Proof. By REL-BIND it suffices to show:

- $\Delta \models e_1 \precsim e_2 : \tau \times \sigma$, but this is exactly our assumption;
- for any v, w: $[\![\tau \times \sigma]\!]_\Delta(v, w) \twoheadrightarrow \Delta \models \pi_1(v) \precsim \pi_1(w) : \tau$.

By VAL-PROD we have values v_i, w_i for $i \in \{1, 2\}$ such that $v = (v_1, v_2)$ and $w = (w_1, w_2)$ and $[\![\tau]\!]_\Delta(v_1, w_1) * [\![\sigma]\!]_\Delta(v_2, w_2)$. Using REL-PURE-L and REL-PURE-R we reduce the goal $\Delta \models \pi_1(v_1, v_2) \precsim \pi_1(w_1, w_2) : \tau$ to $\Delta \models v_1 \precsim w_1 : \tau$. At this point we apply REL-RETURN. □

4.4.4 Fundamental theorem and refinements of open terms

The type-directed structural rules[3] are also used for proving the following theorem, which is a standard result for logical relation models of type systems:

Theorem 4.3 (Fundamental theorem for closed terms)**.** If expression e is well typed, *i.e.*, $\vdash e : \tau$, then e refines itself, *i.e.*, the judgment $e \precsim e : \tau$ is derivable in ReLoC.

We wish to prove this theorem by induction on the typing derivation. But in order to make it work, we need to generalize the theorem to open terms (*e.g.*, in

[3] Our *type-directed structural rules* are often called *compatibility lemmas* in the logical relation literature.

order to deal with the REC-TYPED case). Consequently, we need to generalize ReLoC's refinement judgment $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ to open terms e_1 and e_2 whose free variables are bound by the typing context Γ. To define the refinement judgment for open terms, we first define a standard notion of a *closing substitution*.

Definition 4.4. A mapping $\gamma : Var \to Val \times Val$ is a *closing substitution w.r.t. the typing environment* Γ, notation $[\![\Gamma]\!]^*_\Delta(\gamma)$, if

$$\forall (x, \tau) \in \Gamma.\ [\![\tau]\!]_\Delta(\gamma_1(x), \gamma_2(x)),$$

where $\gamma_i(x) = \pi_i(\gamma(x))$ is the i-th projection of $\gamma(x)$.

Definition 4.5. The *refinement judgment* $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ *for open terms* is defined as:

$$\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau \ \triangleq\ \forall \gamma.\ [\![\Gamma]\!]^*_\Delta(\gamma) \twoheadrightarrow \Delta \models \gamma_1(e_1) \precsim \gamma_2(e_2) : \tau.$$

Using the refinement judgment for open terms we can now state versions of the type-directed structural rules for open terms. For example:

$$\frac{\Delta \mid x{:}\tau, \Gamma \models e_1 \precsim e_2 : \sigma}{\Delta \mid \Gamma \models \lambda x.\, e_1 \precsim \lambda x.\, e_2 : \tau \to \sigma}$$

This rule can be proven by unfolding the definition of the refinement judgment for open terms and proceeding as in Lemma 4.2. Finally, we can state and prove the generalized version of the fundamental theorem for open terms:

Theorem 4.6 (Fundamental theorem for open terms)**.** If $\Xi \mid \Gamma \vdash e : \tau$, then $\Delta \mid \Gamma \models e \precsim e : \tau$ is derivable in ReLoC, for all Δ which contain the variables from Ξ.

Proof. By induction on the typing derivation, using the versions of the type-directed structural rules for open terms. □

With the refinement judgments generalized to open terms, we can state and the generalized version of Theorem 4.1 for open terms, which we prove in Section 4.7.5.

Theorem 4.7 (Soundness for open terms)**.** Let Ξ be a type environment. Suppose that refinement judgment $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ is derivable in ReLoC for all Δ which contain the variables from Ξ. Then $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$.

4.5 Relational specifications in ReLoC

Due to its first-class refinement judgments, ReLoC can be used to give *relational specifications* to programs. Similar to Hoare triples, relational specifications abstract away from a program's implementation by expressing its behavior in terms of a pre- and postcondition. Relational specifications apply to the situation when the expression on the one side of the refinement contains a program subject to specification, while the expression on the other side is arbitrary. In Figure 4.5 in Section 4.3 we saw an example of a right-hand side relational specifications for locks, which we then used to verify a counter module.

We start this section by describing the general format of non-atomic relational specifications (Section 4.5.1). Non-atomic relational specifications are sufficient to give strong specifications for the right-hand left, but due to the demonic nature of the left-hand side, we often need stronger specifications for the left-hand side (Section 4.5.2). We therefore introduce *logically atomic relation specifications*, which generalize da Rocha Pinto *et al.*'s TaDA-style specifications [RDG14] (Section 4.5.3) and Svendsen *et al.*'s HOCAP-style specifications [SBP13] (Section 4.5.4) from the Hoare-logic setting to the relational setting. Finally, we show how to use logically atomic specifications to verify a ticket lock (Section 4.5.5).

4.5.1 Non-atomic relational specifications

4.5.1.1 Right-hand side relational specifications

Consider the following implementation of a lock, which we refer to as a *spin lock*:

$$\begin{aligned}
&\mathsf{newlock} \triangleq \lambda().\ \mathsf{ref}(\mathbf{false})\\
&\mathsf{acquire} \triangleq \lambda\ell.\ \mathsf{if}\ \mathsf{CAS}(\ell, \mathbf{false}, \mathbf{true})\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{acquire}\ \ell\\
&\mathsf{release} \triangleq \lambda\ell.\ \ell \leftarrow \mathbf{false}
\end{aligned}$$

For this specific implementation, we can prove the rules in Figure 4.5 in Section 4.3.4 by defining the lock predicates as follows:

$$\mathsf{isLock}_{\mathsf{s}}(lk, b) \triangleq lk \in Loc * lk \mapsto_{\mathsf{s}} b.$$

The rules for locks in Figure 4.5 follow a certain pattern. For an expression e_2 that, under precondition P, reduces to v, with postcondition $Q(x, v)$, we have the following rule:

$$\frac{P \qquad \forall x, v.\ Q(x, v) \twoheadrightarrow \Delta \models_{\mathcal{E}} e_1 \precsim K[\,v\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,e_2\,] : \tau}$$

The postcondition $Q : X \times Val \to iProp$ also depends on a type X, provided by the provider of the rule. This rule pattern can be considered a relational version of a Hoare triple for a program on the right-hand side of the refinement judgment.

4.5.1.2 Left-hand side relational specifications

We can formulate a similar pattern for programs on the left-hand side of the refinement judgment:

$$\frac{P \qquad \forall x, v.\ Q(x, v) \twoheadrightarrow \Delta \models K[\,v\,] \precsim e_2 : \tau}{\Delta \models K[\,e_1\,] \precsim e_2 : \tau}$$

Using this pattern we can state and prove a relational version of the standard separation logic specification for locks, which is shown in Figure 4.7. This specification makes use of the lock predicate $\mathsf{isLock}_{\mathsf{i}}(\gamma, lk, R)$, which states that lk protects the resources described by the proposition R. When creating a lock using newlock, the resources R have to be given up, and the persistent lock predicate $\mathsf{isLock}_{\mathsf{i}}(\gamma, lk, R)$ is

$$\textsc{newlock-l}\ \frac{R \qquad \forall lk,\, \gamma.\ \mathsf{isLock_i}(\gamma, lk, R) \twoheadrightarrow K[\,lk\,] \precsim e_2 : \tau}{K[\,\mathsf{newlock}\ ()\,] \precsim e_2 : \tau} \qquad \textsc{is-lock-pers}\ \frac{\mathsf{isLock_i}(\gamma, lk, R)}{\Box\,\mathsf{isLock_i}(\gamma, lk, R)}$$

$$\textsc{acquire-l}\ \frac{\mathsf{isLock_i}(\gamma, lk, R) \qquad (\mathsf{locked}(\gamma) \twoheadrightarrow R \twoheadrightarrow K[\,()\,] \precsim e_2 : \tau)}{K[\,\mathsf{acquire}\ lk\,] \precsim e_2 : \tau}$$

$$\textsc{release-l}\ \frac{\mathsf{isLock_i}(\gamma, lk, R) \qquad \mathsf{locked}(\gamma) \qquad R \qquad K[\,()\,] \precsim e_2 : \tau}{K[\,\mathsf{release}\ lk\,] \precsim e_2 : \tau}$$

Figure 4.7: Left-hand side relational specification for locks.

given in return. A thread that acquires a lock by calling acquire gets access to R for the duration of the critical section, and has to give R back when calling release. The token $\mathsf{locked}(\gamma)$, where γ is a *ghost name* associated to the lock, makes sure that a lock can only be released when it has been acquired. To prove the left-hand side specification for the spin lock, we define the lock predicate $\mathsf{isLock_i}(\gamma, lk, R)$ following the usual definition in Iris:

$$\mathsf{isLock_i}(\gamma, lk, R) \triangleq lk \in Loc * \boxed{(lk \mapsto_{\mathsf{i}} \mathbf{false} * \mathsf{locked}(\gamma) * R) \vee (lk \mapsto_{\mathsf{i}} \mathbf{true})}^{\mathcal{N}_{\mathsf{lock}}}$$

This definition uses an Iris invariant to express that the lock is either unlocked ($lk \mapsto_{\mathsf{i}} \mathbf{false}$), in which case it holds the token $\mathsf{locked}(\gamma)$ and the resources R, or locked ($lk \mapsto_{\mathsf{i}} \mathbf{true}$), in which case it holds no resources, since those are held by the thread that acquired the lock. The token $\mathsf{locked}(\gamma)$ is an exclusive resource that is obtained from Iris's ghost theory.

4.5.1.3 Left- versus right-hand side relational specifications

As we have seen in the specifications for the lock, there is an asymmetry between the left- and right-hand side specifications. The left-hand side specification of acquire (ACQUIRE-L) can be used regardless of whether the lock is unlocked, whereas the right-hand specification (ACQUIRE-R) can be used solely if the lock is unlocked (*i.e.*, if $\mathsf{isLock_s}(lk, \mathbf{false})$). This is due to the demonic nature of the left-hand side and the angelic nature of the right-hand side. For acquire on the left-hand side, we have to consider an arbitrary execution, whereas for acquire on the right-hand side we have to provide an execution ourselves. That is, for acquire on the right-hand side we have to show that it actually acquires the lock and reduces to $()$, which is only possible when the lock is unlocked. For this reason we use the predicate $\mathsf{isLock_s}(lk, b)$, which tracks the state b of the lock.

4.5.2 The need for logically atomic specifications

Recall from Section 4.4.1 that for any primitive (stateful) operation we have a symbolic execution rule that allows the client to access shared resourced stored in an invariant. For example, the rule REL-STORE-L for the store operation is as follows:

$$\frac{{}^{\top}\!\Rightarrow^{\mathcal{E}}\left(\ell \mapsto_{\mathsf{i}} - \mathrel{-\!\!*\!\!\triangleright} \left(\ell \mapsto_{\mathsf{i}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,()\,] \precsim e_2 : \tau\right)\right)}{\Delta \models K[\,\ell \leftarrow v\,] \precsim e_2 : \tau}$$

Concretely, the update modality ${}^{\top}\!\Rightarrow^{\mathcal{E}}$ in the premise of this rule allows users to use INV-ACCESS to access an invariant for the duration of the operation. The mask $\mathcal{E}$ in the refinement judgment $\Delta \models_{\mathcal{E}} K[\,()\,] \precsim e_2 : \tau$ (that appears in the premise of the rule) forces the user to close the invariant at the end of the duration of the operation. The ability to open an invariant is sound because operations such as store are *physically atomic*—*i.e.*, they reduce in one step. As a consequence of being physically atomic, other threads cannot observe that the invariant has been broken during the execution of the operation.

In contrast, methods of a concurrent program module are typically composed of several operations and hence they are not physically atomic. For example, consider the increment function inc_i of the fine-grained counter module from Figure 4.1:

$$\mathsf{inc}_i \triangleq \mathsf{rec}\ \mathit{inc}\ c = \mathsf{let}\ n = !c\ \mathsf{in}\\ \qquad\qquad \mathsf{if}\ \mathsf{CAS}(c, n, 1+n)\ \mathsf{then}\ n\ \mathsf{else}\ \mathit{inc}\ c$$

This function is a compound expression that does not reduce to a value in a single step. Nevertheless, during the execution of this function there is a single instant at which the whole operation actually appears to take the effect—namely the successful reduction of the compare-and-set operation (CAS). This instant is called the *linearization point*. What it means is that, for an outside observer, the method inc_i behaves *as if* it was atomic, and we wish to express that in this function's relational specification.

This phenomenon is called *logical atomicity* in the literature, and has been studied extensively in the context of Hoare-style logics [JP11; RDG14; SBP13; Jun+15; Jun+20]. In the upcoming subsections we will how to generalize the concept of logical atomicity to the relational setting, and how that gives rise to *logically atomic relational specifications*. Concretely, we generalize da Rocha Pinto *et al.*'s TaDA-style specifications [RDG14] (Section 4.5.3) and Svendsen *et al.*'s HOCAP-style specifications [SBP13] (Section 4.5.4) from the Hoare-logic setting to the relational setting. Establishing the formal comparison between the two styles is out of the scope of this paper. Rather, we demonstrate that both approaches can be applied to the context of relational specifications.

4.5.3 TaDA-style relational specifications

4.5.3.1 Formulating TaDA-style specifications

We take inspiration from the encoding of TaDA-style logically atomic Hoare triples in Iris [Jun+15] and assign the following logically atomic relational specification to inc_i:

$$\textsc{inc-i-l-tada}\quad \frac{\square\, {}^{\top}\!\Rrightarrow^{\mathcal{E}} \exists n.\, c \mapsto_{\mathsf{i}} n * \left(\begin{array}{l}\left(c \mapsto_{\mathsf{i}} n \;{}^{\mathcal{E}}\!\Rrightarrow\!\!\!\!\ast^{\top}\, \mathsf{True}\right) \wedge \\ \left(c \mapsto_{\mathsf{i}} (n+1) \mathrel{-\!\!*} \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau\right)\end{array}\right)}{K[\,\mathsf{inc}_i\; c\,] \precsim e : \tau}$$

Contrary to the non-atomic specification, we do not have $c \mapsto_{\mathsf{i}} n$ as a premise of the rule directly, but instead the premise contains a way of obtaining $c \mapsto_{\mathsf{i}} n$. The typical way of obtaining $c \mapsto_{\mathsf{i}} n$ is by accessing an invariant, which is formally done by using the update modality ${}^{\top}\!\Rrightarrow^{\mathcal{E}}$ in the premise combined with INV-ACCESS from Figure 4.6.

To justify the remaining part of the premise of the rule we need to take a closer took at the behavior of $\mathsf{inc}_i\; c$, whose implementation (Figure 4.1) we recall to be as follows:

$$\mathsf{inc}_i \triangleq \mathsf{rec}\; \mathit{inc}\; c = \mathsf{let}\; n = !c\; \mathsf{in}\\ \qquad\qquad \mathsf{if}\; \mathsf{CAS}(c, n, 1+n)\; \mathsf{then}\; n\; \mathsf{else}\; \mathit{inc}\; c$$

The compare-and-set operation (CAS) can either succeed or fail. If it succeeds, then we have managed to update our resources to $c \mapsto_{\mathsf{i}} (n+1)$, and we can proceed with proving $\models_{\mathcal{E}} K[\,n\,] \precsim e : \tau$ under that premise. This explains the $(c \mapsto_{\mathsf{i}} (n+1) \mathrel{-\!\!*} \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau)$ clause. If, however, the compare-and-set fails, then we need to be able to restart the whole computation of $\mathsf{inc}_i\; c$. For that we must be able to return $c \mapsto_{\mathsf{i}} n$ to the invariant. Hence the $(c \mapsto_{\mathsf{i}} n \;{}^{\mathcal{E}}\!\Rrightarrow\!\!\!\!\ast^{\top}\, \mathsf{True})$ clause. (The same clause is used for performing operations that do not modify the state, such as dereferencing.)

Finally, we know that the computation can either succeed or be restarted—but not both. We have to accommodate for both situations, just not at the same time. Hence the last two clauses described here are connected by an intuitionistic conjunction ($\wedge$), instead of the separating conjunction ($*$).

4.5.3.2 Using TaDA-style specifications

We use the logically atomic relational specification INC-I-L-TADA to prove the refinement that we have seen in Section 4.3.4.2. The new proof is more modular since it does not appeal to the definition of inc_i. The refinement that we want to prove is as follows:

$$\boxed{I_{\mathsf{cnt}}}^{\mathcal{N}} \mathrel{-\!\!*} \mathsf{inc}_i\; c_i \precsim \mathsf{inc}_s\; c_s\; l : \mathsf{int}$$

Recall that $I_{\mathsf{cnt}} \triangleq \exists n \in \mathbb{N}.\, c_i \mapsto_{\mathsf{i}} n * c_s \mapsto_{\mathsf{s}} n * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false})$. To prove this goal, we use INC-I-L-TADA. After introducing the persistence modality (using $\square$-INTRO, which is allowed, because there are no ephemeral assumptions in our context), this gives the

following new goal (under the assumption $\boxed{I_{\mathsf{cnt}}}^{\mathcal{N}}$):

$$ {}^{\top}\!\Rightarrow^{\top\setminus\mathcal{N}} \exists n.\, c_i \mapsto_{\mathsf{i}} n * \begin{pmatrix} \left(c_i \mapsto_{\mathsf{i}} n \;{}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True}\right) \wedge \\ \left(c_i \mapsto_{\mathsf{i}} (n+1) \mathrel{-\!\!*} \models_{\top\setminus\mathcal{N}} n \precsim \mathsf{inc}_s\ c_s\ l : \mathsf{int}\right) \end{pmatrix} $$

At this point we can open up the invariant I_{cnt} (using INV-ACCESS), and thereby introduce the update modality. The contents of the invariant provides us with a witness for the existential quantifier and allows us to discharge $c \mapsto_{\mathsf{i}} n$. We are left with proving the conjunction:

$$ \left(c_i \mapsto_{\mathsf{i}} n \;{}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True}\right) \quad \wedge \quad \left(c_i \mapsto_{\mathsf{i}} (n+1) \mathrel{-\!\!*} \models_{\top\setminus\mathcal{N}} n \precsim \mathsf{inc}_s\ c_s\ l : \mathsf{int}\right) $$

under the assumption of the unused resources $\mathsf{isLock}_{\mathsf{s}}(l, \mathbf{false})$ and $c_s \mapsto_{\mathsf{s}} n$ from the invariant, and the invariant closing resource $\triangleright I_{\mathsf{cnt}} \;{}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True}$.

The first conjunct corresponds to the case in which we close the invariant without modifying anything in our current context (*i.e.*, the compare-and-set has failed). It follows directly from the invariant closing resource. It thus remains to prove the second conjunct (*i.e.*, the compare-and-set has succeeded), which means we should prove $\models_{\top\setminus\mathcal{N}} n \precsim \mathsf{inc}_s\ c_s\ l : \mathsf{int}$ under the assumptions $c_i \mapsto_{\mathsf{i}} (n+1)$ and $\mathsf{isLock}_{\mathsf{s}}(l, \mathbf{false})$ and $c_s \mapsto_{\mathsf{s}} n$ and $\triangleright I_{\mathsf{cnt}} \;{}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True}$. At this point we finish the proof by symbolically executing $\mathsf{inc}_s\ c_s\ l$ on the right-hand side before closing the invariant using invariant closing resource.

4.5.3.3 General format of TaDA-style specifications

The general format of logically atomic rules for logical refinements is the following:

$$ \frac{R \qquad \Box\, {}^{\top}\!\Rightarrow^{\mathcal{E}} \exists x.\, P(x) * \begin{pmatrix} \left(P(x) \;{}^{\mathcal{E}}\!\Rrightarrow^{\top} \mathsf{True}\right) \wedge \\ \left(\forall v.\, Q(x,v) * R \mathrel{-\!\!*} \models_{\mathcal{E}} K[\,v\,] \precsim e_2 : \tau\right) \end{pmatrix}}{K[\,e_1\,] \precsim e_2 : \tau} $$

Here, $P : X \to \mathit{iProp}$ is a predicate describing consumed resources, and $Q : X \times \mathit{Val} \to \mathit{iProp}$ is a predicate describing produced resources, both dependent on a type X supplied by the provider of the rule (*e.g.*, a library that exports the program e_1). This parameter X is selected on per-specification basis. For example, for the counter module X is going to be the type of natural numbers.

We include a frame R, which can be chosen by the client, for the following reason. The second premise of the rule resides below a persistence modality. Whenever we prove a goal of the form $\Box P$ we must prove P using only persistent resources, and thus have to throw all the ephemeral resources away (see $\Box$-INTRO in Section 4.4.2). However, we do not want to throw away all the ephemeral resources that we have for eternity (as they might be needed to close invariants afterwards or to proceed otherwise with the proof), so we give them up only temporarily, by collecting them in R.

4.5.3.4 Proving TaDA-style specifications

Following the general scheme, we now state and prove the TaDA-style specification of our increment function:

$$\frac{R \quad {}^{\top}\!\Rrightarrow^{\mathcal{E}} \square\exists n.\, c \mapsto_{\mathsf{i}} n * \left(\begin{array}{l}\left(c \mapsto_{\mathsf{i}} n \;{}^{\mathcal{E}}\!\Rrightarrow^{\top} \mathsf{True}\right) \wedge \\ \left(c \mapsto_{\mathsf{i}} (n+1) * R \mathrel{-\!\!*} \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau\right)\end{array}\right)}{K[\,\mathsf{inc}_i\; c\,] \precsim e : \tau}\;\textsc{inc-i-l-tada-gen}$$

To prove this specification, we proceed by Löb induction and symbolic execution. At the point when we need to symbolically dereference c we apply REL-LOAD-L. We then use the update that we have as a premise of the specification to obtain $c \mapsto_{\mathsf{i}} n$ for some n. After providing $c \mapsto_{\mathsf{i}} n$ for the load operation, we use the first conjunct $c \mapsto_{\mathsf{i}} n * {}^{\mathcal{E}}\!\Rrightarrow^{\top} \mathsf{True}$ to restore the mask on the refinement judgment.

After that we have to symbolically execute the compare-and-set operation; we apply REL-CAS-L and use the update that we have as a premise again to obtain $c \mapsto_{\mathsf{i}} m$ for some m, as needed for REL-CAS-L. If $m \neq n$, then the compare-and-set operation has failed, and we can restart the proof first by restoring the mask on the refinement judgment (using the closing update), and then using the Löb induction hypothesis. If $m = n$, then the compare-and-set operation has succeeded, and the points-to connective is updated to $c \mapsto_{\mathsf{i}} (n+1)$. Then we can use the second conjunct $c \mapsto_{\mathsf{i}} (n+1) * R \mathrel{-\!\!*} \models_{\mathcal{E}} K[\,n\,] \precsim e : \tau$ to arrive at the exact conclusion that we need: $\models_{\mathcal{E}} K[\,n\,] \precsim e : \tau$.

4.5.4 HOCAP-style relational specifications

We now present another form of logically atomic relational specifications—*HOCAP-style logical atomic relational specifications*, which are based on the logically atomic specifications by Svendsen *et al.* [SBP13] in the eponymous logic. Contrary to TaDA-style specifications, which come in a precisely specified format (Section 4.5.3.3), HOCAP-style specifications do not have a precise format. This provides the flexibility that not only can they be used to represent linearization points, but they can also be used to represent arbitrary observable interactions with the abstract state. This flexibility allows us to give strong specifications to non-linearizable methods (Section 4.5.4.4), which we use in the ticket lock refinement proof (Section 4.5.5).

4.5.4.1 Formulating HOCAP-style specifications

Let us consider the HOCAP-style specification in Figure 4.8 for the fine-grained concurrent counter from Figure 4.1 in Section 4.1. Contrary to the TaDA-style specification, the HOCAP-style specification does not expose the underlying state of the counter (*i.e.*, $\ell \mapsto_{\mathsf{i}} n$) directly, but instead provides an abstract view of the state through abstract predicates.

The persistent predicate $\mathsf{cnt\_ctx}_\gamma(c, \mathcal{N})$ asserts that the value c represents a counter. The specification is parameterized by a namespace $\mathcal{N}$ for the internal invariants associated with the specification. The ghost name γ is used to link c with the predicates $\mathsf{cnt}_\gamma(q, n)$ and $\mathsf{cnt\_auth}_\gamma(n)$, which describe the abstract state of the

Rules for abstract predicates:

$$\textsc{cnt-agree}\quad \frac{\mathsf{cnt\_auth}_\gamma(n) \qquad \mathsf{cnt}_\gamma(q,m)}{n=m} \qquad\qquad \textsc{cnt-agree'}\quad \frac{\mathsf{cnt}_\gamma(q_1,n) \qquad \mathsf{cnt}_\gamma(q_2,m)}{n=m}$$

$$\textsc{cnt-combine}\quad \frac{\mathsf{cnt}_\gamma(q_1,n) * \mathsf{cnt}_\gamma(q_2,n)}{\mathsf{cnt}_\gamma(q_1+q_2,n)} \qquad \textsc{cnt-update}\quad \frac{\mathsf{cnt\_auth}_\gamma(n) \qquad \mathsf{cnt}_\gamma(1,m)}{\Rightarrow \mathsf{cnt\_auth}_\gamma(k) * \mathsf{cnt}_\gamma(1,k)} \qquad \textsc{Cnt-persistent}\quad \frac{\mathsf{cnt\_ctx}_\gamma(c,\mathcal{N})}{\Box\,\mathsf{cnt\_ctx}_\gamma(c,\mathcal{N})}$$

Relational specification:

$$\textsc{new-i-l-hocap}\quad \frac{\forall c\,\gamma.\ \mathsf{cnt\_ctx}_\gamma(c,\mathcal{N}) * \mathsf{cnt}_\gamma(1,n) \mathrel{-\!\!*} K[\,c\,] \precsim e_2 : \tau}{K[\,\mathsf{ref}(n)\,] \precsim e_2 : \tau}$$

$$\textsc{inc-i-l-hocap}\quad \frac{\mathcal{E} \cap \mathcal{N}^{\uparrow} = \emptyset \qquad \mathsf{cnt\_ctx}_\gamma(c,\mathcal{N}) \qquad (\forall n.\ \mathsf{cnt\_auth}_\gamma(n) \overset{\top\setminus\mathcal{N}}{\Rightarrow\!\!\!*} \overset{\top\setminus\mathcal{N}\setminus\mathcal{E}}{} \mathsf{cnt\_auth}_\gamma(n+1) * \models_{\top\setminus\mathcal{E}} K[\,n\,] \precsim e_2 : \tau)}{K[\,\mathsf{inc}_i\ c\,] \precsim e_2 : \tau}$$

$$\textsc{read-i-l-hocap}\quad \frac{\mathcal{E} \cap \mathcal{N}^{\uparrow} = \emptyset \qquad \mathsf{cnt\_ctx}_\gamma(c,\mathcal{N}) \qquad (\forall n.\ \mathsf{cnt\_auth}_\gamma(n) \overset{\top\setminus\mathcal{N}}{\Rightarrow\!\!\!*} \overset{\top\setminus\mathcal{N}\setminus\mathcal{E}}{} \mathsf{cnt\_auth}_\gamma(n) * \models_{\top\setminus\mathcal{E}} K[\,n\,] \precsim e_2 : \tau)}{K[\,\mathsf{read}\ c\,] \precsim e_2 : \tau}$$

Figure 4.8: HOCAP-style logically atomic relational specification for a fine-grained concurrent counter.

counter. The predicate $\mathsf{cnt}_\gamma(q,n)$ provides the *client view* of the abstract state of the counter. It is similar to the fractional heap points-to connective from separation logic—it associates a value (a natural number n) to the counter, and can be split and combined according to the fractional component q (CNT-AGREE', CNT-COMBINE). The predicate $\mathsf{cnt\_auth}_\gamma(m)$ provides the *module view* of the abstract state of the counter. It agrees with the client view (CNT-AGREE) and can be used together with $\mathsf{cnt}_\gamma(1,n)$ to update the value associated to the counter (CNT-UPDATE).

Ownership of the module view predicate $\mathsf{cnt\_auth}_\gamma(m)$ is given to the user only during the execution of the counter operations. Consider, for example, the specification INC-I-L-HOCAP for the atomic increment function inc_i. From the client's point of view, there is only one place where inc_i observably interacts with the abstract state of the counter, namely during its linearization point. For this point of access, the user

has to provide the update:

$$\mathsf{cnt\_auth}_\gamma(n) \ {}^{\top\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\!\!\ast^{\top\setminus\mathcal{N}\setminus\mathcal{E}} \ \mathsf{cnt\_auth}_\gamma(n+1) * \models_{\top\setminus\mathcal{E}} K[\,n\,] \precsim e_2 : \tau.$$

The user is given the module view $\mathsf{cnt\_auth}_\gamma(n)$ of the counter, and has to update it to $\mathsf{cnt\_auth}_\gamma(n+1)$. For that, the user has to appeal to CNT-UPDATE and has to provide $\mathsf{cnt}_\gamma(q,n)$ themselves (either as an immediate resource or from an invariant in $\mathcal{E}$, which can be opened thanks to the update modality). After the abstract state is updated, the user has to prove the refinement judgment $\models_{\top\setminus\mathcal{E}} K[\,n\,] \precsim e_2 : \tau$. Similar to the TaDA-style specifications, the mask on the refinement is set to $\top\setminus\mathcal{E}$, allowing the user to perform some reasoning on the right-hand side before closing all the invariants from $\mathcal{E}$.

4.5.4.2 Using HOCAP-style specifications

We use the HOCAP-style logically atomic relational specification from Figure 4.8 to prove the refinement that we have seen in Section 4.3.4.2:

$$\mathsf{counter}_i \precsim \mathsf{counter}_s : (\mathsf{unit} \to \mathsf{int}) \times (\mathsf{unit} \to \mathsf{int}).$$

Since the HOCAP-style specifications are stated in terms of abstract predicates, instead of the $\ell \mapsto_{\mathsf{i}} n$ connective, we need a slightly different invariant than the one we used for the proof using the TaDA-style specification in Section 4.5.3.2, namely:

$$\mathsf{cnt\_ctx}_\gamma(c_i, \mathcal{N}.\mathsf{cnt}) * \boxed{\exists n \in \mathbb{N}.\ \mathsf{cnt}_\gamma(1,n) * c_s \mapsto_{\mathsf{s}} n * \mathsf{isLock}_{\mathsf{s}}(lk, \mathbf{false})}^{\mathcal{N}.\mathsf{inv}}.$$

After having established this invariant, the refinement proofs for the increment and read methods proceed similar to the corresponding TaDA-style proofs (Section 4.5.3.2), except that in the increment case the user has to use CNT-UPDATE alongside INC-I-L-HOCAP in order to update the ghost state $\mathsf{cnt}_\gamma(1,n)$ accordingly. We do not give the proof here, and direct the reader to the accompanying Coq mechanization.

In Section 4.5.5 we will another example of a client using the HOCAP-style specification for the counter.

4.5.4.3 Proving HOCAP-style specifications

We discuss how to prove that the implementation of the fine-grained counter meets the HOCAP-style specifications. To do so, we first use Iris's ghost theory to define the predicates $\mathsf{cnt}_\gamma(q,n)$ and $\mathsf{cnt\_auth}_\gamma(n)$ (the details of this definition are omitted). We then define the predicate $\mathsf{cnt\_ctx}_\gamma(c,\mathcal{N})$, which provides the internal invariant of the module:

$$\mathsf{cnt\_ctx}_\gamma(c,\mathcal{N}) \triangleq c \in Loc * \boxed{\exists n \in \mathbb{N}.\ c \mapsto_{\mathsf{i}} n * \mathsf{cnt\_auth}_\gamma(n)}^{\mathcal{N}}$$

This invariant states that the physical value n of c corresponds to the logical value n of the predicate $\mathsf{cnt\_auth}_\gamma(n)$. To see how this invariant is used, let us consider the proof of INC-I-L-HOCAP for the inc_i operation. Since inc_i is defined recursively, we prove

this rule by Löb induction. We proceed by symbolically executing the left-hand side, accessing the invariant $\mathcal{N}$ to dereference c for some value n. It then remains to show:

$$\text{if CAS}(c, n, 1+n) \text{ then } n \text{ else } \mathsf{inc}_i\ c \precsim e_2 : \tau.$$

At this point we use the atomic symbolic execution rule for compare-and-set REL-CAS-L (with $\mathcal{E} = \mathcal{N}$). We introduce the update modality ${}^{\top}\!\Rightarrow^{\top\setminus\mathcal{N}}$ we obtain by accessing the invariant $\mathcal{N}$ using Iris's strong invariant access rule INV-ACCESS-STRONG, which is needed because we need to access invariants in a non-well-bracketed way:

INV-ACCESS-STRONG

$$\frac{\mathcal{N}^{\uparrow} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}}\ {}^{\mathcal{E}}\!\Rightarrow\!\!\ast^{\mathcal{E}\setminus\mathcal{N}} \triangleright P * (\forall \mathcal{E}'.\ \triangleright P\ {}^{\mathcal{E}'}\!\Rightarrow\!\!\ast^{\mathcal{E}'\cup\mathcal{N}} \mathsf{True})}$$

It remains to consider two cases. If the compare-and-set (CAS) has failed, we close the invariant (by setting $\mathcal{E}' = \top \setminus \mathcal{N}$) and appeal to the induction hypothesis. If the compare-and-set has succeeded, we are left to show the following:

$$\cdots * c \mapsto_{\mathsf{i}} (n+1) * \mathsf{cnt\_auth}_\gamma(n) \twoheadrightarrow \models_{\top\setminus\mathcal{N}} n \precsim e_2 : \tau.$$

We first use the update ${}^{\top\setminus\mathcal{N}}\!\Rightarrow\!\!\ast^{\top\setminus\mathcal{N}\setminus\mathcal{E}}$ that is provided by the premise of the rule to update $\mathsf{cnt\_auth}_\gamma(n)$ into $\mathsf{cnt\_auth}_\gamma(n+1)$. This moreover provides a proof of $\models_{\top\setminus\mathcal{E}} n \precsim e_2 : \tau$. At this point our goal becomes $\models_{\top\setminus\mathcal{N}\setminus\mathcal{E}} n \precsim e_2 : \tau$. We close the invariant $\mathcal{N}$ (by setting $\mathcal{E}' = \top \setminus \mathcal{N} \setminus \mathcal{E}$) and restore the mask on the refinement proposition in our goal, resulting in $\models_{\top\setminus\mathcal{E}} n \precsim e_2 : \tau$, which is exactly what we obtained from the update ${}^{\top\setminus\mathcal{N}}\!\Rightarrow\!\!\ast^{\top\setminus\mathcal{N}\setminus\mathcal{E}}$.

4.5.4.4 HOCAP-style specifications for non-linearizable operations

Using HOCAP-style specifications we can also specify operations that are not linearizable. Consider the following "weak increment" function that we can add to the counter module:

$$\mathsf{wkincr} \triangleq \lambda c.\ c \leftarrow (!c + 1)$$

The function increments the value in the location c non-atomically. What kind of specification can we give to wkincr? To answer this we have to examine what the update $\Rightarrow\!\!\ast$ represents in the HOCAP-style specifications. In the previous examples with linearizable functions, the updates represented observations about the abstract state that the clients could make, and they corresponded to the linearization points. But there is no reason why we should pin them to linearization points only. Rather, we can let the update correspond to any operation that is observable through the abstract state. In wkincr there are two points where such operations happen, which we can represent through two nested updates:

CNT-WK-INCR-L

$$\frac{\mathcal{E} \cap \mathcal{N}^{\uparrow} = \emptyset \qquad \mathsf{cnt\_ctx}_\gamma(c, \mathcal{N})}{\forall n.\ \mathsf{cnt\_auth}_\gamma(n) \Rightarrow\!\!\ast_{\top\setminus\mathcal{N}} \mathsf{cnt\_auth}_\gamma(n) * (\forall m.\ \mathsf{cnt\_auth}_\gamma(m)\ {}^{\top\setminus\mathcal{N}}\!\Rightarrow\!\!\ast^{\top\setminus\mathcal{N}\setminus\mathcal{E}} \mathsf{cnt\_auth}_\gamma(n+1) * \models_{\top\setminus\mathcal{E}} K[\,()\,] \precsim e_2 : \tau)}{K[\,\mathsf{wkincr}\ c\,] \precsim e_2 : \tau}$$

$$\begin{aligned}
\mathsf{newlock}_{\mathsf{TL}} &\triangleq \lambda().\ (\mathsf{ref}(0), \mathsf{ref}(0)) \\
\mathsf{acquire}_{\mathsf{TL}} &\triangleq \lambda(l_{\mathsf{o}}, l_{\mathsf{n}}).\ \mathsf{let}\ n = \mathsf{inc}_i\ l_{\mathsf{n}}\ \mathsf{in}\ \mathsf{wait_loop}\ n\ l_{\mathsf{o}} \\
\mathsf{wait_loop} &\triangleq \lambda n\ l_{\mathsf{o}}.\ \mathsf{if}\ (n = \mathsf{read}\ l_{\mathsf{o}})\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{wait_loop}\ n\ l_{\mathsf{o}} \\
\mathsf{release}_{\mathsf{TL}} &\triangleq \lambda(l_{\mathsf{o}}, l_{\mathsf{n}}).\ \mathsf{wkincr}\ l_{\mathsf{o}}
\end{aligned}$$

Figure 4.9: Ticket lock implementation.

The first update binds the value n, which is obtained from the initial read operation $!c$. In the conclusion of this update the client needs to return the $\mathsf{cnt\_auth}_\gamma(n)$ predicate, as in the specification READ-I-L-HOCAP. In addition to that predicate, the client has to provide the second update, in which $\mathsf{cnt\_auth}_\gamma(m)$ has to be updated to $\mathsf{cnt\_auth}_\gamma(n+1)$, corresponding to the assignment $c \leftarrow n+1$. The value m corresponds to the intermediate state of the counter, which might have changed in between the dereferencing of c and the assignment to it. The presence of two updates differentiates methods that have a linearization point, such as inc_i, and non-linearizable methods, such as wkincr.

In the next section we will see how a client might use the specification CNT-WK-INCR-L.

4.5.5 Ticket lock refinement from HOCAP-style specs

We show how our HOCAP-style relational specifications for the fine-grained concurrent counter (Figure 4.8) is used to prove that a *ticket lock* refines a *spin lock* (or, rather, that a ticket lock refines any lock satisfying the specification in Figure 4.5). The proof in this section demonstrates several important features of ReLoC. First, it demonstrates compositionality of proofs in ReLoC both by employing relational specifications for the left- and right-hand sides, and by reducing the refinement proof of a program module into separate reusable refinement proofs of the module functions. Second, the proof highlights the integration of Iris ghost state to facilitate CAP-style [Din+10] reasoning with abstract predicates.

A ticket lock [MS91, Section 2.2] is a ticket-based data structure for mutual exclusion, which is fair—threads racing to enter a critical section will gain access to it in the order of arrival at the critical section. The implementation of the ticket lock is given in Figure 4.9. The two locations associated with the lock, l_{o} and l_{n}, point to the identifiers of the current owner of the lock, and to the total number of issued tickets, respectively. When a thread wants to enter a critical section using the $\mathsf{acquire}_{\mathsf{TL}}$ function, it first requests a new ticket (by atomically increasing the value of l_{n} using the inc_i function), and then spins until the value of the current owner of the lock matches the ticket number (using wait_loop).

The function $\mathsf{release}_{\mathsf{TL}}$ uses the weak increment wkincr (Section 4.5.4.4) on the location l_{o}. It does not need to use an atomic increment (*i.e.*, inc_i), because, if the lock

$$\textsc{newIssuedTickets}\quad \Rrightarrow \exists \gamma.\, \mathsf{issuedTickets}_\gamma(0)$$

$$\textsc{issueNewTicket}\quad \frac{\mathsf{issuedTickets}_\gamma(m)}{\Rrightarrow \mathsf{issuedTickets}_\gamma(m+1) * \mathsf{ticket}_\gamma(m)}$$

$$\textsc{ticket-nondup}\quad \frac{\mathsf{ticket}_\gamma(n) \qquad \mathsf{ticket}_\gamma(n)}{\mathsf{False}}$$

Figure 4.10: The ticket ghost theory.

is used in a well-bracketed manner, only the owner of the lock will be calling the $\mathsf{release}_{\mathsf{TL}}$ function.

Concretely, the refinement that we wish to show is the following:

$$\mathsf{pack}(\mathsf{newlock}_{\mathsf{TL}}, \mathsf{acquire}_{\mathsf{TL}}, \mathsf{release}_{\mathsf{TL}}) \precsim$$
$$\mathsf{pack}(\mathsf{newlock}, \mathsf{acquire}, \mathsf{release}) : \exists \alpha.\, (\mathsf{unit} \to \alpha) \times (\alpha \to \mathsf{unit}) \times (\alpha \to \mathsf{unit}).$$

Here, newlock, acquire and release are any operations that satisfy the relational specification from Figure 4.5 (for example, the spin lock from Section 4.5.1.1).

Proof outline. To prove the refinement above we employ our general strategy for proving refinements for stateful program modules in ReLoC:

1. We define an invariant locklnv linking together the underlying representations of each individual pair of locks, which we use to define a witness for the existential type α.
2. We prove the refinements for each method in the signature.
3. Finally, we combine those proofs together into a module refinement proof. This is what we refer to as a *component-wise* refinement proof.

We stress that to carry out the proof we neither need to refer to the implementation of the fine-grained concurrent counter (on the left-hand side), nor to the implementation of the spin lock (on the right-hand side). Rather, we only refer to the HOCAP-style specification for the fine-grained counter and the relational specification for the spin lock.

Proof of the refinement. To match up the physical representation of tickets in the lock we use Iris's ghost theory to define abstract predicates tracking the tickets. We will use two ghost predicates: $\mathsf{issuedTickets}_\gamma(m)$ saying that m tickets have been issued in total, and $\mathsf{ticket}_\gamma(n)$ representing the n-th ticket. The predicates satisfy the rules in Figure 4.10.

To prove the refinement of lock modules, we need to pick a relation (serving as the interpretation for the witness α of the existential type) that links the two modules

together. We use the the relation locklnt defined as follows:

$$\mathsf{locklnv}_\gamma(\gamma_o,\gamma_n,lk) \triangleq \exists (o\, n : \mathbb{N})\,(b : \mathbb{B}).\ \mathsf{cnt}_{\gamma_o}(1,o) * \mathsf{cnt}_{\gamma_n}(1,n) * \mathsf{isLock}_s(lk,b) * \mathsf{issuedTickets}_\gamma(n) * (\mathsf{ticket}_\gamma(o) \vee b = \mathbf{false})$$

$$\mathsf{locklnt}((l_o,l_n),lk) \triangleq \exists \gamma_o,\gamma_n,\gamma.\ \mathsf{cnt\_ctx}_{\gamma_o}(l_o,\mathcal{N}.\mathsf{o}) * \mathsf{cnt\_ctx}_{\gamma_n}(l_n,\mathcal{N}.\mathsf{n}) * \boxed{\mathsf{locklnv}_\gamma(\gamma_o,\gamma_n,lk)}^{\mathcal{N}.\mathsf{inv}}$$

Here, (l_o,l_n) is the ticket lock on the left-hand side, and lk is the specification lock on the right-hand side. The locklnt relation states that l_o and l_n are concurrent counters with ghost names γ_o and γ_n, that satisfy the invariant locklnv. This invariant describes the relation between the values representing two locks. It states that the values of the counters l_o and l_n are o and n, respectively, and that exactly n tickets have been issued. Furthermore, the right-hand side lock lk is locked iff the ticket $\mathsf{ticket}_\gamma(o)$ of the current owner of the lock is in the invariant; that is, $\mathsf{ticket}_\gamma(o)$ was given up by a thread that acquired the lock.

Using REL-PACK we subdivide the main refinement proof into three refinements for the functions that constitute the lock module:

1. $[\alpha := \mathsf{locklnt}] \models \mathsf{newlock}_{\mathsf{TL}} \precsim \mathsf{newlock} : \mathsf{unit} \to \alpha$;
2. $[\alpha := \mathsf{locklnt}] \models \mathsf{acquire}_{\mathsf{TL}} \precsim \mathsf{acquire} : \alpha \to \mathsf{unit}$;
3. $[\alpha := \mathsf{locklnt}] \models \mathsf{release}_{\mathsf{TL}} \precsim \mathsf{release} : \alpha \to \mathsf{unit}$.

Proposition 4.8. $[\alpha := \mathsf{locklnt}] \models \mathsf{newlock}_{\mathsf{TL}} \precsim \mathsf{newlock} : \mathsf{unit} \to \alpha$.

Proof. By rule REL-REC it suffices to show $[\alpha := \mathsf{locklnt}] \models \mathsf{newlock}_{\mathsf{TL}}\ () \precsim \mathsf{newlock}\ () : \alpha$. By applying the symbolic execution rules and NEW-I-L-HOCAP we are left with the goal:

$$\mathsf{cnt\_ctx}_{\gamma_n}(l_n,\mathcal{N}.\mathsf{n}) * \mathsf{cnt\_ctx}_{\gamma_o}(l_o,\mathcal{N}.\mathsf{o}) * \mathsf{cnt}_{\gamma_o}(1,0) * \mathsf{cnt}_{\gamma_n}(1,0) * \mathsf{isLock}_s(lk,\mathbf{false}) \twoheadrightarrow [\alpha := \mathsf{locklnt}] \models (l_o,l_n) \precsim lk : \alpha.$$

From the premises we can allocate the invariant $\mathsf{locklnv}_\gamma(\gamma_o,\gamma_n,lk)$, and obtain $\mathsf{locklnt}((l_o,l_n),lk)$. We finish the proof with REL-RETURN. □

To prove the $\mathsf{acquire}_{\mathsf{TL}}$ refinement we need the following helper lemma.

Lemma 4.9. $\mathsf{ticket}_\gamma(m) \vdash [\alpha := \mathsf{locklnt}] \models \mathsf{wait\_loop}\ m\ l_o \precsim \mathsf{acquire}\ lk : \mathsf{unit}$, provided we have $\boxed{\mathsf{locklnv}_\gamma(\gamma_o,\gamma_n,lk)}^{\mathcal{N}.\mathsf{inv}}$.

Proof. We prove the refinement by Löb induction and symbolic execution. After some pure symbolic executions steps we are left with the goal:

$$[\alpha := \mathsf{locklnt}] \models \mathsf{if}\ (m = \mathsf{read}\ l_o)\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{wait\_loop}\ m\ l_o \precsim \mathsf{acquire}\ lk : \mathsf{unit}.$$

We then apply READ-I-L-HOCAP, after which it remains to prove

$$\mathsf{cnt\_auth}_{\gamma_o}(o) \overset{\mathcal{E}'}{\Rightarrow\!\!\!\!\ast}{}^{\mathcal{E}'\setminus\mathcal{N}} \mathsf{cnt\_auth}_{\gamma_o}(o) * [\alpha := \mathsf{locklnt}] \models \mathsf{if}\ (m = o)\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{wait\_loop}\ m\ l_o \precsim \mathsf{acquire}\ lk : \mathsf{unit}.$$

for any number o. In case $m \neq o$, we symbolically execute the left-hand side and appeal to the induction hypothesis. In case $m = o$, we proceed by accessing the invariant $\boxed{\mathsf{lockInv}_\gamma(\gamma_o, \gamma_n, lk)}^{\mathcal{N}.\mathsf{inv}}$. Note that it cannot be the case that $b = \mathbf{true}$, because then we would have two copies of $\mathsf{ticket}_\gamma(o)$: one from the assumption of the lemma and one from the invariant. Thus, the case $b = \mathbf{true}$ can be eliminated by TICKET-NONDUP. Then it must be the case that $b = \mathbf{false}$. In that case we have $\mathsf{isLock}_s(lk, \mathbf{false})$ and we can apply ACQUIRE-R to update it to $\mathsf{isLock}_s(lk, \mathbf{true})$, changing the right-hand side to ().

We finish by closing the invariant, picking this time $b = \mathbf{true}$ and storing the $\mathsf{ticket}_\gamma(o)$ from the assumption of the lemma in the invariant. □

Proposition 4.10. $[\alpha := \mathsf{lockInt}] \models \mathsf{acquire}_{\mathsf{TL}} \precsim \mathsf{acquire} : \alpha \to \mathsf{unit}$.

Proof. We use REL-REC and symbolic execution rules, and then INC-I-L-HOCAP and Lemma 4.9. When we apply INC-I-L-HOCAP, we use the update to issue a new ticket using ISSUENEWTICKET. This ticket will be used for the assumption of Lemma 4.9. □

Proposition 4.11. $[\alpha := \mathsf{lockInt}] \models \mathsf{release}_{\mathsf{TL}} \precsim \mathsf{release} : \alpha \to \mathsf{unit}$.

Proof. We use REL-REC, and then symbolic execution and CNT-WK-INCR-L, after which the new goal becomes

$$\mathsf{cnt\_auth}_{\gamma_o}(n) \Rrightarrow\!\!\!\!\ast_{\mathcal{E}'} \mathsf{cnt\_auth}_{\gamma_o}(n) * (\forall m.\, \mathsf{cnt\_auth}_{\gamma_o}(m) \overset{\mathcal{E}'}{\Rrightarrow}\!\!\!\!\ast^{\mathcal{E}'\setminus\mathcal{N}} \mathsf{cnt\_auth}_{\gamma_o}(n+1) * \models_{\top\setminus\mathcal{N}} () \precsim \mathsf{release}\ lk : \tau)$$

for an arbitrary n. By framing $\mathsf{cnt\_auth}_{\gamma_o}(n)$, it suffices to show

$$\mathsf{cnt\_auth}_{\gamma_o}(m) \overset{\mathcal{E}'}{\Rrightarrow}\!\!\!\!\ast^{\mathcal{E}'\setminus\mathcal{N}} \mathsf{cnt\_auth}_{\gamma_o}(n+1) * \models_{\top\setminus\mathcal{N}} () \precsim \mathsf{release}\ lk : \tau$$

for an arbitrary m. We utilize this update by accessing the invariant and getting access to $\mathsf{cnt}_{\gamma_o}(1, o)$. Using this proposition and $\mathsf{cnt\_auth}_{\gamma_o}(m)$ we apply CNT-UPDATE to get

$$\mathsf{cnt}_{\gamma_o}(1, n+1) * \mathsf{cnt\_auth}_{\gamma_o}(n+1).$$

We frame the second separating conjunct, and use RELEASE-R to reduce the right-hand side to (). Finally we close the invariant and finish the proof with REL-RETURN. □

4.6 Speculative reasoning using prophecy variables

In addition to Iris's ordinary ghost state mechanism, which allows to reason about the history of a program, Iris has recently been extended with a mechanism for speculative reasoning based on *prophecy variables*, which allows to reason about the future of a program [AL91; Jun+20]. A prophecy variable is a ghost variable that can reference a value that is determined in the future of a program's execution. While the program that is subject to verification cannot make use of the value of a prophecy variable itself—they are ghost variables—the value can be used in proofs, *e.g.*, to speculatively choose a reduction step in the right-hand program. In this section we

show how Iris's mechanism for prophecy variables is integrated into ReLoC and can be put to action to prove challenging refinements.

We start this section by illustrating the need for prophecy variables with a motivational example (Section 4.6.1). We then introduce the proof rules for prophecy variables in ReLoC (Section 4.6.2), and use them to verify the motivational example (Section 4.6.3). We finish with another example demonstrating the applicability of prophecy variables to algebraic reasoning for concurrent programs (Section 4.6.4).

4.6.1 Motivational example

The ghost state mechanism of Iris that we have seen so far has allowed us to reason about the history of the program's execution. However, in some cases that is not enough, and it is required to take the future of the program's execution into account. As a simple motivating example let, us consider the implementations new_coin and new_coin_lazy of a coin module in Figure 4.11. They both implement a virtual coin that can be flipped (using the first closure) and whose value can be read (using the second closure), but there is an important difference. The eager version (new_coin) calculates the flipped value in the flip function immediately—using the rand function in Figure 4.12, which gives a non-deterministic Boolean value—and the read function just reads that value. In contrast, the lazy version (new_coin_lazy) does not perform any non-deterministic calculations in its flip operation flip_lazy. Instead, it sets the value of the coin to "undetermined" (*i.e.*, **None**), and postpones the actual calculation to the read_lazy function.

While these two implementations are rather different, they are contextually equivalent—for clients of the module it is not observable if the coin is flipped eagerly or lazily. To prove that, we wish to establish the following refinements in ReLoC:

$$\mathsf{new\_coin} \precsim \mathsf{new\_coin\_lazy} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool})$$

$$\mathsf{new\_coin\_lazy} \precsim \mathsf{new\_coin} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool})$$

The first refinement can be proved with the tools that we have already described. We start by symbolically executing both implementations, obtaining references c and c_l to the internal state of the eager coin and lazy coin, respectively. We then establish the following invariant linking together the two internal states:

$$\boxed{\exists (b : \mathbb{B}).\ c \mapsto_{\mathsf{i}} b * (c_l \mapsto_{\mathsf{s}} \mathbf{None} \lor c_l \mapsto_{\mathsf{s}} \mathbf{Some}(b))}.$$

This invariant can be easily shown to be preserved during the flip $\precsim$ flip_lazy refinement proof. During the read $\precsim$ read_lazy refinement proof we can choose which value rand () reduces to based on the current value of c, using REL-RAND-R.

However, we cannot prove the second refinement with the same strategy. The problem is that during the flip_lazy $\precsim$ flip refinement we reset the value of c_l (on the left-hand side) to **None**, and then we have to establish a simulation on the right-hand side by picking a value for rand () that will be assigned to c. But this value has to be the same value that is picked by rand () in read_lazy. Thus we have to pick a value "from the future".

The eager and lazy implementation:

$$
\begin{aligned}
\mathsf{new\_coin} &\triangleq \mathsf{let}\ c = \mathsf{ref}(\mathbf{false})\ \mathsf{in} \\
&\quad ((\lambda().\ \mathsf{flip}\ c), (\lambda().\ \mathsf{read}\ c)) \\
\mathsf{flip} &\triangleq \lambda c.\ c \leftarrow \mathsf{rand}\ () \\
\mathsf{read} &\triangleq \lambda c.\ !c
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{new\_coin\_lazy} &\triangleq \mathsf{let}\ c = \mathsf{ref}(\mathbf{Some}(\mathbf{false}))\ \mathsf{in} \\
&\quad ((\lambda().\ \mathsf{flip\_lazy}\ c), (\lambda().\ \mathsf{read\_lazy}\ c)) \\
\mathsf{flip\_lazy} &\triangleq \lambda c.\ c \leftarrow \mathbf{None} \\
\mathsf{read\_lazy} &\triangleq \lambda c.\ \mathsf{match}\ !c\ \mathsf{with} \\
&\qquad \mid \mathbf{Some}(v) \rightarrow v \\
&\qquad \mid \mathbf{None} \rightarrow \\
&\qquad\quad \mathsf{let}\ x = \mathsf{rand}\ ()\ \mathsf{in} \\
&\qquad\quad \mathsf{if}\ \mathsf{CAS}(c, \mathbf{None}, x) \\
&\qquad\quad \mathsf{then}\ x\ \mathsf{else}\ \mathsf{read\_lazy}\ c\ ()
\end{aligned}
$$

The instrumented lazy implementation with prophecy variables:

$$
\begin{aligned}
\overbrace{\mathsf{new\_coin\_lazy}} &\triangleq \mathsf{let}\ c = \mathsf{ref}(\mathbf{Some}(\mathbf{false}))\ \mathsf{in} \\
&\quad \mathsf{let}\ p = \mathsf{newproph}\ \mathsf{in} \\
&\quad \mathsf{let}\ lk = \mathsf{newlock}\ ()\ \mathsf{in} \\
&\quad \Big((\lambda().\ \overbrace{\mathsf{flip\_lazy}}\ c\ lk), (\lambda().\ \overbrace{\mathsf{read\_lazy}}\ c\ lk\ p)\Big) \\
\overbrace{\mathsf{flip\_lazy}} &\triangleq \lambda c\ lk.\ \mathsf{acquire}\ lk;\ c \leftarrow \mathbf{None};\ \mathsf{release}\ lk \\
\overbrace{\mathsf{read\_lazy}} &\triangleq \lambda c\ lk\ p.\ \mathsf{acquire}\ lk; \\
&\qquad \mathsf{let}\ r = \mathsf{match}\ !c\ \mathsf{with} \\
&\qquad\quad \mid \mathbf{Some}(v) \rightarrow v \\
&\qquad\quad \mid \mathbf{None} \rightarrow \\
&\qquad\qquad \mathsf{let}\ x = \mathsf{rand}\ ()\ \mathsf{in} \\
&\qquad\qquad c \leftarrow \mathbf{Some}(x); \\
&\qquad\qquad \mathsf{resolve}\ p\ \mathsf{to}\ x;\ x \\
&\qquad \mathsf{in}\ \mathsf{release}\ lk;\ r
\end{aligned}
$$

Figure 4.11: The implementations of the coin module.

$$\mathsf{rand} \triangleq \lambda().\ \mathsf{let}\ y = \mathsf{ref}(\mathbf{false})\ \mathsf{in}\ \mathsf{fork}\ \{y \leftarrow \mathbf{true}\};\ !y$$

$$\textsc{rel-rand-l}\ \frac{\forall b \in \mathbb{B}.\ (K[\,b\,] \precsim t : \tau)}{K[\,\mathsf{rand}\ ()\,] \precsim t : \tau} \qquad \textsc{rel-rand-r}\ \frac{b \in \mathbb{B} \qquad e \precsim K[\,b\,] : \tau}{e \precsim K[\,\mathsf{rand}\ ()\,] : \tau}$$

Figure 4.12: The implementation and relational specification of the rand function.

$$\textsc{rel-newproph-l}\ \frac{\forall \vec{v}\ p.\ \mathsf{proph}(p, \vec{v}) \twoheadrightarrow \Delta \models K[\,p\,] \precsim e_2 : \tau}{\Delta \models K[\,\mathsf{newproph}\,] \precsim e_2 : \tau}$$

$$\textsc{rel-resolveproph-l}\ \frac{{}^{\top}\!\Rightarrow^{\mathcal{E}} \exists \vec{v}.\ \mathsf{proph}(p, \vec{v}) * \left(\forall \vec{w}.\ (\vec{v} = w :: \vec{w}) * \mathsf{proph}(p, \vec{w}) \twoheadrightarrow \Delta \models_{\mathcal{E}} K[\,()\,] \precsim e_2 : \tau\right)}{\Delta \models K[\,\mathsf{resolve}\ p\ \mathsf{to}\ w\,] \precsim e_2 : \tau}$$

$$\textsc{rel-newproph-r}\ \frac{\forall p.\ (\Delta \models_{\mathcal{E}} e_1 \precsim K[\,p\,] : \tau)}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{newproph}\,] : \tau} \qquad \textsc{rel-resolveproph-r}\ \frac{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,()\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{resolve}\ p\ \mathsf{to}\ w\,] : \tau}$$

Figure 4.13: The ReLoC proof rules for prophecy variables.

To facilitate this style of reasoning, prophecy variables have been introduced into Iris [Jun+20]. Originally, prophecy variables were used to prove refinements between state machines [AL91]. Lately they have been used in Iris for establishing linearizability of concurrent data structures without a fixed linearization point. In the rest of this section we show how we integrated prophecy variables into ReLoC.

4.6.2 Prophecy instructions and proof rules

While Iris's ordinary ghost state mechanism only appears at the level of the logic, prophecy variables appear as instrumented instructions in the source program.[4] The instruction newproph creates a new prophecy variable. The instruction resolve p to v resolves a prophecy variable p to a value v.

The symbolic execution rules for the prophecy instructions are given in Figure 4.13. In the right-hand side, the prophecy instructions are no-ops and therefore do not have any pre- or post-conditions. Prophecy instructions that appear on the left-hand side, however, operate on additional ghost state, and thus have pre- and postconditions. The ghost predicate $\mathsf{proph}(p, \vec{v})$ says that the prophecy variable p

[4] The semantics of HeapLang has to be instrumented to support prophecy variables, we refer the reader to [Jun+20, Section 3] for details.

will be resolved, in the future, with values from the vector $\vec{v}$. Initially, a prophecy variable created with newproph has an arbitrary vector $\vec{v}$ associated with it. Only after symbolically executing resolve p to w we learn that this vector $\vec{v}$ contains w at the head position. The trick behind the prophecy variables ghost state is that we can already refer to the head element of $\vec{v}$ before resolving it to some w. We will see how to use this in establishing the refinement between the lazy coin and the eager coin in the next section. Note that the rules for the left-hand side are written in logically atomic style: compare, for example, REL-RESOLVEPROPH-L and REL-STORE-L.

To see how the instrumented instructions for prophecy variables are used, suppose we want to prove a contextual refinement $e_1 \precsim_{ctx} e_2 : \tau$ that involves speculative reasoning. We first prove a refinement $\hat{e}_1 \precsim e_2 : \tau$, where $\hat{e}_1$ is a version of e_1 instrumented with prophecy variables, and then prove $e_1 \precsim \hat{e}_1 : \tau$ to show that the prophecy variables can be erased. By soundness of ReLoC and transitivity of contextual refinement, this gives a contextual refinement $e_1 \precsim_{ctx} e_2 : \tau$ that refers only to the original programs.

4.6.3 Proving the coin refinement

To prove the refinement $\mathsf{new\_coin\_lazy} \precsim \mathsf{new\_coin} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool})$ from Section 4.6.1 we instrument the lazy implementation new_coin_lazy with prophecy variables so we can speculate on the outcome of rand in read_lazy. The instrumented implementation $\widehat{\mathsf{new\_coin\_lazy}}$ is shown in Figure 4.12. In addition to prophecy variables, we also instrumented the implementation with locks to ensure that there is no interference between updating the reference c and resolving the prophecy variable p. With the instrumented program at hand, we will prove the chain of refinements:

$$\mathsf{new\_coin\_lazy} \precsim \widehat{\mathsf{new\_coin\_lazy}} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool})$$
$$\widehat{\mathsf{new\_coin\_lazy}} \precsim \mathsf{new\_coin} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool}).$$

Via ReLoC's soundness theorem, we can compose these refinements at the level of contextual refinement to obtain:

$$\mathsf{new\_coin\_lazy} \precsim_{ctx} \mathsf{new\_coin} : (\mathsf{unit} \to \mathsf{unit}) \times (\mathsf{unit} \to \mathsf{bool}).$$

Note that that we only use the instrumented implementation $\widehat{\mathsf{new\_coin\_lazy}}$ for the intermediate step, which means that prophecy variables and locks do not appear at all in the final statement above. The approach of using prophecies as an intermediate step works not just for closed programs, but also for open programs, as it does not rely on an erasure theorem [Jun+20, Section 3.5]. Moreover, as the example demonstrates, it allows us to make use of locks in the instrumented program.[5].

The first refinement ($\mathsf{new\_coin\_lazy} \precsim \widehat{\mathsf{new\_coin\_lazy}}$) is easy to prove, we simply use the no-op symbolic execution rules for prophecies on the right-hand side

[5] *Atomic prophecy resolution* was introduced in [Jun+20] as an alternative to locks to deal with atomicity of prophecy resolution.

(Figure 4.13). The second refinement ($\overbrace{\text{new\_coin\_lazy}}^{} \precsim \text{new\_coin}$) is where the mechanism of prophecy variables comes to help. We symbolically execute the allocation parts of $\overbrace{\text{new\_coin\_lazy}}^{}$ and new_coin. We then use the relational specification for locks (Section 4.5.1.2) with the following lock invariant:

$$\exists \vec{v}.\, \mathsf{proph}(p, \vec{v}) * \Big((c_l \mapsto_{\mathsf{i}} \mathbf{None} * c \mapsto_{\mathsf{s}} (hd\ \vec{v})) \vee\ (\exists (b : \mathbb{B}).\, c_l \mapsto_{\mathsf{i}} \mathbf{Some}(b) * c \mapsto_{\mathsf{s}} b) \Big).$$

This invariant says that if the value of the lazy coin is **None**, then the value of the eager coin is determined by the prophecy variable p. There are two main implications of this:

1. In the refinement between $\overbrace{\text{flip\_lazy}}^{}$ and flip, the invariant can be (re)established, because we can pick the value of rand () on the right-hand side to be the head element of $\vec{v}$—the future value of the lazy coin is already bound at this point.
2. In the refinement between $\overbrace{\text{read\_lazy}}^{}$ and read (specifically, in the **None** branch), we obtain a non-deterministic Boolean x from symbolically executing rand () on the left-hand side, and we update the value of c_l to be x. Moreover, we resolve the prophecy variable p to x, which gives us much desired information: the head element of $\vec{v}$ was x all along! This information allows us to transition from the left disjunct to the right disjunct in the invariant and complete the proof.

4.6.4 Algebraic reasoning about non-deterministic choice

In this section we give another example of the use of prophecy variables: we verify several algebraic properties of non-deterministic choice modulo contextual equivalence. (In)equational theories of the non-deterministic choice operator were previously considered in the context of domain theory, where non-determinism is usually modeled using power domains [Plo76; Smy76], and in the context of algebraic effects [SV20; JSV10]. Power domains and the denotational semantics approach does not seem to scale easily to languages with concurrency and higher-order store. An operational approach to equational theory of a programming language with non-determinism was considered in [BBS13] using step-indexed logical relations. There the authors show several contextual equivalences involving non-determinism, both finite (*e.g.*, picking a Boolean) and countable (*e.g.*, picking a natural number). In this subsection, we provide conceptually simple proofs for contextual equivalences involving finite non-determinism only. However, we were also able to prove that non-deterministic choice and sequential composition distribute over each other. Proving this crucially relies on speculative reasoning which we formalize using prophecy variables.

We do not have a non-deterministic choice operation built-in the language, but we can define it using the rand function from Section 4.6.1. The operation or non-deterministically executes one of its thunked arguments:

$$\mathsf{or}\ t_1\ t_2 \triangleq \mathsf{if}\ \mathsf{rand}\ ()\ \mathsf{then}\ t_1\ ()\ \mathsf{else}\ t_2\ ()$$

We write $e_1 \oplus e_2$ for $\mathsf{or}\ (\lambda().\ e_1)\ (\lambda().\ e_2)$. The expression $e_1 \oplus e_2$ thus non-deterministically reduces to either e_1 or e_2. From the rules for the rand function

(Figure 4.12), we derive the following symbolic execution rules for $\oplus$:

$$\textsc{rel-or-l}\ \frac{(\Delta \models K[e_1] \precsim t : \tau) \land (\Delta \models K[e_2] \precsim t : \tau)}{\Delta \models K[e_1 \oplus e_2] \precsim t : \tau}$$

$$\textsc{rel-or-r-1}\ \frac{\Delta \models_{\mathcal{E}} t \precsim K[e_1] : \tau}{\Delta \models_{\mathcal{E}} t \precsim K[e_1 \oplus e_2] : \tau} \qquad \textsc{rel-or-r-2}\ \frac{\Delta \models_{\mathcal{E}} t \precsim K[e_2] : \tau}{\Delta \models_{\mathcal{E}} t \precsim K[e_1 \oplus e_2] : \tau}$$

The rules for $\oplus$ are reminiscent of the rules for disjunction ($\lor$) in sequent calculus. To symbolically execute $\oplus$ on the left-hand side (*c.f.* to eliminate $\lor$) it is necessary to establish refinements for both operands (*c.f.* to consider both disjuncts), and to symbolically execute $\oplus$ on the right-hand side (*c.f.* to introduce $\lor$) it suffices to establish a refinement for one of the operands (*c.f.* prove one of the disjuncts).

Assume that e_1, e_2, e_3 are closed programs of type τ. Then using rel-or-r-1, rel-or-r-2, and rel-or-l, we prove the following equivalences:

$$e_1 \simeq_{ctx} e_1 \oplus e_1 : \tau \qquad e_1 \oplus e_2 \simeq_{ctx} e_2 \oplus e_1 : \tau \qquad e_1 \simeq_{ctx} e_1 \oplus \mathsf{diverge} : \tau$$

$$e_1 \oplus (e_2 \oplus e_3) \simeq_{ctx} (e_1 \oplus e_2) \oplus e_3 : \tau \qquad (e_1 \oplus e_2); e_3 \simeq_{ctx} (e_1; e_3) \oplus (e_2; e_3) : \tau$$

The equational theory that we obtain here is similar to the one obtained from the Hoare power domain, as $e_1 \oplus \mathsf{diverge}$ (where diverge is an infinite loop) is identified with e_1. The last equation states that non-deterministic choice distributes over sequential composition, and is standard in, *e.g.*, process calculi. What is less standard is the following equation, which is not validated by models based on bisimulation:

$$e_1; (e_2 \oplus e_3) \simeq_{ctx} (e_1; e_2) \oplus (e_1; e_3) : \tau.$$

This equation, however, holds in Kleene algebra-like models [Hoa+11; Koz94]. If we think about proving this equation using the symbolic execution rules for $\oplus$, then we can observe that proving the refinement in right-to-left direction

$$(e_1; e_2) \oplus (e_1; e_3) \precsim e_1; (e_2 \oplus e_3) : \tau$$

is possible, and, by rel-or-l, it boils down to proving two refinements:

$$e_1; e_2 \precsim e_1; (e_2 \oplus e_3) : \tau \qquad e_1; e_3 \precsim e_1; (e_2 \oplus e_3) : \tau.$$

However, proving the refinement in left-to-right direction is harder:

$$e_1; (e_2 \oplus e_3) \precsim (e_1; e_2) \oplus (e_1; e_3) : \tau.$$

If we want to use the symbolic execution rules for $\oplus$, we have to "synchronize" both sides on e_1. To do that, we have to pick a branch for $\oplus$ on the right-hand side before we get to use rel-or-l on the left-hand side, but we do not know ahead of time which branch to pick. To resolve this dependency, we use a prophecy variable to speculate on which branch $e_2 \oplus e_3$ will be taken on the left-hand side, and use the value of

this prophecy variable to choose the appropriate branch of $(e_1; e_2) \oplus (e_1; e_3)$ on the right-hand side.

The intermediate program that is instrumented with prophecy variables is as follows:

$$\begin{array}{l}\mathsf{let}\ p = \mathsf{newproph}\ \mathsf{in}\\ e_1;\ \big((\mathsf{resolve}\ p\ \mathsf{to}\ 0; e_2) \oplus (\mathsf{resolve}\ p\ \mathsf{to}\ 1; e_3)\big)\end{array}$$

We can easily verify that the original program $e_1;(e_2 \oplus e_3)$ refines the instrumented one. To verify that the instrumented program refines $(e_1; e_2) \oplus (e_1; e_3)$ we symbolically execute newproph and obtain a predicate $\mathsf{proph}(p, \vec{v})$ associating a vector of future values $\vec{v}$ to the newly created prophecy variable p. Then we examine the head element w of the prophecy values $\vec{v}$. If w is 0, then we apply REL-OR-R-1, otherwise we apply REL-OR-R-2. Without loss of generality, suppose that w is 0; that is, $\vec{v} = 0 :: \vec{w}$ for some tail $\vec{w}$. First we "synchronize" the refinement proof on e_1 on both sides. Then we apply REL-OR-L. Because the premises of REL-OR-L are joined by intuitionistic conjunction $\wedge$, we can use the resource $\mathsf{proph}(p, \vec{v})$ for verifying both refinements:

$$\mathsf{proph}(p, 0 :: \vec{v'}) \twoheadrightarrow \mathsf{resolve}\ p\ \mathsf{to}\ 0; e_2 \precsim e_2 : \tau$$
$$\mathsf{proph}(p, 0 :: \vec{v'}) \twoheadrightarrow \mathsf{resolve}\ p\ \mathsf{to}\ 1; e_3 \precsim e_2 : \tau$$

The first refinement is reduced to $e_2 \precsim e_2 : \tau$, which follows from the fundamental property (Theorem 4.6) and the assumption that e_2 is well-typed. To prove the second refinement we symbolically execute resolve p to 1 on the left-hand side, at which point we reach a contradiction $0 = 1$.

4.7 The logical relations model of ReLoC

ReLoC extends Iris with logical connectives and corresponding proof rules for reasoning about refinements. In this section we show how this is achieved by modeling the connectives of ReLoC through a shallow embedding in Iris and proving the logical rules of ReLoC as mere lemmas in Iris. We describe how the refinement judgment $e_1 \precsim e_2 : \tau$ is modeled through Iris's weakest preconditions and a *ghost thread pool* construction (Section 4.7.1) combined with a *binary logical relation* $[\![\tau]\!]_\Delta$ that describes when values are related (Section 4.7.2). We then summarize how the ReLoC proof rules (Section 4.7.4) and soundness theorem (Section 4.7.5) are proved. The key definitions of the ReLoC model are shown in Figure 4.14.

The construction of our model generalizes prior work by Turon *et al.* [Tur+13; TDB13], which culminated in the CaReSL logic, and was subsequently mechanized in Iris by Krebbers *et al.* [KTB17] and Timany [Tim18]. We discuss the differences in Section 4.7.3.

4.7.1 The refinement judgment

Recall from Section 4.3.1 that the intuitive meaning of the refinement proposition $e_1 \precsim e_2 : \tau$ is that *any* behavior of e_1 can be simulated by *some* behavior of e_2. This

Refinement judgments:

$$\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau \triangleq \forall i, K.\ \mathsf{spec\_ctx} \mathrel{-\!\!*} i \Rightarrow K[e_2] \ {}^{\mathcal{E}}\!\Rightarrow\!\!\!\!\!\!\ast^{\top} \\ \mathsf{wp}\ e_1 \{v_1.\ \exists v_2.\ i \Rightarrow K[v_2] * [\![\tau]\!]_\Delta(v_1, v_2)\}$$

Interpretation of types:

$$\begin{aligned}
[\![\alpha]\!]_\Delta &\triangleq \lambda(v_1, v_2).\ \Delta(\alpha)(v_1, v_2) \\
[\![\mathsf{unit}]\!]_\Delta &\triangleq \lambda(v_1, v_2).\ v_1 = v_2 = () \\
[\![\mathsf{bool}]\!]_\Delta &\triangleq \lambda(v_1, v_2).\ (v_1 = v_2 = \mathbf{true}) \vee (v_1 = v_2 = \mathbf{false}) \\
[\![\mathsf{int}]\!]_\Delta &\triangleq \lambda(v_1, v_2).\ \exists n \in \mathbb{Z}.\ v_1 = v_2 = n \\
[\![\tau \times \sigma]\!]_\Delta &\triangleq \lambda(v_1, v_2).\ \exists w_1, w_2, w_1', w_2'.\ v_1 = (w_1, w_2) * v_2 = (w_1', w_2') * \\
&\qquad [\![\tau]\!]_\Delta(w_1, w_1') * [\![\sigma]\!]_\Delta(w_2, w_2') \\
[\![\tau + \sigma]\!]_\Delta &\triangleq \lambda(v_1, v_2).\ \exists w_1, w_2.\ (v_1 = \mathbf{inl}\ (w_1) * v_2 = \mathbf{inl}\ (w_2) * [\![\tau]\!]_\Delta(w_1, w_2)) \vee \\
&\qquad (v_1 = \mathbf{inr}\ (w_1) * v_2 = \mathbf{inr}\ (w_2) * [\![\sigma]\!]_\Delta(w_1, w_2)) \\
[\![\tau \to \sigma]\!]_\Delta &\triangleq \lambda(v_1, v_2).\ \Box(\forall w_1, w_2.\ [\![\tau]\!]_\Delta(w_1, w_2) \mathrel{-\!\!*} (\Delta \models v_1\ w_1 \precsim v_2\ w_2 : \sigma)) \\
[\![\forall \alpha.\ \tau]\!]_\Delta &\triangleq \lambda(v_1, v_2).\ \Box\Big(\forall \Phi \in \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_{\Box}.\ ([\alpha := \Phi], \Delta \models v_1\langle\rangle \precsim v_2\langle\rangle : \tau)\Big) \\
[\![\exists \alpha.\ \tau]\!]_\Delta &\triangleq \lambda(v_1, v_2).\ \exists \Phi \in \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_{\Box}.\ [\![\tau]\!]_{[\alpha:=\Phi],\Delta}(v_1, v_2) \\
[\![\mu \alpha.\ \tau]\!]_\Delta &\triangleq \mu\Phi.\ \lambda(v_1, v_2).\ \exists w_1, w_2.\ v_1 = \mathsf{fold}(w_1) * v_2 = \mathsf{fold}(w_2) * \triangleright [\![\tau]\!]_{[\alpha:=\Phi],\Delta}(w_1, w_2) \\
[\![\mathsf{ref}\ \tau]\!]_\Delta &\triangleq \lambda(v_1, v_2).\ \exists \ell_1, \ell_2 \in \mathit{Loc}.\ v_1 = \ell_1 * v_2 = \ell_2 * \\
&\qquad \boxed{\exists w_1, w_2.\ \ell_1 \mapsto_{\mathsf{i}} w_1 * \ell_2 \mapsto_{\mathsf{s}} w_2 * [\![\tau]\!]_\Delta(w_1, w_2)}^{(\ell_1, \ell_2)}
\end{aligned}$$

Figure 4.14: The model of ReLoC in Iris.

intuitive idea is modeled in Iris as follows:

$$\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau \triangleq \forall i, K.\ \mathsf{spec\_ctx} \mathrel{-\!\!*} i \Rightarrow K[e_2] \ {}^{\mathcal{E}}\!\Rightarrow\!\!\!\!\!\!\ast^{\top} \\ \mathsf{wp}\ e_1 \{v_1.\ \exists v_2.\ i \Rightarrow K[v_2] * [\![\tau]\!]_\Delta(v_1, v_2)\}$$

This definition is quite a mouthful, so let us go over it piece by piece. First, it involves Iris's *weakest precondition* connective $\mathsf{wp}\ e\ \{\Phi\}$, which gives the weakest precondition under which execution of e is safe, and when e returns with value v, the postcondition $\Phi(v)$ holds. Second, it involves the *ghost thread pool* connective $i \Rightarrow e$, which is defined through Iris's ghost theory, and states that the i-th ghost thread is executing a program e. Putting these pieces together (ignoring $\mathsf{spec\_ctx}$ and ${}^{\mathcal{E}}\!\Rightarrow\!\!\!\!\!\!\ast^{\top}$ for now), this definition states that if a (ghost) thread i is executing right-hand side e_2, and left-hand side e_1 reduces to some value v_1, then a corresponding execution can be

$$\textsc{step-pure}\quad \frac{\mathsf{spec\_ctx} \qquad i \Rightarrow e \qquad e \to_{\mathsf{pure}} e'}{\models_{\mathcal{E}} i \Rightarrow e'} \qquad \textsc{step-alloc}\quad \frac{\mathsf{spec\_ctx} \qquad i \Rightarrow K[\,\mathsf{ref}(v)\,]}{\models_{\mathcal{E}} \exists \ell.\, i \Rightarrow K[\,\ell\,] * \ell \mapsto_{\mathsf{s}} v}$$

$$\textsc{step-store}\quad \frac{\mathsf{spec\_ctx} \qquad i \Rightarrow K[\,\ell \leftarrow w\,] \qquad \ell \mapsto_{\mathsf{s}} v}{\models_{\mathcal{E}} i \Rightarrow K[\,()\,] * \ell \mapsto_{\mathsf{s}} w} \qquad \textsc{step-fork}\quad \frac{\mathsf{spec\_ctx} \qquad i \Rightarrow K[\,\mathsf{fork}\ \{e\}\,]}{\models_{\mathcal{E}} \exists j.\, i \Rightarrow K[\,()\,] * j \Rightarrow e}$$

Figure 4.15: Selected rules for the ghost thread pool.

made so that (ghost) thread i is executing right-hand side v_2. The result values v_1 and v_2 of the left-hand and right-hand side should be related via the value interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$, which we model in Section 4.7.2 via a logical relation. The quantification over K closes the definition under evaluation contexts. The expression e_1 on the left-hand side does not need to be closed under evaluation contexts because weakest preconditions enjoy the rule: $\mathsf{wp}\, e\, \{w.\, \mathsf{wp}\, K[\,w\,]\, \{\Phi\}\} \twoheadrightarrow \mathsf{wp}\, K[\,e\,]\, \{\Phi\}$.

The ghost thread pool predicates satisfy a number of symbolic execution rules corresponding to executions in the operational semantics. A selection of these rules is given in Figure 4.15. The spec_ctx proposition is an Iris invariant that ties together the thread pool connectives $i \Rightarrow e$ and the heap assertions $\ell \mapsto_{\mathsf{s}} v$ with a matching execution on the right-hand side. We will explain the role of spec_ctx in Section 4.7.5.

We should emphasize that the combination of the weakest precondition and the ghost thread pool in the definition of $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ model the demonic nature of e_1 and the angelic nature of e_2. To prove the weakest precondition $\mathsf{wp}\, e_1\, \{v_1.\, \exists v_2.\, i \Rightarrow K[\,v_2\,] * [\![\tau]\!]_\Delta(v_1, v_2)\}$ one has to consider all behaviors of e_1, but has to establish only a single matching execution for e_2 by using the appropriate rules for the ghost thread pool.

4.7.2 The logical relation

The interpretation of types $[\![\tau]\!]_\Delta(v_1, v_2)$, as defined in Figure 4.14, expresses when two values v_1 and v_2 are related at type τ (in context Δ). The definition of $[\![\tau]\!]_\Delta(v_1, v_2)$ follows the usual structure of a logical relation, it is defined recursively on the structure of the type τ and uses the corresponding logical connectives via the Curry-Howard isomorphism. For example, products are defined via (separating) conjunction, sums are defined via disjunction, functions are defined via (separating) implication, universal types are defined via universal quantification, *etc.*

The interpretation of recursive types and reference types are somewhat more interesting, as they make use of Iris-specific connectives. The interpretation of the recursive type $\mu\alpha.\, \tau$ makes use of Iris's guarded fixed point operator $\mu x.\, t$, which is used to define recursive predicates without a restriction of the variance of the recursive occurrence x in t, but requires x to appear in *guarded* position, *i.e.*, under the later modality $\triangleright$ [Jun+18b, Section 5.6]. To define the interpretation of the

reference type ref τ, we use the invariant

$$\boxed{\exists w_1, w_2.\, \ell_1 \mapsto_{\mathsf{i}} w_1 * \ell_2 \mapsto_{\mathsf{s}} w_2 * [\![\tau]\!]_\Delta(w_1, w_2)}^{(\ell_1,\ell_2)},$$

which states that whatever values are stored in ℓ_1 and ℓ_2 are always related at type $[\![\tau]\!]_\Delta$.

The persistence modality $\square$ in the interpretation for function types and universal types is used to ensure that the type interpretation is persistent and prevents the kind of issues described in Section 4.4.2. Similarly, in the interpretation of the universal and existential types we quantify over a *persistent* predicate $\Phi \in \mathit{Val} \times \mathit{Val} \to \mathit{iProp}_\square$, where $\mathit{iProp}_\square$ is the subset of Iris propositions that is persistent.

4.7.3 Differences with prior work.

The definition of the refinement $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ and value interpretation $[\![\tau]\!]_\Delta(v_1, v_2)$ generalize the versions by Krebbers *et al.* [KTB17] and Timany *et al.* [Tim18], which in turn adapted ghost thread pools by Turon *et al.* [Tur+13; TDB13] by modeling these in Iris. The main novelty is that our refinement judgment $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ is a first-class Iris proposition, instead of a meta-logical proposition. As we have demonstrated throughout this paper, this modification is simple, albeit crucial for writing conditional refinements and to obtain high-level proof rules for refinements.

Furthermore, to obtain high-level proof rules for invariants, we have equipped the refinement judgment with a mask $\mathcal{E}$, which keeps track of the invariants that may be opened. To give the appropriate semantics to the mask $\mathcal{E}$, our definition involves the update modality ${}^{\mathcal{E}}\!\Rightarrow\!\!\!\!\!\!\ast^{\top}$. Note that the definition by Krebbers *et al.* [KTB17] and Timany [Tim18] is logically equivalent to $\vdash \Delta \models_\top e_1 \precsim e_2 : \tau$, where the derivability relation $\vdash$ of Iris is used to turn the judgment into a meta theoretical proposition, and the mask is set to $\top$.

4.7.4 Deriving the primitive rules

In Section 4.4.3 we have demonstrated that ReLoC's primitive monadic (REL-RETURN and REL-BIND) and symbolic execution rules can be used to derive ReLoC's high-level proof rules, such as its type-directed structural rules. In this section, we indicate how ReLoC's primitive rules are proved by unfolding the definition of the refinement judgment. We prove the symbolic execution rules through the following auxiliary rules, which allow us to lift Iris's rules for weakest preconditions and the ghost thread pool rules (Figure 4.15) to the refinement judgment:

REL-WP-L
$$\frac{\mathsf{wp}_\top\, e_1\, \{v_1.\, K[\,v_1\,] \precsim e_2 : \tau\}}{K[\,e_1\,] \precsim e_2 : \tau}$$

REL-WP-ATOMIC-L
$$\frac{{}^{\top}\!\Rightarrow^{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e_1\, \{v_1.\ \models_{\mathcal{E}} K[\,v_1\,] \precsim e_2 : \tau\} \qquad \mathsf{atomic}(e_1)}{K[\,e_1\,] \precsim e_2 : \tau}$$

REL-STEP-R
$$\frac{\forall j, K'.\, \mathsf{spec\_ctx} * j \Rightarrow K'[K[e_2]] \Rightarrow\!\!\!\!\ast_{\mathcal{E}}\, \exists v_2.\, j \Rightarrow K'[K[v_2]] * \models_{\mathcal{E}} e_1 \precsim K[v_2] : \tau}{\models_{\mathcal{E}} e_1 \precsim K[e_2] : \tau}$$

The rule REL-WP-L says that we can "take out" an expression e_1 in context K on the left-hand side, and reason about it using Iris's weakest precondition. The rule REL-WP-ATOMIC-L is similar, but it also allows for opening an invariant around e_1, in case e_1 is atomic.[6] The rule REL-STEP-R says that if we have an expression e_2 on the right-hand side in an evaluation context K, and we can reduce e_2 to a value v_2, using the ghost thread pool rules, then we can reduce the refinement proposition to $\models_{\mathcal{E}} e_1 \precsim K[\,v_2\,] : \tau$.

4.7.5 Soundness

Utilizing the definitions in this section, we outline the proof of the soundness theorem (Theorem 4.7), which says that ReLoC's refinement judgment is sound w.r.t. contextual refinement. Formally, if $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ is derivable in ReLoC for any Δ with $\Xi \subseteq \mathsf{dom}(\Delta)$, then $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$. To prove this theorem we make use of two key lemmas: adequacy of the refinement judgment (Theorem 4.12), and the fact that the refinement judgment is a precongruence (Lemma 4.13).

Theorem 4.12 (Adequacy of ReLoC)**.** If $\vdash \Delta \models e_1 \precsim e_2 : \tau$ is derivable in ReLoC, and $(e_1, \sigma) \to^*_{\mathsf{tp}} (v_1 :: \vec{e_f}_1, \sigma_1')$, then there exists $v_2, \vec{e_f}_2$, and σ_2' such that $(e_2, \sigma) \to^*_{\mathsf{tp}} (v_2 :: \vec{e_f}_2, \sigma_2')$.

Lemma 4.13. Let $\mathcal{C}$ be a well-typed context $\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\Xi' \mid \Gamma' \vdash \tau')$, then we have

$$\square(\forall \Delta.\, \Delta \mid \Gamma \models e_1 \precsim e_2 : \tau) \mathrel{-\!\!*} (\forall \Delta'.\, \Delta' \mid \Gamma' \models \mathcal{C}[\,e_1\,] \precsim \mathcal{C}[\,e_2\,] : \tau')$$

where Δ and Δ' contain at least the type variables in Ξ and Ξ', respectively.

Lemma 4.13 is proved by induction on $\mathcal{C}$ making using of ReLoC's type-directed structural rules (Section 4.4.4). The proof of Theorem 4.12 is rather involved, so before we discuss that, let us see how we prove the soundness theorem by putting these two lemmas together.

Proof of Theorem 4.7 (Soundness for open terms). Let Ξ be a type environment, and suppose that $\Delta \mid \Gamma \models e_1 \precsim e_2 : \tau$ is derivable in ReLoC for any Δ with $\Xi \subseteq \mathsf{dom}(\Delta)$. To prove $\Xi \mid \Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$, suppose we have typed context $\mathcal{C} : (\Xi \mid \Gamma \vdash \tau) \Rightarrow (\emptyset \mid \emptyset \vdash \tau')$, and reduction $(\mathcal{C}[\,e_1\,], \emptyset) \to^*_{\mathsf{tp}} (v_1 :: \vec{e_f}_1, \sigma_1)$. By Lemma 4.13, we have $\mathcal{C}[\,e_1\,] \precsim \mathcal{C}[\,e_2\,] : \tau'$. Then, by Theorem 4.12, we get that $(\mathcal{C}[\,e_2\,], \emptyset) \to^*_{\mathsf{tp}} (v_2 :: \vec{e_f}_2, \sigma_2)$ for some $v_2, \vec{e_f}_2$ and σ_2, which concludes the proof. □

Proof of Theorem 4.12 (Adequacy of ReLoC). Suppose that $\Delta \models e_1 \precsim e_2 : \tau$ is derivable in ReLoC, and we have $(e_1, \sigma) \to^*_{\mathsf{tp}} (v_1 :: \vec{e_f}_1, \sigma_1')$. Now we should exhibit $v_2, \vec{e_f}_2$, and σ_2' such that $(e_2, \sigma) \to^*_{\mathsf{tp}} (v_2 :: \vec{e_f}_2, \sigma_2')$. The high-level structure of the proof is as follows. First, we allocate the thread pool invariant spec_ctx and $0 \Mapsto e_2$ for the main-thread of the right-hand side. Second, by definition of the refinement judgment, we obtain a weakest precondition $\mathsf{wp}\ e_1\ \{v_1.\, \exists v_2.\, 0 \Mapsto v_2 * [\![\tau]\!]_\Delta(v_1, v_2)\}$. Third, by opening

[6] Iris's weakest precondition connective $\mathsf{wp}_{\mathcal{E}}\ e\ \{\Phi\}$ is also equipped with a mask to keep track of which invariants may be opened. This was the inspiration for the mask annotation at ReLoC's refinement judgment.

spec_ctx and using adequacy of Iris's weakest preconditions, we obtain $(e_2, \sigma) \to^*_{\mathsf{tp}} (v_2 :: \vec{e_f}_2, \sigma_2')$.

Carrying out these steps in detail—notably, setting up the required ghost theory for the ghost thread pool—involves some intricate reasoning using Iris features that are out of scope for this paper. We thus refer the interested reader to the Coq mechanization, and only highlight the key part—the definition of the thread pool invariant spec_ctx:

$$\mathsf{spec\_ctx} \triangleq \exists \vec{e}_0, \sigma_0. \boxed{\exists \vec{e}, \sigma. \mathsf{spec\_inv}(\vec{e}, \sigma) * (\vec{e}_0, \sigma_0) \to^*_{\mathsf{tp}} (\vec{e}, \sigma)}^{\mathcal{N}_{\mathrm{ReLoC}}}.$$

The invariant asserts that given an initial configuration $(\vec{e}_0, \sigma_0)$ for the right-hand side (which we set to be (e_2, σ) when allocating the invariant), the configuration $(\vec{e}, \sigma)$ can be reached via the reduction $(\vec{e}_0, \sigma_0) \to^*_{\mathsf{tp}} (\vec{e}, \sigma)$. Here, $\mathsf{spec\_inv}(\vec{e}, \sigma)$ is a connective defined using Iris's ghost theory that keeps track of the configuration of the ghost thread pool and ensures it is consistent with the $\Rightarrow$ and $\mapsto_{\mathsf{s}}$ connectives. The latter is essential, as it allows us to conclude from $\mathsf{spec\_inv}(\vec{e}, \sigma)$ and $0 \Rightarrow v_2$ (as given by the post condition of the weakest precondition in the definition of the refinement judgment) that $\vec{e}$ is equal to $v_2 :: \vec{e_f}_2$ for some $\vec{e_f}_2$. By definition of the invariant spec_ctx, this gives us a reduction $(e_2, \sigma) \to^*_{\mathsf{tp}} (v_2 :: \vec{e_f}_2, \sigma_2')$ for the right-hand side, which is needed to conclude the third step of the proof. □

4.8 The Coq mechanization of ReLoC

The Coq mechanization of ReLoC provides a soundness proof of ReLoC and infrastructure to carry out interactive tactic-based refinement proofs. It is built on top of the mechanization of Iris in Coq [Iri20] and the Iris Proof Mode/MoSeL framework for tactic-based proofs in separation logic [KTB17; Kre+18]. In this section we examine the way ReLoC's language and type system are defined (Section 4.8.1), and how the ReLoC logic is defined on top of that (Section 4.8.2). We then describe ReLoC's tactic support for interactive refinement proofs, which allows us to seamlessly carry out proofs in Coq similar to those we have seen in this paper (Section 4.8.3). Finally, we give an overview of the source code (Section 4.8.4).

4.8.1 The programming language

Iris is a programming language independent framework, which means that it can be instantiated with a programming language of choice. In this paper, we do not make use of this generality, and use HeapLang—the default language shipped with Iris's Coq development, which is essentially an untyped version of the language we considered in Section 4.2. HeapLang is represented via a deep embedding and comes with a set of notations so that programs can be written in Coq-style syntax. For example, the Boolean implementation bitbool of the bit module from Section 4.3.3 is written as follows:

```
Definition bit_bool : expr :=
  (#true, (λ: "b", ~"b"), (λ: "b", "b")).
```

Binders in HeapLang are represented as strings, which makes it possible to write programs in a human-readable way. This works well in practice because expression-level substitution only acts on closed terms, and thus does not need to be capture avoiding.

We equip HeapLang with a type system in the usual way—types `type` are defined as an inductive data type, and the typing judgment `typed` is defined as an inductive relation:

```
Inductive type :=
  | TVar : var → type
  | TProd : type → type → type
  | TArrow : type → type → type
  | TExists : {bind 1 of type} → type
  | (* ... *).
Inductive typed : stringmap type → expr → type → Prop :=
  | Var_typed Γ x τ :
     Γ !! x = Some τ → (Γ ⊢ₜ Var x : τ)
  | Pair_typed Γ e₁ e₂ τ₁ τ₂ :
     (Γ ⊢ₜ e₁ : τ₁) → (Γ ⊢ₜ e₂ : τ₂) → (Γ ⊢ₜ (e₁, e₂) : τ₁ * τ₂)
  | Fst_typed Γ e τ₁ τ₂ :
     (Γ ⊢ₜ e : τ₁ * τ₂) → (Γ ⊢ₜ Fst e : τ₁)
  | (* ... *).
```

We use the notation $\Gamma \vdash_t \texttt{e} : \tau$ for `typed Γ e τ`, and overload the standard Coq notations for types, *e.g.*, we use the notation $\tau_1 * \tau_2$ for `TProd` $\tau_1\,\tau_2$ and $\tau_1 \to \tau_2$ for `TArrow` $\tau_1\,\tau_2$. Since type-level substitution acts on (potentially) open terms, and therefore needs to be capture avoiding, we use De Bruijn indices to represent type-level binders through the Autosubst Coq library [STS15]. For example, the type $\mathsf{TBit} \triangleq \exists\alpha.\alpha \times (\alpha \to \alpha) \times (\alpha \to \mathsf{bool})$ from Section 4.3.3 is represented in Coq as follows (`#` is notation for `TVar`):

```
Definition bitτ : type := ∃: #0 * (#0 → #0) * (#0 → TBool).
```

4.8.2 The ReLoC logic

Recall from Section 4.7 that ReLoC is defined as a shallow definition in Iris—the ReLoC connectives are definitions in Iris, and the ReLoC proof rules are lemmas in Iris. In Coq we follow the same approach. At the core of ReLoC we have the definition `lrel` of *semantic types*, *i.e.*, persistent Iris relations over HeapLang values:

```
Record lrel Σ := LRel {
  lrel_car :> val → val → iProp Σ;
  lrel_persistent v₁ v₂ : Persistent (lrel_car v₁ v₂)
}.
```

Here, `iProp Σ` is the type of Iris propositions.[7] The record bundles together a relation together with a proof that it is persistent. The notation :> declares the field `lrel_car` as a coercion. In the Coq mechanization of ReLoC we generalize the refinement judgment $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ to range over semantic types (`lrel`) instead of syntactic types (`type`):

```
Definition refines (E : coPset) (e1 e2 : expr) (A : lrel Σ)
    : iProp Σ := ∀ j K, spec_ctx -∗ j ⤇ fill K e2 ={E,⊤}=∗
          WP e1 {{ v1, ∃ v2, j ⤇ fill K (of_val v2) ∗ A v1 v2 }}.
```

We use the notation `REL e1 << e2 @ E : A` for `refines E e1 e2 A`. This definition makes use of the ghost thread pool connectives `spec_ctx` and `j ⤇ e`, as discussed in Section 4.7.1, and which were originally defined in Coq in [KTB17].

To formalize the refinement judgment on syntactic types, we first define the semantic interpretation $[\![\tau]\!]_\Delta$, denoted as `interp τ Δ` in Coq, which maps syntactic types τ to semantic types. To define the semantic interpretation, we define semantic type formers, which are combinators on semantic types corresponding to each syntactic type former. For example, the semantic product type is defined as follows:

```
Definition lrel_prod (A B : lrel Σ) : lrel Σ := LRel (λ v1 v2,
  ∃ w1 w2 w1' w2', ⌜v1 = (w1,w1')%V⌝ ∧ ⌜v2 = (w2,w2')%V⌝ ∧ A w1 w2 ∗ B w1' w2').
```

Here, we use Iris's notion $\ulcorner\varphi\urcorner$ to embed Coq propositions φ : `Prop` into Iris, although on paper we take the equality predicate to be primitive. With the above definitions at hand, we can now define ReLoC's refinement judgment $\Delta \models_{\mathcal{E}} e_1 \precsim e_2 : \tau$ as `REL e1 << e2 @ E : interp τ Δ`.

The proof rules. For example, the rule REL-LOAD-R is formalized as the following lemma:

```
Lemma refines_load_r E K l q v e1 A :
  ↑ relocN ⊆ E →
  l ↦s{q} v -∗
  (l ↦s{q} v -∗ REL e1 << fill K (of_val v) @ E : A) -∗
  REL e1 << fill K !#l @ E : A.
```

The lemma states that, under the assumption that $\mathsf{relocN}^{\uparrow} \subseteq \mathcal{E}$ (*i.e.*, ReLoC's internal invariants are available in the mask $\mathcal{E}$), the following separation logic formula holds:

$$\ell \overset{q}{\mapsto}_{\mathsf{s}} v \twoheadrightarrow (\ell \overset{q}{\mapsto}_{\mathsf{s}} v \twoheadrightarrow \models_{\mathcal{E}} t \precsim K[v] : A) \twoheadrightarrow \models_{\mathcal{E}} t \precsim K[!\ell] : A$$

This is exactly the internalization of REL-LOAD-R. The other ReLoC proof rules are mechanized in a similar way.

[7] The parameter Σ describes the kind of ghost state available in Iris. It is an important but technical detail that can safely be ignored for the purpose of this paper. An interested reader is directed to [Jun+18b, §4.7].

Soundness. The versions of ReLoC's soundness theorem for closed (Theorem 4.1) and open terms (Theorem 4.7) are stated in Coq as follows:

```
Lemma refines_sound Σ `{relocPreG Σ} e1 e2 τ :
  (∀ `{relocG Σ} Δ, ⊢ REL e1 << e2 : interp τ Δ) →
  ∅ ⊨ e1 ≼ctx e2 : τ.
Lemma refines_sound_open Σ `{relocPreG Σ} Γ e1 e2 τ :
  (∀ `{relocG Σ} Δ, ⊢ {Δ;Γ} ⊨ e1 ≼log e2 : τ) →
  Γ ⊨ e1 ≼ctx e2 : τ.
```

Here, $\Gamma \vDash e_1 \precsim_{ctx} e_2 : \tau$ is the notion of contextual refinement, $\{\Delta;\Gamma\} \vDash e_1 \precsim_{log} e_2 : \tau$ is the refinement judgment lifted to open expressions, and $\vdash P$ expresses that the Iris proposition `P` is derivable.

Example proof: refinement of the bit module. In order to prove the contextual refinement $\emptyset \vDash$ `bit_bool` $\precsim_{ctx}$ `bit_nat` : `bit`τ from Section 4.3.3, it suffices to prove the following:

```
Lemma bit_refinement Δ :
  ⊢ REL bit_bool << bit_nat : interp bitτ Δ.
```

To prove this lemma, we use the relation `R`, which is the same as the one in Section 4.3.3, but wrapped into a semantic type (`lrel`) to ensure it is persistent:

```
Definition R : lrel Σ := LRel (λ v1 v2,
  (⌜v1 = #true⌝ ∧ ⌜v2 = #1⌝) ∨ (⌜v1 = #false⌝ ∧ ⌜v2 = #0⌝)).
```

Using the relation `R`, a Coq proof of the desired refinement is as follows:

```
Lemma bit_refinement Δ :
  ⊢ REL bit_bool << bit_nat : interp bitτ Δ.
Proof.
  unfold bitτ; simpl.
  (* apply REL-PACK *)
  iApply (refines_exists R).
  (* repeatedly apply REL-PAIR *)
  progress repeat iApply refines_pair.
  - (* apply REL-RETURN and solve the goal *)
    rel_values.
  - (* ... *)
Qed.
```

Finally, we combine `bit_refinement` with the soundness theorem to get a closed proof of contextual refinement:

```
Theorem bit_ctx_refinement : ∅ ⊨ bit_bool ≼ctx bit_nat : bitτ
Proof. auto using (refines_sound relocΣ), bit_refinement. Qed.
```

It is important to emphasize that the contextual refinements, which we obtain in theorems like `bit_ctx_refinement` above, are closed propositions in Coq. The statement (the type) of `bit_ctx_refinement` does not refer to ReLoC or Iris. This illustrates that

```
------------------*
P -* (P -* Q) -* Q
```

(a) Before executing any tactics.

```
"H1" : P
"H2" : P -* Q
------------------*
Q
```

(b) After `iIntros "H1 H2".`

Figure 4.16: Interactive proof of lemma `example` in IPM.

the only parts of the trusted code base of our development are the notions that are involved in the definition of contextual refinement, *i.e.*, the operational semantics and typing of contexts.

4.8.3 Tactic support for interactive proofs

To prove refinement judgments, like the bit refinement

```
REL bit_bool << bit_nat : interp bitτ Δ
```

from the previous section, we can repeatedly apply the Iris lemmas corresponding to the ReLoC proof rules. However, doing so directly quickly becomes unwieldy, as the user has to manually provide the resources (like the precondition `l` $\mapsto_s\{$`q`$\}$ `v` of `refines_load_r`), and manually select the evaluation context `K`. For better usability we provide tactic support for symbolic execution.

Interactive separation logic proofs. To explain the tactics for ReLoC that we have defined, let us first look at the general tactic support in Iris. The Iris Proof Mode (IPM) [KTB17] and its successor MoSeL [Kre+18] allow us to carry out separation logic proofs interactively, in the style of regular tactic-based proofs in Coq. IPM provides a convenient representation of sequents for separation logic and tactics for manipulating them, allowing for interactive proof development in the style of regular proofs in Coq. To illustrate this, consider the following separation logic tautology:

```
Lemma example (P Q : iProp Σ) : P -* (P -* Q) -* Q.
Proof. iIntros "H1 H2". iApply ("H2" with "H1"). Qed.
```

The intermediate results can be seen in Figure 4.16. Applying `iIntros "H1 H2".` introduces the hypothesis `P` and `P -* Q` into the IPM context, giving them names `H1` and `H2`, respectively. Then, `iApply ("H2" with "H1").` applies the separating implication `P -* Q` to the goal, using the hypothesis `H1 : P` as the assumption.

Symbolic execution tactics. In addition to tactics like `iIntros` and `iApply`, IPM provides tactic for symbolic execution in weakest preconditions. We built similar tactics on top of IPM for symbolic execution in refinement judgments. To demonstrate that, consider the following example:

```
"Hl" : l ↦ₛ #0
-----------------------------*
REL #2 << (!#l + #2) : lrel_int
```

(a) Before applying `rel_load_r`.

```
"Hl" : l ↦ₛ #0
-----------------------------*
REL #2 << (#0 + #2) : lrel_int
```

(b) After applying `rel_load_r`.

```
"Hl" : l ↦ₛ #0
-----------------------------*
REL #2 << #(0 + 2) : lrel_int
```

(c) After applying `rel_pures_r`.

Figure 4.17: Interactive refinement proof of lemma `example_load` in ReLoC.

```
Lemma example_load l :
  l ↦ₛ #0 -∗ REL #2 << (!#l + #2) : lrel_int.
Proof. iIntros "Hl". rel_load_r. rel_pures_r. rel_values. Qed.
```

The results of `rel_load_r` and `rel_pures_r` can be seen in Figure 4.17. The tactic `rel_load_r` symbolically executes the dereferencing operation, and the tactic `rel_pures_r` symbolically executes as many pure reduction steps as possible. The tactic `rel_values` finishes the goal since both sides are values. Similarly, we built tactics for all other language connectives (both on the left- and right-hand side). The tactics were developed in a similar way to the weakest-precondition tactics from IPM, and we refer the reader to [KTB17] for details.

4.8.4 Overview of the source code

The Coq mechanization contains around 10300 lines of code, of which approximately 1. 1315 lines for mechanization of the model of ReLoC (Section 4.7), including the adequacy theorem (Theorem 4.12), and the primitive and derived rules (Section 4.4); 2. 1200 lines for the tactics (Section 4.8.3); 3. 1450 lines for the mechanization of the type system (Section 4.2), and the soundness theorem for open term (Theorem 4.7); 4. 6050 lines for the examples and case studies (including the case studies we describe in the upcoming Sections 4.10.1 and 4.10.2); 5. and 140 lines for tests (mainly regression tests for the tactics).

4.9 Related work

We described some of the most closely related work in the introduction (Section 4.1), we now discuss other related work on logical relations models, relational logics, atomic specifications, speculative reasoning, and linearizability.

Logical relations models. Logical relations models over denotational and operational semantics have an extensive history. To cover advanced programming language features such as recursive types and higher-order references, logical relations with step-indexing have been introduced [AAV02; Ahm04; ADR09; Bir+11]. Step-indexing has shown to be very effective by a large body of work on step-indexed logical relations models, *e.g.*, [NDR11; HD11; BST12; ÇPG16; RG18]. However, in these papers step-indices appear explicitly in the definition of the logical relations model and the proofs about it. In contrast in this paper we have used the "logical approach" to step-indexed logical relations. This approach, pioneered by Dreyer *et al.*in the LSLR logic [DAB09], hides step-indices by abstracting and internalizing them in a logic using the later modality ($\triangleright$) [App+07]. Dreyer *et al.*used this approach to construct a binary logical relations model for System F with recursive types [DAB09], and later extended the approach as part of the LADR logic to cover existential types and references [Dre+10].

The logical approach to logical relations was further refined by Turon *et al.* [Tur+13; TDB13], culminating in the CaReSL logic, who showed how Hoare triples and ghost thread pools can be used to define a binary logical relation for fine-grained concurrency. Subsequently, a version of this binary logical relation was defined and mechanized in Iris by Krebbers *et al.* [KTB17] and Timany [Tim18]. However, in these papers, logical refinement judgments are meta-logical statements, and because of that, there are no high-level proof rules for establishing and combining refinements. Instead, to prove a refinement judgment, the user of the logic had to unfold the definition of the refinement judgment, and reason directly in CaReSL or Iris. In this work we provide a generalization that makes refinement judgments first-class logical statements, which is crucial to reason abstractly about invariants and formulate atomic specifications. The technical differences are discussed in Section 4.7.3. Thus we really make use of the fact that Iris is a *higher-order* logic—CaReSL is only a second-order logic and it would not be possible to make refinement judgments first-class in CaReSL (indeed Iris is not only based on CaReSL, but just as much on the *higher-order* iCAP logic of Svendsen and Birkedal [SB14]). We also provide a mechanization in Coq with tactical support that supports the same backwards reasoning style that is employed for proving weakest preconditions in Iris [KTB17].

Apart from the directions that we explored in this paper, there has been an abundance of work on logical relations models in Iris. Binary logical relations models in Iris have been used for proving contextual equivalence in the context of Haskell's `ST` monad [Tim+18], first-class per-thread continuations [TB19], and types-and-effect systems [KSB17]. Unary logical relations models in Iris have been used for proving type safety and data-race freedom of the Rust type system [Jun+18a; Dan+20; Jun+21], type safety of session types [Hin+21], type safety of Scala's core calculus DOT [Gia+20], and robust safety [SGD17; Sam+20a]. Logical relations in Iris have also been used for showing other relational properties such as termination-preserving refinement [TJH17], non-interference of concurrent programs [FKB21b], and recovery refinements (refinements in the presence of potential crashes) [Cha+19]. Nearly all of the aforementioned developments have accompanying mechanizations in Coq, and in some of those mechanizations the authors define their own tactics.

They define tactics for either their version of weakest preconditions or for derived operations, but, to the best of our knowledge, they do not define tactics for reasoning about the logical relation directly.

Relational logics. Logics for proving relational properties of programs have a long history, going back to the earlier work of Plotkin and Abadi [PA93]. Since then many relational logics have been developed addressing various applications, *e.g.*, probabilistic properties in security [BGZ09; Bar+12; Bar+13] and cost analysis [Çiç+17; Rad+18]. Here we discuss some more recent work on relational logics that are capable of proving program refinements, with a focus on logics with support for higher-order languages, languages with mutable state, and languages with concurrency.

Earlier work on relational logics targeted programming languages with mutable state, but no concurrency. Relational Hoare logic [Ben04] and Relational Separation logic [Yan07] can be used for reasoning about relational properties for first-order imperative programs, and they have inspired several extensions, for example to probabilistic languages [BGZ09].

Relational Higher Order Logic (RHOL) [Agu+19] is a recent relational higher-order logic for reasoning about relational properties of programs using relational refinement types. The main judgment of RHOL allows one to prove that a relational formula φ holds for two expressions, which do not necessarily have the same type. While it is not directly possible to reason about expressions with different types in ReLoC, we can relate them by using a type variable α and a suitable interpretation of α in the environment Δ. The authors prove soundness of RHOL and show how to embed a number of type systems into it. They provide proofs of various relational properties such as non-interference and relative cost, as provided by the systems they embed into RHOL. In our work we consider only one (family of) relation(s), namely the logical relation for contextual refinement. The programming language considered in RHOL is a pure terminating variant of simply-typed PCF, while we consider a much richer programming language with general references and concurrency.

Liang and Feng developed a relational rely-guarantee style logic [LF13], which can be used to prove refinement for fine-grained concurrent algorithms (including those with helping) but, in contrast to ReLoC, it can only be used to reason about first-order programs.

A relational logic for a sequential class-based language with dynamically allocated objects has been introduced by Banerjee *et al.* [BNN16]. Their relational logic is based on region logic [BNR13], a first-order logic, which is amenable to SMT-based automation. Their relational logic is aimed at proving refinement and non-interference. The approach was further extended in [NBN19] to cover representation independence proofs using per-modules invariants and coupling relations. In contrast, we focus on reasoning about refinements, but also treat concurrent programs and higher-order store, and we provide tool support for tactic-based interactive verification in Coq.

While not a logic in the strict sense, Relational Hoare Type Theory (RHTT) [NBG13] is a dependent type theory for specification and verification of relational properties of higher-order programs with mutable first-order state, capable of expressing

information flow and access control properties. The object programming language of RHTT and the type system itself are shallowly embedded in Coq.

Atomic specifications. To our knowledge, we are the first to study logically atomic specifications in the relational setting. Logically atomic specifications originate in Hoare-style program logics. Jacobs and Piessens [JP11] have originally developed a methodology for specifying logically atomic operations. In their approach, specifications are parameterized by auxiliary code that is performed at the linearization point. This approach was refined to what we refer to as HOCAP-style specifications, originally introduced in the context of the eponymous logic [SBP13], where the role of auxiliary code is filled by *view shifts* [Din+13], which in this paper are given by Iris's update modality $\Rightarrow\!\!\!\!\!\!\times$ (Section 4.5.4). Compared to the original Jacobs-Piessens approach, in HOCAP-style specifications, the physical state that a logically atomic function operates on is hidden behind an abstract predicate. Furthermore, HOCAP-style specifications can also be formulated for non-logically atomic operations, as we have seen in Section 4.5.4.4. The HOCAP-style specifications were later adopted in the iCAP logic [SB14] and Iris logic [BB20, Chapter 11].

Because Jacobs-Piessens and HOCAP-style specifications require parameterizing the (ghost) functions that are executed at the linearization points, such specifications are often referred to as higher-order. As an alternative to this higher-order approach, da Rocha Pinto *et al.*have introduced the notion of logically atomic triples in their program logic TaDA [RDG14; Roc17]. Logically atomic triples are a first-order construct, built in as a primitive construct into the logic, which can be used to specify the atomic updates that a program performs. The atomic triples can be systematically composed in the style of Hoare logic. A more detailed comparison between the first-order and higher-order approach is given in [DRG18]. TaDA-style logically atomic triples were adapted for Iris by Jung *et al.* [Jun+15; Jun+20]. Specifically, they are encoded as derived constructs, using the Jacobs-Piessens approach, that satisfy the TaDA-style rules.

Speculative reasoning. To facilitate speculative reasoning, we employ the mechanism for prophecy variables recently introduced in Iris [Jun+20]. Prophecy variables were first introduced by Abadi and Lamport [AL91] for the purpose of proving refinements of state machines. The idea to use prophecy variables in program logic originates in the rely-guarantee style logic of Vafeiadis [Vaf08], although his treatment of prophecy variables is informal, and he appeals to Abadi and Lamport [AL91] for soundness.

Prophecy variables are not the only tool for carrying out speculative reasoning. Both CaReSL [TDB13] and extended LRG [LF13] are program logics capable of proving refinements of programs with future-dependent linearization points. Both employ, albeit in different forms, a mechanism for recording multiple potential logical states of the program. These multiple states can then be coalesced into a single one, once the linearization point is determined, and that resulting state is used for establishing the refinement.

Other approaches [Khy+17; Del+17] for proving linearizability of algorithms with future-dependent linearization points use Hoare logics with auxiliary state to track the abstract history of a program as a partial order. The crucial property is that all total extensions of the partial order result in valid linear histories of the program.

Other work on linearizability. One of the main application of ReLoC is to prove linearizability of concurrent algorithms, by reducing it to contextual refinements. Proving linearizability has a long history, and the program logic based approach is not the only one. Other methods include automated model checking based solutions [Liu+09; VYY09; Čer+10; Bur+10] and static analysis, in particular shape analysis, [Ami+07; Ber+08; Vaf09]. The model checking approaches in question do not prove linearizability, but automatically *check* execution traces for linearizability, bounding the heap or the number of threads. Indeed, model checking approaches are designed to find bugs in a "push-button" fashion and can generate counterexample traces. Approaches based on static analysis are usually sound even for unbounded heaps and threads, but limited to first-order programs.

4.10 Discussion and conclusion

In this paper we have presented ReLoC—the first mechanized relational logic for proving refinements of fine-grained concurrent higher-order programs. We have demonstrated that ReLoC is expressive enough to formally prove contextual refinements of concurrent programs in a modular way, by employing relational specifications of programs. Moreover, the mechanization of ReLoC in Coq allows us to carry out tactic-based interactive proofs in an intuitive way, by using ReLoC's type-directed structural rules and symbolic execution rules, coupled with the powerful mechanisms from Iris, such as invariants, ghost state, and prophecy variables.

In the remainder of this paper we discuss other case studies that we have mechanized in ReLoC (Section 4.10.1), discuss the "escape hatch" of ReLoC (Section 4.10.2) for verifying programs that cannot be handled by ReLoC, and outline some directions for future work (Section 4.10.3).

4.10.1 Other examples and case studies

In addition to the examples that we have presented in the paper, we have mechanized a number of examples from the literature on logical relations in ReLoC in Coq. Below we give a short summary of those examples.

- Linearizability of the Treiber stack [Tre86];
- Refinement of higher-order cell objects from [KW06; ADR09];
- Refinement of a symbol lookup table and a name generation module from [ADR09];
- Many equivalences from [DNB12], adapted for the concurrent setting, including variations of the "awkward example" from [PS98], and the "higher-order profiling" example modified to use the atomic increment function inc_i;

- Equivalence between different ways of defining the fixed point combinators;
- Equivalence between late-choice and early-choice examples from [Tur+13];
- Algebraic laws for the parallel composition operation and its interaction with non-deterministic choice and sequential composition, inspired by the work on Concurrent Kleene Algebra [Hoa+11];
- Linearizability of the Michael-Scott queue [MS96], mechanized by Simon Friis Vindum and Lars Birkedal [VB21].

4.10.2 The "escape hatch"

The rules of ReLoC are sound, but not complete. In particular, there are some examples that cannot be verified in ReLoC completely. One class of such examples that we know of, are refinements of fine-grained concurrent data structures with *external* linearization points (as opposed to fixed linearization points or future-dependent linearization points; see [DD15] for a survey outlining the differences). Such external linearization points are present, for example, in algorithms that use *helping* or *work-stealing*. Fortunately, ReLoC's model on top of Iris provides an "escape hatch" that still allows us to verify some data structures with helping.

In the appendix [FKB21a] we consider an example of such a data-structure: a fine-grained concurrent stack with helping, a simplified version of the elimination-backoff stack from [HSY04]. We prove that this stack with helping refines a coarse-grained stack (thus showing that the stack with helping is linearizable). The stack with helping is interesting because two threads that perform a push and pop operation concurrently can *eliminate* each other, by exchanging data through a side channel, thus reducing the contention for the top node of the stack. To verify this example we make use of ReLoC's "escape hatch"—we unfold the definition of ReLoC's refinement judgment, and perform an explicit proof in terms of ReLoC's model in Iris so we can explicitly manipulate the ghost thread pool. As we demonstrate, the "escape hatch" does not render ReLoC useless for this example: we still use ReLoC's proof rules to carry out the majority of the proof. Only for a small part of the proof we need to work in the model. This is achieved by encapsulating the *elimination* mechanism of the stack, for which we can provide a logically atomic relational specification that is proved in the model of ReLoC. This specification can then be used through ReLoC's high-level rules to verify the complete data structure without further breaking the abstraction.

4.10.3 Future work

In future work we would like to examine the possibility of a more principled approach to specifying and verifying algorithms with helping, without having to reason in the model of ReLoC. In addition, it would be interesting to explore alternative approaches to speculative reasoning that do not involve prophecy variables. Furthermore, we would like to study applications of ReLoC to type-directed program transformations (for example typed closure conversion [AB08]) and message-passing programs (for example, by integration with the Iris-based Actris logic [HBK20; Hin+21]).

It would also be interesting to see how the ReLoC approach can be used for verifying other kinds of refinements, for example termination-sensitive refinements [TJH17] or refinements in the presence of crashes [Cha+19].

5 SeLoC: a logic for proving non-interference

5.1 Introduction

Non-interference is a form of *information flow control* (IFC) used to express that confidential information cannot be leaked to attackers. To establish non-interference of modern programs, it is crucial to develop verification techniques that support challenging programming paradigms and programming constructs such as concurrency. Furthermore, to scale up these techniques to larger programs, it is important that they are compositional. That is, they should make it possible to establish non-interference of program modules in isolation, without having to consider all possible interference from the environment and other program modules.

Much effort has been put into developing these verification techniques. In terms of expressivity, techniques have been developed that support dynamically allocated references and higher-order functions [PS03; RG18; Zda02], and concurrency [MSS11; SS00; MSE18; Mur+16; SMS20; Kar+18; EM19]. Despite recent advancements, the expressivity of available techniques for non-interference still lags behind the expressivity of techniques for functional correctness, which have seen major breakthroughs since the seminal development of concurrent separation logic [OHe07; Bro07]. There are several reasons for this.

First, a lot of prior work on non-interference focused on type systems and type system-like logics, *e.g.*, [PS03; MSS11; MSE18; EM19; Kar+18]. Such systems provide strong automation (by means of type checking), but lack capabilities to reason about functional correctness, and are thus inherently restrictive in the kind of programs they can verify. For example, it may be the case that the confidentiality of the contents of a reference depends on runtime information instead of solely static information (this is called *value-dependent classification* [ZM07; Mur+16; NBG13; LC15; GTA19]).

Second, proving non-interference is harder than proving functional correctness. While functional correctness is a property about each single run of a program, non-interference is stated in terms of multiple runs of the same program. One has to show that for different values of confidential inputs, the attacker cannot observe a different behavior.

To overcome the aforementioned shortcomings, we take a new approach that combines program logics and type systems: we present a concurrent separation logic for non-interference on top of which we build a type system for non-interference.

Program modules whose non-interference relies on functional correctness (and thus cannot be type checked) can be assigned a type through a manual proof in our separation logic. This combination of separation logic and type checking makes it possible compositionally to establish non-interference of programs that consist of untyped and typed parts.

Although ideas from concurrent separation logic have been employed in the context of non-interference before [Kar+18; EM19], we believe that in the context of non-interference the combination of typing and separation logic is new. Moreover, our approach provides a number of other advantages compared to prior work:

- Our separation logic supports *fine-grained concurrency*. That is, it can verify programs that use low-level atomic operations like compare-and-set to implement lock-free concurrent data structures and high-level synchronization mechanisms such as locks/mutexes. In prior work, such mechanisms were taken to be language primitives.
- Our separation logic is *higher-order*, making it possible to assign very general specifications to program modules.
- Our separation logic is *relational*, making it possible to reason about multiple runs of a program with different values for confidential inputs.
- Our separation logic provides a powerful *invariant* mechanism to describe protocols on the shared state, making it possible to reason about sophisticated forms of sharing, as in value-dependent classifications.

In order to build our logic we make use of the Iris framework for concurrent separation logic [Jun+15; Jun+16; Kre+17; Jun+18b], which provides basic building blocks, including the invariant mechanism. To combine typing and separation logic, we follow recent work on *logical relations* in Iris [KTB17; KSB17; Tim+18; FKB18; Jun+18a; Jun+21], but apply it to non-interference instead of functional correctness or contextual refinement.

Contributions. We introduce **SeLoC**, the first separation logic for non-interference that combines typing and manual proof.

- We present a number of challenging examples that can be verified using SeLoC (Section 5.2).
- SeLoC supports a language with fine-grained concurrency, higher-order functions, and dynamic (higher-order) references (Section 5.3.1). SeLoC is sound w.r.t. a standard timing-sensitive notion of non-interference—*strong low-bisimulations*—by Sabelfeld and Sands [SS00] (Section 5.3.2).
- To verify challenging programs, SeLoC features a relational version of weakest preconditions, which integrates seamlessly with the powerful mechanism for invariants and protocols of the Iris framework (Section 5.4).
- Using the technique of logical relations, we build a type system on top of SeLoC. By building a type system on top of separation logic, we can establish non-interference of programs that consist of typed and untyped parts (Section 5.5).
- To compose proofs of program modules that cannot be type checked (because their interface relies on functional correctness), we show how to express modular separation logic specifications for non-interfence in SeLoC (Section 5.6).

- We prove soundness of SeLoC by constructing a bisimulation out of a separation logic proof (Section 5.7).
- We have mechanized SeLoC, its type system, its soundness proof, and all examples in the paper and appendix, in Coq (Section 5.8). The mechanization can be found online at [FKB20a].

In Section 5.9 we discuss some more specialized topics and features of SeLoC, such as absence of sensitivity labels on reference types and branching on high-sensitivity data. In Section 5.11 we discuss related work and we conclude in Section 5.12.

5.2 Motivating examples

Before we proceed with the formal development of the paper in Section 5.3, we present a number of challenging programs to demonstrate the expressivity of SeLoC.

5.2.1 Modularity and data structures

To guarantee non-interference, one should prove that *high-sensitivity* (*i.e.*, confidential) information cannot leak via *low-sensitivity* (*i.e.*, publicly observable) outputs. Apart from such explicit leaks, one has to prove the absence of implicit leaks that arise from the timing behavior of the program. To avoid timing leaks, Agat and Sands [AS01] outlined the "worst-case principle": a non-interfering algorithm operating on high-sensitivity data should have the same best-case and worst-case execution time. We apply this design principle to a set data structure that stores high-sensitivity elements. The implementation can be type checked using our approach, automatically providing a proof of timing-sensitive non-interference.

To encapsulate the internal set representation, we first present the interface of our data structure. This interface is given using *closures* (*i.e.*, higher-order functions):[1]

$$\mathsf{val\ new\_set} : \mathsf{unit} \to \begin{Bmatrix} \mathsf{lookup} : \mathsf{int}^{\mathbf{H}} \to \mathsf{bool}^{\mathbf{H}}; \\ \mathsf{insert} \ \ : \mathsf{int}^{\mathbf{H}} \to \mathsf{unit} \end{Bmatrix}$$

The function new_set allocates an empty set, and returns a record with functions that operate on the set. The function lookup takes a high-sensitivity integer—typed as $\mathsf{int}^{\mathbf{H}}$, where $\mathbf{H}$ refers to the *high*-sensitivity of the data— and returns a high-sensitivity Boolean—typed as $\mathsf{bool}^{\mathbf{H}}$—that signifies whether the argument is in the set or not. The function insert takes a high-sensitivity integer, and adds it to the set.

Figure 5.1 shows an implementation of our set interface using a sorted dynamic array[2] store in the variable *arr*. To make the data structure thread-safe, the operations are protected by the lock *lk*.

To implement the function lookup, we make use of binary search—but with a twist to avoid timing leaks. An ordinary version of binary search would terminate once it has found the element, making it possible to observe if the element is in the set

[1] When using modules or classes, the same kind of considerations apply.

[2] The full implementation, including the array operations, can be found in the Coq mechanization.

```
let new_set () =
    let k = ref(1) in
    let arr = ref(new_array 1 None) in
    let lk = newlock () in
    ⎧ lookup x = acquire lk;                                                  ⎫
    ⎪     let r = lookup_loop (!arr) (!k) 0 (cap (!k)) x false in             ⎪
    ⎪     release lk; r                                                       ⎪
    ⎨ insert x = acquire lk;                                                  ⎬
    ⎪    insert_loop arr k 0 x;                                               ⎪
    ⎪    release lk                                                           ⎪
    ⎩                                                                         ⎭
let rec lookup_loop a k l r x is_found =
    if k = 0 then is_found else
    let i = (l + r)/2 in
    let e = array_get a i in
    let lr_1 = (i + 1, r) in let lr_2 = (l, i − 1) in
    let (l, r) = if (e < x) then lr_1 else lr_2 in
    lookup_loop a (k − 1) l r x (is_found ∨ (e = x))
let rec  insert_loop arr k i x =
    if i ≥ cap (!k)
    then k ← !k + 1;
         resize_array arr (cap (!k));
         array_set !arr i x
    else match array_get !arr i with
        | None → array_set !arr i x
        | Some(v) → let xv = (x, v) in let vx = (v, x) in
                    let (p_1, p_2) = if (x ≤ v) then xv else vx in
                    array_set !arr i p_1;
                    insert_lookup arr k (i + 1) p_2
```

Figure 5.1: Implementation of a set using the "worst-case principle".

via timing. Our implementation ensures that lookup takes the same time regardless of whether the element is in the set. To achieve that, we represent the set using an array whose size n satisfies $\mathsf{cap}(k) = n$, for some k:

$$\mathsf{cap}(0) = 0 \qquad\qquad \mathsf{cap}(k+1) = 1 + 2 \cdot \mathsf{cap}(k)$$

This guarantees that the array can be recursively partitioned into two sub-arrays of the same size and a pivot element in the middle. If the number of actual elements in the set is less than $\mathsf{cap}(k)$, the array is padded with a dummy element.[3]

If at some iteration of lookup_loop we find that the element x is present in the array, we make note of that fact but still continue with the recursion until the array is no longer splittable. Thus, the function lookup is always executed with k levels of recursion for an array of size $\mathsf{cap}(k)$. In the implementation of lookup_loop we pass the parameter k and decrease it on every recursive call.

The function insert traverses the whole array and is thus always executed with $\mathsf{cap}(k)$ levels of recursion. If the array is full, then it is dynamically resized to the size $\mathsf{cap}(k+1)$. In summary, both lookup and insert operations employ a low-sensitivity termination condition.

We use our type system (described in Section 5.5) to type check the implementation against the interface. Of special note here is the type checking of the if branching. In the implementation of lookup_loop and insert_loop we branch on high-sensitivity data. Notably, in lookup_loop we compare the argument x with the pivot e (both are high-sensitivity integers), and descend into one of the partitions of the array depending on this comparison. Branching on high-sensitivity data is not secure in general, but in this case the branching is secure. This is because both branches simply return variables (lr_1 and lr_2), *i.e.*, they do not perform any computations, and thus do not leak information about the high-sensitivity condition via timing.[4]

5.2.2 Typing via manual proof

The example in the previous section made use of various operations on arrays: array_make, array_get, and array_set. When reasoning about the set data structure, we assumed that these array operations are safe and secure, *i.e.*, when one tries to access an out-of-bounds index, array_get returns a dummy element, instead of reading arbitrary memory.

The programming language that we consider does not have safe arrays as a primitive construct. Instead, safe arrays are implemented as a library: an array is stored together with its length, and the unsafe operations are protected by dynamic checks. Naturally, such operations cannot be type checked in an ML-style type system, because their safety and security depends on functional correctness. However, one of the core features of our approach is that such functions can be assigned types through a manual separation logic proof in SeLoC. Such a manual proof takes functional properties (e.g., that the index is within the array bounds) into account. Once we manually verify that the array library satisfies the desired typing, we can compose it with the type checked example from the previous section to obtain a library that guarantees safety and non-interference for its clients.[5]

[3] For comparisons $<$ and equality $=$ checks we assume that the dummy element **None** is the greatest element and that it is not equal to any actual element in the array, which are of the form $\mathbf{Some}(x)$.

[4] In a low-level language like C the branching can be written using arithmetic.

[5] The proof of the array library and its integration in the type checking of the set data structure can be found in the Coq mechanization.

$$
\begin{aligned}
\textsf{let rec thread1 } \mathit{out}\ r = (&\textsf{if } \neg\,!\,r.\mathit{is\_classified}\\
&\quad \textsf{then } \mathit{out} \leftarrow\, !\,r.\mathit{data}\ \ \textsf{else}\ \ ());\\
&\textsf{thread1 } \mathit{out}\ r\\
\textsf{let thread2 } r = {}& r.\mathit{data} \leftarrow 0;\\
&r.\mathit{is\_classified} \leftarrow \textbf{false}\\
\textsf{let prog } \mathit{out}\ \mathit{secret} = \textsf{let } r = {}&\left\{\begin{array}{l}\mathit{data} = \textsf{ref}(\mathit{secret});\\ \mathit{is\_classified} = \textsf{ref}(\textbf{true})\end{array}\right\}\\
&\textsf{in } (\textsf{thread1 } \mathit{out}\ r) \,\|\, (\textsf{thread2 } r)
\end{aligned}
$$

Figure 5.2: Lock-free value-dependent classification.

The combination of typing and manual proof is important for compositionality and scalability: challenging library code whose security relies on functional correctness (such as the library for safe arrays) can be manually verified using separation logic, and then used to automatically type check other libraries (such as the set data structure).

5.2.3 Fine-grained concurrency

As shown in Section 5.2.2, the ability to fall back to a manual proof is useful to assign types to code that uses operations such as array indexing whose safety and security relies on functional correctness. This ability becomes even more pertinent for (fine-grained) concurrent programs, where the safety and security can depend on specific protocols on data that is shared between threads.

To demonstrate the application to concurrency, we consider the program prog in Figure 5.2, which is a lock-free version of a similar lock-based program in [EM19]. The program runs two threads in parallel, both of which operate on a reference r.data. The data in this reference has a *value-dependent classification*: the value of the flag $r.\mathit{is\_classified}$ determines the sensitivity of r.data. If the flag r.*is_classified* is set to **false**, then the data stored in r.data is classified with low-sensitivity, and if it is set to **true**, the the data is classified with high-sensitivity. The record r initially contains high-sensitivity data from the integer variable secret. The first thread thread1 checks if the record r is classified (*i.e.*, the flag $r.\mathit{is\_classified}$ is **true**), and if it is not, it leaks the data r.data to an attacker-observable channel *out*. The second thread thread2 overwrites the data stored in r and resets the classification flag.

Due to the precise interplay of the two threads, the program prog is secure, in the sense that it does not leak the data secret onto the public channel *out*. Since our example does not use locks, there are more possible interleavings than in the original example in [EM19], and consequently there are more things that could potentially go wrong in thread1:

1. the data r.data can still be classified even if the bit $r.\mathit{is\_classified}$ is set to **false**;

2. the classification of the data stored in r might change between reading the field *is_classified* and reading the actual data from the field data.

Notice that if we replace the second thread by the expression below, where the two operations in thread2 have been swapped, then we would violate the first condition:

$$\mathsf{let}\ \mathsf{thread2}_{\mathsf{bad}}\ r = r.\mathit{is\_classified} \leftarrow \mathbf{false};\ r.\mathsf{data} \leftarrow 0$$

To verify that both of these situations cannot occur, we have to establish a *protocol* on accessing the record r. The protocol should ensure that at the moment of reading $r.\mathit{is\_classified}$ the data $r.\mathsf{data}$ has the correct classification (ruling out situation 1). The protocol should also ensure a form of *monotonicity*: whenever the classification becomes low (*i.e.*, $r.\mathit{is\_classified}$ becomes **false**), $r.\mathsf{data}$ is not going to contain high-sensitivity data for the rest of the program (ruling out situation 2).

The security of thread1, and the whole program, depends on the specific protocol attached to the record r and that the protocol is followed by all the components that operate on it. In particular, for this example the security depends on the fact that classification only changes in a *monotone* way. We outline the proof of safety and security of this example in Section 5.4.4.

5.2.4 Higher-order functions and dynamic references

As shown in this section, higher-order functions are useful for modularity—they can be used to model interfaces. However, since they can operate on encapsulated state, they are difficult to reason about. Fortunately, SeLoC's protocol mechanism is also applicable to proving non-interference of functions with encapsulated state. Consider the program awk, a variation of the "awkward example" of Pitts and Stark [PS98]:

$$\mathsf{let}\ \mathsf{awk}\ v = \mathsf{let}\ x = \mathsf{ref}(v)\ \mathsf{in}\ \lambda f.\ x \leftarrow 1; f(); !x$$

When applied to a value v, the program awk returns a closure that, when invoked, always returns low-sensitivity data from the reference x, even if the original value v has high-sensitivity. Intuitively, awk v returns a closure that does not leak any data, even if the original value v passed to awk had high-sensitivity. The lack of leaks crucially relies on the following facts:

- the reference x is allocated in, and remains local to, the closure, it cannot be accessed without invoking the closure;
- the reference x can be updated only in a monotone way: once the original value v gets overwritten with 1, the reference x never holds a high-sensitivity value again.

To see why second condition is important, consider $\mathsf{awk}_{\mathsf{bad}}$, which violates the monotonicity, and is thus not secure:

$$\mathsf{let}\ \mathsf{awk}_{\mathsf{bad}}\ v = \mathsf{let}\ x = \mathsf{ref}(v)\ \mathsf{in}\ \lambda f.\ x \leftarrow v; x \leftarrow 1; f(); !x$$

Let $h = \mathsf{awk}_{\mathsf{bad}}\ v$ for a high-sensitivity value v. Now, when running $h\ (\lambda x.\ \mathsf{fork}\ \{h(\mathsf{id})\})$, an attacker could influence the scheduler so that the first dereference $!x$ happens just after the assignment $x \leftarrow v$ in the forked-off thread, causing v to leak.

Pitts and Stark studied the "awkward example" to motivate the difficulties of reasoning about higher-order functions and state. They were interested in contextual equivalence, but as we can see, similar considerations apply to non-interference.

5.3 Preliminaries

In this section we describe the programming language that we consider in this paper (Section 5.3.1), and the non-interference property that SeLoC establishes (Section 5.3.2).

5.3.1 Object language semantics

SeLoC is defined over the programming language HeapLang, which we described in Chapter 2. We briefly recall the syntax and semantics of HeapLang. It is an an ML-like with higher-order mutable references, recursion, and fork-based concurrency. Its values and expressions are:

$$\begin{aligned} v \in \mathit{Val} &::= \mathsf{rec}\ f\ x = e \mid (v_1, v_2) \mid \mathbf{true} \mid \mathbf{false} \mid \ldots \\ e, t \in \mathit{Expr} &::= x \mid \mathsf{rec}\ f\ x = e \mid e_1(e_2) \mid \mathsf{fork}\ \{e\} \\ &\quad \mid \mathsf{ref}(e) \mid\ !\, e \mid e_1 \leftarrow e_2 \mid \mathsf{CAS}(e_1, e_2, e_3) \mid \ldots \end{aligned}$$

We omit the usual operations on pairs, sums, and integers. The *atomic* compare-and-set operation $\mathsf{CAS}(e_1, e_2, e_3)$ checks if the value stored at the location e_1 is equal to e_2, and, if so, sets the value at e_1 to e_3. The $\mathsf{fork}\ \{e\}$ construct creates a new thread, which will execute the expression e. The construct $\mathsf{rec}\ f\ x = e$ is a recursive λ-function, whose body e can refer to the function f itself and the argument x.

We use the following syntactic sugar: $(\lambda x.\, e) \triangleq (\mathsf{rec}\ \_\ x = e)$, $(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \triangleq ((\lambda x.\, e_2)\, e_1)$, and $(e_1; e_2) \triangleq (\mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2)$, where we use _ as an anonymous binder, in place of a variable name. HeapLang has no primitive syntax for records, so they are modeled using pairs. Arrays are omitted in the paper, but they are present in the Coq mechanization.

HeapLang features dynamic thread creation, so we can implement the parallel composition operation using fork:

$$\begin{aligned} \mathsf{let\ rec\ join}\ x = \mathsf{match}\ !x\ \mathsf{with}\ \mathbf{Some}(v) &\to v \\ \mid \mathbf{None} &\to \mathsf{join}\ x \\ \mathsf{let\ par}(f_1, f_2) = \mathsf{let}\ x = \mathsf{ref}(\mathbf{None})\ \mathsf{in}& \\ \mathsf{fork}\ \{x \leftarrow \mathbf{Some}(f_1())\}& \\ \mathsf{let}\ v_2 = f_2()\ \mathsf{in}\ (\mathsf{join}\ x, v_2)& \\ e_1 \parallel e_2 \triangleq \mathsf{par}(\lambda\_.\, e_1, \lambda\_.\, e_2)& \end{aligned}$$

The operational semantics of HeapLang is split into three reduction relations: thread-local head reduction $\to_{\mathsf{h}}$, thread-local reduction $\to_{\mathsf{t}}$, and thread-pool reduction $\to_{\mathsf{tp}}$. The thread-local head reduction is of the form $(e_1, \sigma_1) \to_{\mathsf{h}} (e_2, \sigma_2)$, where e_i is an

expression, and σ_i is a heap, *i.e.*, a finite map from locations to values ($State \triangleq Loc \xrightarrow{\text{fin}} Val$).

The thread-local head reduction is lifted to the thread-local reduction using *call-by-value evaluation contexts*:

$$K \in ECtx ::= [\bullet] \mid K(v_2) \mid e_1(K) \mid \text{if } K \text{ then } e_1 \text{ else } e_2 \mid \ldots$$

The thread-local reduction is of the form $(e_1, \sigma_1) \to_{\mathsf{t}} (\vec{e}_2, \sigma_2)$. The second component contains a list $\vec{e}_2$ of expressions to accommodate forked-off threads as in STEP-FORK:

$$\frac{\text{STEP-LIFT}\quad (e_1, \sigma_1) \to_{\mathsf{h}} (e_2, \sigma_2)}{(K[\,e_1\,], \sigma_1) \to_{\mathsf{t}} (K[\,e_2\,], \sigma_2)} \qquad \frac{\text{STEP-FORK}\quad \vec{e} = K[\,()\,]\; e}{(K[\,\text{fork}\;\{e\}\,], \sigma) \to_{\mathsf{t}} (\vec{e}, \sigma)}$$

The thread-pool reduction $\to_{\mathsf{tp}}$ is defined by lifting the thread-local reduction to *configurations* $(\vec{e}, \sigma)$. Here, $\vec{e}$ contains all threads, including values for the threads that have terminated. In the definition of $\to_{\mathsf{tp}}$ we non-deterministically select an expression to take a thread-local step:

$$\frac{(e_i, \sigma_1) \to_{\mathsf{t}} (e_i' \vec{e}, \sigma_2)}{(e_0 \ldots e_i \ldots e_n, \sigma_1) \to_{\mathsf{tp}} (e_0 \ldots e_i' \ldots e_n \vec{e}, \sigma_2)}$$

5.3.2 Strong low-bisimulations

To state the soundness theorem of SeLoC in Section 5.4.3, we adapt a timing-sensitive notion of non-interference for concurrent programs known as *strong low-simulations* on configurations by Sabelfeld and Sands [SS00]. To define this notion, we first fix a set $\mathfrak{L} \subseteq Loc$ of *output locations*, which we assume to be low-sensitivity observable locations. For simplicity, we require these locations to contain integers.

Definition 5.1. Heaps σ_1 and σ_2 are *low-equivalent for output locations* $\mathfrak{L} \subseteq Loc$, denoted as $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$, if they are defined and agree on all the $\mathfrak{L}$-locations, *i.e.*,

$$\forall \ell \in \mathfrak{L}.\ (\sigma_1(\ell) = \sigma_2(\ell)) \wedge (\sigma_1(\ell) \neq \bot) \wedge \sigma_1(\ell) \in \mathbb{Z}.$$

Definition 5.2. A *a strong low-bisimulation* is a partial equivalence (*i.e.*, symmetric and transitive) relation $\mathcal{R}$ on configurations such that:

1. If $(v\vec{e}, \sigma_1)\ \mathcal{R}\ (w\vec{t}, \sigma_2)$, then $v = w$;
2. If $(\vec{e}, \sigma_1)\ \mathcal{R}\ (\vec{t}, \sigma_2)$, then $|\vec{e}| = |\vec{t}|$ and $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$;
3. If $(e_0 \ldots e_i \ldots e_n, \sigma_1)\ \mathcal{R}\ (t_0 \ldots t_i \ldots t_n, \sigma_2)$ and $(e_i, \sigma_1) \to_{\mathsf{t}} (e_i' \vec{e}, \sigma_1')$, then there exist t_i', $\vec{t}$, and σ_2' such that:
   - $(t_i, \sigma_2) \to_{\mathsf{t}} (t_i' \vec{t}, \sigma_2')$;
   - $(e_0 \ldots e_i' \ldots e_n \vec{e}, \sigma_1')\ \mathcal{R}\ (t_0 \ldots t_i' \ldots t_n \vec{t}, \sigma_2')$.

Notice that the first expression in the thread-pool is the main thread. The first condition in Definition 5.2 thus states that the return values of the main-thread should agree.

To model the input/high-sensitivity data we use free variables. For simplicity we assume that the input data consists of integers. We then arrive at the following top-level definition of security.

Definition 5.3 (Security)**.** An expression e with free variables $\vec{x}$ is *secure* if for any heap σ with $\sigma \sim_{\mathsf{L}} \sigma$, and any sequences of integers $\vec{i}$, $\vec{j}$ with $|\vec{i}| = |\vec{j}| = |\vec{x}|$, there exists a strong low-bisimulation $\mathcal{R}$ such that $(e[\vec{i}/\vec{x}], \sigma)\ \mathcal{R}\ (e[\vec{j}/\vec{x}], \sigma)$.

5.3.3 Non-determinism and non-interference

The semantics presented in Section 5.3.1 is deterministic on the thread-local level, but we can still account for non-determinism arising from a scheduler. Consider the program rand, which uses intrinsic non-determinism of the thread-pool semantics to return either **true** or **false**:

$$\mathsf{let}\ \mathsf{rand}\ () = \mathsf{let}\ x = \mathsf{ref}(\mathbf{true})\ \mathsf{in}\ \ \mathsf{fork}\ \{x \leftarrow \mathbf{false}\}; !\, x$$

This program is secure w.r.t. Definition 5.3 (we will prove this in Section 5.4 using SeLoC).

It is worth pointing out that if we modify the program and insert an additional assignment of a high-sensitivity value h to x, then the resulting program is *not* secure:

$$\begin{aligned}\mathsf{let}\ \mathsf{rand}_{bad}\ () = \mathsf{let}\ x = \mathsf{ref}(\mathbf{true})\ \mathsf{in}&\\ \mathsf{fork}\ \{x \leftarrow h\}; \mathsf{fork}\ \{x \leftarrow \mathbf{false}\}; !\, x&\end{aligned}$$

The program is not secure because an attacker can pick a scheduler that always executes the leaking assignment, or, even simpler, can run the program many times under the uniform scheduler. Because the program is not secure, we cannot prove it in SeLoC. In SeLoC, we would verify each thread separately, and we would not be able to verify the forked-off thread $x \leftarrow h$ (precisely because it makes the non-determinism of assignments to the reference x dangerous).

5.4 Overview of SeLoC

We provide an overview of SeLoC by presenting its proof rules for relational reasoning (Section 5.4.1), its invariant mechanism (Section 5.4.2), its soundness theorem (Section 5.4.3), and finally its protocol mechanism (Section 5.4.4), which we apply to the verification of the program prog from Section 5.2.3. The grammar of SeLoC is:

$$\begin{aligned} P, Q \in \mathit{iProp} ::= {} & \mathsf{True} \mid \mathsf{False} \mid \forall x.\, P \mid \exists x.\, P \mid P * Q \\ & \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto_\theta v \mid \mathsf{awp}_\theta\, e\, \{\Phi\} && (\theta \in \{\mathsf{L}, \mathsf{R}\}) \\ & \mid \mathsf{dwp}_{\mathcal{E}}\, e_1\ \&\ e_2\ \{\Phi\} \\ & \mid \boxed{P}^{\mathcal{N}} \mid \triangleright P \mid \Box P \mid {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \mid \ldots \end{aligned}$$

SeLoC features the standard separation logic connectives like separating conjunction ($*$) and magic wand ($-\!\!*$). Since SeLoC is based on Iris [Jun+15; Jun+16; Kre+17;

Jun+18b], it incorporates all the Iris connectives and modalities, in particular the *later modality* ($\triangleright$) for dealing with recursion, the *persistence modality* ($\Box$) for dealing with shareable resources, and the *invariant connective* ($\boxed{P}^{\mathcal{N}}$) and the *update modality* (${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$) for establishing and relying on protocols. We will not introduce the Iris connectives in detail, but rather explain them on a by-need basis. An interested reader is referred to [Jun+18b; BB20] for further details. Various connectives are annotated with *name spaces* $\mathcal{N} \in \mathit{InvName}$ and *invariant masks* $\mathcal{E} \subseteq \mathit{InvName}$ to handle some bookkeeping. When the mask is omitted, it is assumed to be $\top$, the largest mask. We let $\Rightarrow_{\mathcal{E}}$ denote ${}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}}$. Readers who are unfamiliar with Iris can safely ignore the name spaces and invariant masks.

A selection of proof rules of SeLoC is given in Figure 5.3. Each inference rule $\dfrac{P_1 \ \ldots \ P_n}{Q}$ in this paper should be read as an entailment $P_1 * \ldots * P_n \vdash Q$. In the subsequent sections we explain and motivate the rules of SeLoC.

5.4.1 Relational reasoning

The quintessential connective of SeLoC is the *double weakest precondition* $\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\}$. Intuitively, it expresses that any two runs of e_1 and e_2 are related in a lock-step bisimulation-like way, and that the resulting values of any two terminating runs are related by the *postcondition* $\Phi : \mathit{Val} \to \mathit{Val} \to \mathit{iProp}$. We refer to e_1 (resp. e_2) as the *left-hand side* (resp. the *right-hand side*). The double weakest precondition is defined such that if

$$\forall \vec{i}\,\vec{j} \in \mathbb{Z}.\ \mathsf{dwp}\ e[\vec{i}/\vec{x}]\ \&\ e[\vec{j}/\vec{x}]\ \{v_1\ v_2.\ v_1 = v_2\}$$

is derivable (with $\vec{x}$ the free variables of e), then e is secure. We defer the precise soundness statement to Section 5.4.3.

A selection of rules for double weakest preconditions[6] are given in Figure 5.3. Some of these rules are generalizations of the ordinary weakest precondition rules (*e.g.*, DWP-VAL, DWP-WAND, DWP-FUPD, DWP-BIND). The more interesting rules are the *symbolic execution* rules, which allow executing the programs on both sides in a lock-step fashion. If both sides involve a pure-redex, we can use DWP-PURE. The premises $e \to_{\mathsf{pure}} e'$ denote that e deterministically reduces to e' without any side-effects (*e.g.*, $(\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e\ \mathsf{else}\ t) \to_{\mathsf{pure}} e$). If both sides involve a fork, we can use the rule DWP-FORK, which is a generalization of Iris's fork rule to the relational case. To explain SeLoC's rules for symbolic execution of heap-manipulating expressions, we need to introduce some additional machinery:

- Due to SeLoC's relational nature, there are left- and right-hand side versions of the *points-to connectives* $\ell \mapsto_\theta v$, where $\theta \in \{\mathsf{L}, \mathsf{R}\}$, which denote that the value v of location ℓ in the heap associated with the left-hand side program and the right-hand side program, resp.

[6] Some of the SeLoC rules involve the *later modality* $\triangleright$, which is standard for dealing with recursion and impredicative invariants [Jun+18b, Section 5.5]. The occurrences of $\triangleright$ can be ignored for the purposes of this paper.

$$\textsc{dwp-val}\ \frac{\Phi(v_1, v_2)}{\mathsf{dwp}_{\mathcal{E}}\ v_1\ \&\ v_2\ \{\Phi\}} \qquad \textsc{dwp-wand}\ \frac{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Psi\} \qquad (\forall v_1\ v_2.\ \Psi(v_1, v_2) \mathrel{-\!\!*} \Phi(v_1, v_2))}{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\}}$$

$$\textsc{dwp-fupd}\ \frac{\Rrightarrow_{\mathcal{E}} \mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{v_1\ v_2.\ \Rrightarrow_{\mathcal{E}} \Phi(v_1, v_2)\}}{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\}}$$

$$\textsc{dwp-bind}\ \frac{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \left\{v_1\ v_2.\ \mathsf{dwp}_{\mathcal{E}}\ K_1[\,v_1\,]\ \&\ K_2[\,v_2\,]\ \{\Phi\}\right\}}{\mathsf{dwp}_{\mathcal{E}}\ K_1[\,e_1\,]\ \&\ K_2[\,e_2\,]\ \{\Phi\}}$$

$$\textsc{dwp-pure}\ \frac{e_1 \to_{\mathsf{pure}} e_1' \qquad e_2 \to_{\mathsf{pure}} e_2' \qquad \triangleright\, \mathsf{dwp}_{\mathcal{E}}\ e_1'\ \&\ e_2'\ \{\Phi\}}{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\}}$$

$$\textsc{dwp-fork}\ \frac{\triangleright\, \mathsf{dwp}\ e_1\ \&\ e_2\ \{\mathsf{True}\} \qquad \triangleright \Phi((), ())}{\mathsf{dwp}_{\mathcal{E}}\ (\mathsf{fork}\ \{e_1\})\ \&\ (\mathsf{fork}\ \{e_2\})\ \{\Phi\}}$$

$$\textsc{dwp-awp}\ \frac{\mathsf{awp}_{\mathsf{L}}\ e_1\ \{\Psi_1\} \qquad \mathsf{awp}_{\mathsf{R}}\ e_2\ \{\Psi_2\} \qquad (\forall v_1, v_2.\ (\Psi_1(v_1) * \Psi_2(v_2)) \mathrel{-\!\!*} \triangleright \Phi(v_1, v_2))}{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\}}$$

$$\textsc{awp-store}\ \frac{\ell \mapsto_\theta v_1 \qquad (\ell \mapsto_\theta v_2 \mathrel{-\!\!*} \Phi())}{\mathsf{awp}_\theta\ \ell \leftarrow v_2\ \{\Phi\}} \qquad \textsc{awp-load}\ \frac{\ell \mapsto_\theta v \qquad (\ell \mapsto_\theta v \mathrel{-\!\!*} \Phi(v))}{\mathsf{awp}_\theta\ !\ell\ \{\Phi\}}$$

$$\textsc{awp-alloc}\ \frac{\forall \ell.\ \ell \mapsto_\theta v \mathrel{-\!\!*} \Phi(\ell)}{\mathsf{awp}_\theta\ \mathsf{ref}(v)\ \{\Phi\}} \qquad \textsc{dwp-inv-alloc}\ \frac{P \qquad (\boxed{P}^{\mathcal{N}} \mathrel{-\!\!*} \mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\})}{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\}} \qquad \textsc{inv-dup}\ \frac{\boxed{P}^{\mathcal{N}}}{\boxed{P}^{\mathcal{N}} * \boxed{P}^{\mathcal{N}}}$$

$$\textsc{dwp-inv}\ \frac{\mathsf{atomic}(e_1) \qquad \mathsf{atomic}(e_2) \qquad \mathcal{N} \in \mathcal{E} \qquad \boxed{P}^{\mathcal{N}} \qquad (\triangleright P \mathrel{-\!\!*} \mathsf{dwp}_{\mathcal{E} \setminus \mathcal{N}\uparrow}\ e_1\ \&\ e_2\ \{v_1\ v_2.\ P * \Phi(v_1, v_2)\})}{\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\}}$$

Figure 5.3: A selection of the proof rules of SeLoC.

- To avoid a quadratic explosion in combinations of all possible heap-manipulating expressions on the left- and the right-hand side, SeLoC includes a unary weakest precondition. $\mathsf{awp}_\theta\, e\, \{\Phi\}$ for atomic and fork-free expressions.[7] The rules for unary weakest preconditions (*e.g.*, AWP-STORE, AWP-LOAD, AWP-ALLOC) are similar to those of Iris, but each rule is parameterized by a side $\theta \in \{\mathsf{L}, \mathsf{R}\}$.

The rule DWP-AWP connects dwp and awp_θ. For instance, using DWP-AWP, AWP-STORE, and AWP-LOAD, we can derive:

$$\frac{\ell_1 \mapsto_{\mathsf{L}} v_1 \qquad \ell_2 \mapsto_{\mathsf{R}} v_2 \qquad (\ell_1 \mapsto_{\mathsf{L}} v_1 * \ell_2 \mapsto_{\mathsf{R}} v_2') \mathrel{-\!\!*} \mathsf{dwp}\ v_1\ \&\ ()\ \{\Phi\}}{\mathsf{dwp}\ !\ell_1\ \&\ (\ell_2 \leftarrow v_2')\ \{\Phi\}}$$

5.4.2 Invariants

Let us demonstrate, by means of an example, how to use the symbolic execution rules together with the powerful invariant mechanism of Iris. Recall the rand example from Section 5.3.3. We can use invariants to prove the following:

Proposition 5.4. $\mathsf{dwp}\ \mathsf{rand}()\ \&\ \mathsf{rand}()\ \{v_1\, v_2.\ v_1 = v_2\}$.

Proof. First we use DWP-PURE to symbolically execute a β-reduction. We then use DWP-BIND to "focus" on the ref(**true**) subexpression, leaving us with the goal:

$$\begin{aligned} &\mathsf{dwp}\ \mathsf{ref}(\mathbf{true})\ \&\ \mathsf{ref}(\mathbf{true})\ \{\Phi\} \\ \textbf{where}\ \Phi(\ell_1, \ell_2) \triangleq\ &\mathsf{dwp}\ \mathsf{let}\ x = \ell_1\ \mathsf{in}\ \ldots\ \& \\ &\qquad \mathsf{let}\ x = \ell_2\ \mathsf{in}\ \ldots\ \{v_1\, v_2.\ v_1 = v_2\} \end{aligned}$$

We then symbolically execute the allocation, using DWP-AWP and AWP-ALLOC, obtaining $\ell_1 \mapsto_{\mathsf{L}} \mathbf{true}$ and $\ell_2 \mapsto_{\mathsf{R}} \mathbf{true}$:

$$\begin{aligned} &\ell_1 \mapsto_{\mathsf{L}} \mathbf{true} * \ell_2 \mapsto_{\mathsf{R}} \mathbf{true} \\ &\quad \vdash \mathsf{dwp}\ \mathsf{fork}\ \{\ell_1 \leftarrow \mathbf{false}\}; !\ell_1\ \& \\ &\qquad \mathsf{fork}\ \{\ell_2 \leftarrow \mathbf{false}\}; !\ell_2\ \{v_1\, v_2.\ v_1 = v_2\} \end{aligned}$$

It is tempting to use DWP-FORK; but in both the main thread and the forked-off thread we need $\ell_1 \mapsto_{\mathsf{L}} -$ and $\ell_2 \mapsto_{\mathsf{R}} -$ to symbolically execute the dereference and assignment to ℓ_1 and ℓ_2. To share the points-to connectives between both threads, we put them into an Iris-style invariant.

Iris-style invariants are logical propositions denoted as $\boxed{P}^{\mathcal{N}}$, which express that P holds at all times. Unlike in other logics, Iris-style invariants are not attached to locks. Rather, one can explicitly open an invariant during an atomic step of execution to get access to its contents. To create a new invariant we use the DWP-INV-ALLOC rule, which transfers P into the an invariant $\boxed{P}^{\mathcal{N}}$ with a name space $\mathcal{N} \in \mathit{InvName}$. The transfer of P into an invariant makes it possible to share P between different threads

[7] For an atomic expression e, the proposition $\mathsf{awp}_\theta\, e\, \{\Phi\}$ is essentially equivalent to $\mathsf{wp}_\emptyset\, e\, \{\Phi\}$, but defined for the $\ell \mapsto_\theta v$ points-to connectives instead of the untagged $\ell \mapsto v$.

(using INV-DUP). To access an invariant we use the rule DWP-INV. It allow us to *open* an invariant during an atomic symbolic execution step. The *masks* $\mathcal{E} \subseteq \mathit{InvName}$ on dwp are used to keep track of which invariants have been open. This is done to prevent invariant reentrancy.

Returning to our example, we can use DWP-INV-ALLOC to allocate the invariant $I \triangleq \boxed{\exists b \in \mathbb{B}.\, \ell_1 \mapsto_{\mathsf{L}} b * \ell_2 \mapsto_{\mathsf{R}} b}^{\mathcal{N}}$. This invariant not only allows different threads to access ℓ_1 and ℓ_2 (via INV-DUP), but it also ensures that ℓ_1 and ℓ_2 contain the same Boolean value throughout the execution.

The proof then proceeds as follows. We apply DWP-FORK and get two new goals:

1. $I \vdash \mathsf{dwp}\ \ell_1 \leftarrow \mathbf{false}\ \&\ \ell_2 \leftarrow \mathbf{false}\ \{\mathsf{True}\}$;
2. $I \vdash \mathsf{dwp}\ !\ell_1\ \&\ !\ell_2\ \{v_1\, v_2.\, v_1 = v_2\}$.

The invariant I can be used for proving both goals (INV-DUP). The first goal involves proving that the assignment of **false** to ℓ_1 and ℓ_2 is secure. We verify this via DWP-INV, and temporarily opening the invariant I to obtain $\ell_1 \mapsto_{\mathsf{L}} b$ and $\ell_2 \mapsto_{\mathsf{R}} b$. We then apply DWP-AWP, and symbolically execute the assignment to obtain $\ell_1 \mapsto_{\mathsf{L}} \mathbf{false}$ and $\ell_2 \mapsto_{\mathsf{R}} \mathbf{false}$. At the end of this atomic step, we verify that the invariant I still holds.

The second goal is solved in a similar way. When we dereference ℓ_1 and ℓ_2 we know that they contain the same value because of the invariant I. □

5.4.3 Soundness

We now state SeLoC's soundness theorem, which guarantees that verified programs are actually secure w.r.t. Definition 5.3.

As we have described in Section 5.3.2, we fix a set $\mathfrak{L}$ of output locations that we assume to be observable by the attacker. We require these locations to always contain the same data in both runs of the program. To reflect this in the logic, we use an invariant that owns the observable locations and forces them to contain the same values in both heaps:

$$I_{\mathfrak{L}} \triangleq \mathop{\Large *}_{\ell \in \mathfrak{L}} \boxed{\exists i \in \mathbb{Z}.\, \ell \mapsto_{\mathsf{L}} i * \ell \mapsto_{\mathsf{R}} i}^{\mathcal{N}.(\ell,\ell)}$$

When we verify a program under the invariant $I_{\mathfrak{L}}$, we are forced to interact with the locations in $\mathfrak{L}$ as if they are permanently publicly observable. With this in mind we state the soundness theorem, which we prove in Section 5.7.

Theorem 5.5 (Soundness)**.** Suppose that:

$$I_{\mathfrak{L}} \vdash \forall \vec{i}, \vec{j} \in \mathbb{Z}.\, \mathsf{dwp}\ e[\vec{i}/\vec{x}]\ \&\ e[\vec{j}/\vec{x}]\ \{v_1\, v_2.\, v_1 = v_2\}$$

is derivable, where $\vec{x}$ are the free variables of e, and $\vec{i}$ and $\vec{j}$ are lists of integers with $|\vec{i}| = |\vec{j}| = |\vec{x}|$, then:

- the expression e is secure, and,
- the configuration $(e[\vec{i}/\vec{x}], \sigma)$ is safe (*i.e.*, cannot get stuck) for any list of integers $\vec{i}$, and any heap σ with $\sigma \sim_{\mathfrak{L}} \sigma$.

5.4.4 Protocols

Now that we have seen the basics of Iris-style invariants in SeLoC, let us use the protocol mechanism SeLoC inherits from Iris to verify the example prog from Figure 5.2. We prove the following proposition, which serves as a premise for Theorem 5.5, and therefore implies the security of prog.

Proposition 5.6. For any integers $i_1, i_2 \in \mathbb{Z}$, we have

$$I_{\{out\}} \vdash \mathsf{dwp}\ \mathsf{prog}\ out\ i_1\ \&\ \mathsf{prog}\ out\ i_2\ \{v_1\ v_2.\ v_1 = v_2 = ((),())\}.$$

Proof. We first need a derived rule for parallel composition (which we defined in terms of fork in Section 5.3.1). The parallel composition operation satisfies a binary version of the standard specification in Concurrent Separation Logic [OHe07]:

DWP-PAR

$$\frac{\mathsf{dwp}\ e_1\ \&\ t_1\ \{\Psi_1\} \qquad \mathsf{dwp}\ e_2\ \&\ t_2\ \{\Psi_2\} \qquad \left(\forall v_1, v_2, w_1, w_2.\ (\Psi_1(v_1, w_1) * \Psi_2(v_2, w_2)) \mathrel{-\!\!*} \Phi((v_1, w_1), (v_2, w_2))\right)}{\mathsf{dwp}\ (e_1 \parallel e_2)\ \&\ (t_1 \parallel t_2)\ \{\Phi\}}$$

Second, we need to establish a protocol on the way the values in the record r may evolve. We identify three logical states $\mathsf{State} \triangleq \{\mathsf{Classified}, \mathsf{Intermediate}, \mathsf{Declassified}\}$ the record r can be in; visualized in Figure 5.4a:

1. Classified, if the data stored in the record is classified, and *r.is_classified* points to **true**;
2. Intermediate, when the data stored in the record is not classified anymore, but *r.is_classified* still points to **true**;
3. Declassified, when the data stored in the record is not classified and *r.is_classified* points to **false**. This state is final in the sense that once the state of the record becomes Declassified, it forever remains so.

The idea behind the proof is as follows: we use an invariant to track the logical state together with the points-to connectives for the physical state of the record. This way, we ensure that the protocol is followed by both threads.

To model the protocol in SeLoC, we use Iris's mechanism for user-defined ghost state. The exact way this mechanism works is described in [Jun+15; Jun+18b], but is not important for this paper. What is important is that via this mechanism we can define predicates $\mathsf{in\_state}_\gamma, \mathsf{state\_token}_\gamma : \mathsf{State} \to \mathit{iProp}$ that satisfy the rules in Figure 5.4b. The predicate $\mathsf{in\_state}_\gamma$ will be shared using an invariant, while thread2 will own the predicate $\mathsf{state\_token}_\gamma$. Rule STATE-AGREE states that the predicates $\mathsf{in\_state}_\gamma$ and $\mathsf{state\_token}_\gamma$ agree on the logical state. If a thread owns both predicates, it can change the logical state using STATE-CHANGE, but only in a way that respects the transition system. Rule DECLASSIFIED-DUP states that once a thread learns that the record is in the final state, *i.e.*, Declassified, this knowledge remains true forever. The predicates are indexed by a *ghost name* γ to allow for different instances of the transition system using STATE-ALLOC. Figure 5.4c displays the invariant that ties together the ghost and physical state. It is defined for the records r_1 and r_2 on the left- and the right-hand side, resp. We verify each thread separately with respect to this invariant, which we open every time we access the record.

(a) The protocol as a transition system.

STATE-ALLOC
$$\Rightarrow\!\!\!\!\mid_{\mathcal{E}} \exists \gamma.\ \mathsf{in\_state}_\gamma(\mathsf{Classified}) * \mathsf{state\_token}_\gamma(\mathsf{Classified})$$

STATE-AGREE
$$\frac{\mathsf{in\_state}_\gamma(s_1) \qquad \mathsf{state\_token}_\gamma(s_2)}{s_1 = s_2}$$

STATE-CHANGE
$$\frac{s_1 \rightarrow s_2 \qquad \mathsf{in\_state}_\gamma(s_1) \qquad \mathsf{state\_token}_\gamma(s_1)}{\Rightarrow\!\!\!\!\mid_{\mathcal{E}} \mathsf{in\_state}_\gamma(s_2) * \mathsf{state\_token}_\gamma(s_2)}$$

DECLASSIFIED-DUP
$$\frac{\mathsf{state\_token}_\gamma(\mathsf{Declassified})}{\mathsf{state\_token}_\gamma(\mathsf{Declassified}) * \mathsf{state\_token}_\gamma(\mathsf{Declassified})}$$

(b) The rules for ghost state.

$$\boxed{\begin{array}{ll} \big(\mathsf{in\_state}_\gamma(\mathsf{Classified}) * \exists i_1, i_2. & r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \mathbf{true} * r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \mathbf{true} * \\ & r_1.\mathsf{data} \mapsto_{\mathsf{L}} i_1 * r_2.\mathsf{data} \mapsto_{\mathsf{R}} i_2\big) \\ \vee\big(\mathsf{in\_state}_\gamma(\mathsf{Intermediate}) * \exists i. & r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \mathbf{true} * r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \mathbf{true} * \\ & r_1.\mathsf{data} \mapsto_{\mathsf{L}} i * r_2.\mathsf{data} \mapsto_{\mathsf{R}} i\big) \\ \vee\big(\mathsf{in\_state}_\gamma(\mathsf{Declassified}) * \exists i. & r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \mathbf{false} * r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \mathbf{false} * \\ & r_1.\mathsf{data} \mapsto_{\mathsf{L}} i * r_2.\mathsf{data} \mapsto_{\mathsf{R}} i\big) \end{array}}^{\mathcal{N}}$$

(c) The invariant.

Figure 5.4: Value-dependent classification.

Proof of the complete program. We symbolically execute the allocation of the records r_1 and r_2, giving us the resources $r_1.\mathit{is\_classified} \mapsto_{\mathsf{L}} \mathbf{true}$, $r_2.\mathit{is\_classified} \mapsto_{\mathsf{R}} \mathbf{true}$, $r_1.\mathsf{data} \mapsto_{\mathsf{L}} i_1$, and $r_2.\mathsf{data} \mapsto_{\mathsf{R}} i_2$. We then use STATE-ALLOC to obtain $\mathsf{in\_state}_\gamma(\mathsf{Classified})$ and $\mathsf{state\_token}_\gamma(\mathsf{Classified})$. With these resources at hand, we use DWP-INV-ALLOC to establish the invariant in Figure 5.4c, which can be shared between both threads. We use DWP-PAR with $\Psi_1(v_1, v_2) \triangleq \Psi_2(v_1, v_2) \triangleq (v_1 = v_2 = ())$, and use the token $\mathsf{state\_token}_\gamma(\mathsf{Classified})$ for the proof of the second thread.

Proof of thread1. We use the symbolic execution rules for dereferencing $r_1.\mathit{is\_classified}$ and $r_2.\mathit{is\_classified}$ until both of them become **false**. At that point, the invariant tells us that we are in the Declassified state. Subsequently, when using the symbolic execution rule for dereferencing $r_1.\mathsf{data}$ and $r_2.\mathsf{data}$, we use a copy of the predicate $\mathsf{state\_token}_\gamma(\mathsf{Declassified})$ to determine that the last disjunct of the invariant must hold. From that, we know that both $r_1.\mathsf{data}$ and $r_2.\mathsf{data}$ contain the same value. Using this information we can safely symbolically execute the assignments to the output location *out*.

Proof of thread2. We start the proof with the initial predicate $\mathsf{state\_token}_\gamma(\mathsf{Classified})$ and update the logical state with each assignment. The complete formalized proof can be found in the Coq mechanization. □

5.5 Type system and logical relations

We show how to define a type system for non-interference as an abstraction on top of SeLoC using the technique of *logical relations*. While logical relations have been used to model type systems and logics for safety and contextual refinement in (variants of) Iris before [KTB17; KSB17; Tim+18; FKB18; Jun+18a; Jun+21], we—for the first time—use logical relations in Iris to model a type system for non-interference (Section 5.5.1). We moreover show how we can combine type-checked code with code that has been manually verified using double weakest preconditions in SeLoC (Section 5.5.2).

The types that we consider are as follows:

$$\tau \in \mathit{Type} ::= \mathsf{unit} \mid \mathsf{int}^\chi \mid \mathsf{bool}^\chi \mid \tau \times \tau' \mid \mathsf{ref}\ \tau \mid (\tau \to \tau')^\chi$$

Here, $\chi, \xi \in \mathsf{Lbl}$ range over the *sensitivity labels* $\{\mathbf{L}, \mathbf{H}\}$ that form a lattice with $\mathbf{L} \sqsubseteq \mathbf{H}$. While any bounded lattice will do, we use the two-element lattice for brevity's sake.

The typing judgment is of the form $\Gamma \vdash e : \tau$, where Γ is an assignment of variables to types, e is an expression, and τ is a type. The typing rules are given in Figure 5.5. The rule TYPED-OUT shows that every output location $\ell \in \mathfrak{L}$ is typed as a reference to a low-sensitivity integer. By $\tau \sqcup \xi$ we denote the *level stamping* function, defined as follows:

$$\begin{aligned}
\mathsf{unit} \sqcup \xi &\triangleq \mathsf{unit} & (\tau \times \tau') \sqcup \xi &\triangleq (\tau \sqcup \xi) \times (\tau' \sqcup \xi) \\
\mathsf{int}^\chi \sqcup \xi &\triangleq \mathsf{int}^{\chi \sqcup \xi} & (\mathsf{ref}\ \tau) \sqcup \xi &\triangleq \mathsf{ref}\ \tau \\
\mathsf{bool}^\chi \sqcup \xi &\triangleq \mathsf{bool}^{\chi \sqcup \xi} & (\tau \to \tau')^\chi \sqcup \xi &\triangleq (\tau \to \tau')^{\chi \sqcup \xi}
\end{aligned}$$

$$\frac{\chi_1 \sqsubseteq \chi_2}{\mathsf{int}^{\chi_1} <: \mathsf{int}^{\chi_2}} \qquad \frac{\chi_1 \sqsubseteq \chi_2}{\mathsf{bool}^{\chi_1} <: \mathsf{bool}^{\chi_2}} \qquad \frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{\tau_1 \times \tau_2 <: \tau_1' \times \tau_2'}$$

$$\frac{\chi_1 \sqsubseteq \chi_2 \qquad \tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{(\tau_1 \to \tau_2)^{\chi_1} <: (\tau_1' \to \tau_2')^{\chi_2}} \qquad \tau <: \tau$$

$$\frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

(a) Subtyping rules.

$$\text{unit is flat} \qquad \mathsf{int}^{\chi} \text{ is flat} \qquad \mathsf{bool}^{\chi} \text{ is flat}$$

$$\frac{\tau_1 \text{ is flat} \qquad \tau_2 \text{ is flat}}{\tau_1 \times \tau_2 \text{ is flat}}$$

(b) Flat types.

TYPED-SUB
$$\frac{\tau <: \tau' \qquad \Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'}$$

TYPED-VAR
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

TYPED-OUT
$$\frac{\ell \in \mathfrak{L}}{\Gamma \vdash \ell : \mathsf{ref}\ \mathsf{int}^{\mathbf{L}}}$$

TYPED-UNIT
$$\Gamma \vdash () : \mathsf{unit}$$

TYPED-INT
$$\frac{i \in \mathbb{Z}}{\Gamma \vdash i : \mathsf{int}^{\chi}}$$

TYPED-BOOL
$$\frac{b \in \mathbb{B}}{\Gamma \vdash b : \mathsf{bool}^{\chi}}$$

TYPED-BINOP
$$\frac{\Gamma \vdash e : \mathsf{int}^{\chi} \qquad \Gamma \vdash t : \mathsf{int}^{\xi}}{\Gamma \vdash e + t : \mathsf{int}^{\chi \sqcup \xi}}$$

TYPED-REC
$$\frac{f : (\tau \to \tau')^{\chi}, x : \tau, \Gamma \vdash e : \tau' \sqcup \chi}{\Gamma \vdash \mathsf{rec}\ f\ x = e : (\tau \to \tau')^{\chi}}$$

TYPED-APP
$$\frac{\Gamma \vdash e : (\tau \to \tau')^{\chi} \qquad \Gamma \vdash t : \tau}{\Gamma \vdash e\ t : \tau' \sqcup \chi}$$

TYPED-IF-LOW
$$\frac{\Gamma \vdash e : \mathsf{bool}^{\mathbf{L}} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}$$

TYPED-IF-HIGH
$$\frac{\Gamma \vdash e : \mathsf{bool}^{\mathbf{H}} \quad \Gamma \vdash v : \tau \quad \Gamma \vdash w : \tau \quad v, w \text{ are values or variables in } \Gamma \quad \tau \text{ is flat}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ v\ \mathsf{else}\ w : \tau \sqcup \mathbf{H}}$$

TYPED-FORK
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{fork}\ \{e\} : \mathsf{unit}}$$

TYPED-ALLOC
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}(e) : \mathsf{ref}\ \tau}$$

TYPED-DEREF
$$\frac{\Gamma \vdash e : \mathsf{ref}\ \tau}{\Gamma \vdash !e : \tau}$$

TYPED-STORE
$$\frac{\Gamma \vdash e_1 : \mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \leftarrow e_2 : \mathsf{unit}}$$

TYPED-CAS
$$\frac{\Gamma \vdash e_1 : \mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau \qquad \tau \text{ is unboxed}}{\Gamma \vdash \mathsf{CAS}(e_1, e_2, e_3) : \mathsf{bool}^{\mathsf{lbl}(\tau)}}$$

(c) Selected expression typing rules.

Figure 5.5: Typing judgements of the SeLoC type system.

$$
\begin{aligned}
[\![\mathsf{unit}]\!](v_1, v_2) &\triangleq v_1 = v_2 = () \\
[\![\mathsf{int}^\chi]\!](v_1, v_2) &\triangleq v_1, v_2 \in \mathbb{Z} * (\chi = \mathbf{L} \to v_1 = v_2) \\
[\![\mathsf{bool}^\chi]\!](v_1, v_2) &\triangleq v_1, v_2 \in \mathbb{B} * (\chi = \mathbf{L} \to v_1 = v_2) \\
[\![\tau \times \tau']\!](v_1, v_2) &\triangleq \exists w_1, w_2, w_1', w_2'.\ (v_1 = (w_1, w_1')) * (v_2 = (w_2, w_2')) * \\
&\qquad\qquad [\![\tau]\!](w_1, w_2) * [\![\tau']\!](w_1', w_2') \\
[\![\mathsf{ref}\ \tau]\!](v_1, v_2) &\triangleq v_1, v_2 \in \mathit{Loc} * \boxed{\exists w_1\ w_2.\ v_1 \mapsto_{\mathsf{L}} w_1 * v_2 \mapsto_{\mathsf{R}} w_2 * [\![\tau]\!](w_1, w_2)}^{\mathcal{N}.(v_1, v_2)} \\
[\![(\tau \to \tau')^\chi]\!](v_1, v_2) &\triangleq \Box\Big(\forall w_1, w_2.\ [\![\tau]\!](w_1, w_2) \mathrel{-\!\!*} [\![\tau' \sqcup \chi]\!]_{\mathcal{E}}(v_1\ w_1)(v_2\ w_2)\Big) \\
[\![\tau]\!]_{\mathcal{E}}(e_1, e_2) &\triangleq \mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\tau]\!]\}
\end{aligned}
$$

Figure 5.6: The logical relations interpretation of types.

By $\mathsf{lbl}(\tau)$ we denote the *level approximation* of a type, defined as follows:

$$
\begin{aligned}
\mathsf{lbl}(\mathsf{unit}) &\triangleq \mathbf{L} & \mathsf{lbl}(\tau \times \tau') &\triangleq \mathsf{lbl}(\tau) \sqcup \mathsf{lbl}(\tau') \\
\mathsf{lbl}(\mathsf{int}^\chi) &\triangleq \chi & \mathsf{lbl}(\mathsf{ref}\ \tau) &\triangleq \mathbf{L} \\
\mathsf{lbl}(\mathsf{bool}^\chi) &\triangleq \chi & \mathsf{lbl}((\tau \to \tau')^\chi) &\triangleq \chi
\end{aligned}
$$

Intuitively, if we compare two values of type τ, then the result of the comparison should have the sensitivity level $\mathsf{lbl}(\tau)$.

To type check the set data structure from Section 5.2, we need to support benign branching on high-sensitivity Booleans. For that purpose we use the rule TYPED-IF-HIGH. In the rule, both branches should either be values or variables, ensuring that they do not perform any computations. In addition, both branches should be of a *flat type*—$\mathsf{int}^{\mathbf{H}}$, $\mathsf{bool}^{\mathbf{H}}$, or a product of two flat types. Function types are not flat because they can leak via timing behavior outside the if branch itself.

The atomic compare-and-swap operation is well-defined only for *unboxed types*: $\mathsf{int}^\chi, \mathsf{bool}^\chi, \mathsf{unit}, \mathsf{ref}\ \tau$. The intuition behind this is that unboxed types contain word-sized values, *c.f.* unboxed from Section 4.2.

Notice that our type system has no sensitivity labels on reference types, and no program counter label on the typing judgment. While such labels are common in security type systems for languages with (higher-order) references [PS03; Ter08; RG18; Zda02], a direct adaptation of such type systems is not sound with respect to the termination-sensitive notion of non-interference we consider. A counterexample is provided in Section 5.9.1.

5.5.1 Logical relations model

We give a semantic model of our type system using logical relations. The key idea of logical relations is to interpret each type τ as a relation on values, *i.e.*, to each type τ we assign an *interpretation* $[\![\tau]\!] : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$ where iProp is the type of

$$\textsc{logrel-if-low}\quad \frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\llbracket \mathsf{bool}^{\mathbf{L}} \rrbracket\} \qquad \mathsf{dwp}\ t_1\ \&\ t_2\ \{\Phi\} \qquad \mathsf{dwp}\ u_1\ \&\ u_2\ \{\Phi\}}{\mathsf{dwp}\ \mathsf{if}\ e_1\ \mathsf{then}\ t_1\ \mathsf{else}\ u_1\ \&\ \mathsf{if}\ e_2\ \mathsf{then}\ t_2\ \mathsf{else}\ u_2\ \{\Phi\}}$$

$$\textsc{logrel-if-high}\quad \frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\llbracket \mathsf{bool}^{\mathbf{H}} \rrbracket\} \qquad \mathsf{dwp}\ v_1\ \&\ v_2\ \{\llbracket\tau\rrbracket\} \qquad \mathsf{dwp}\ w_1\ \&\ w_2\ \{\llbracket\tau\rrbracket\} \qquad \tau\ \text{is flat}}{\mathsf{dwp}\ \mathsf{if}\ e_1\ \mathsf{then}\ v_1\ \mathsf{else}\ w_1\ \&\ \mathsf{if}\ e_2\ \mathsf{then}\ v_2\ \mathsf{else}\ w_2\ \{\llbracket\tau\rrbracket\}}$$

$$\textsc{logrel-store}\quad \frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\llbracket \mathsf{ref}\ \tau\rrbracket\} \qquad \mathsf{dwp}\ t_1\ \&\ t_2\ \{\llbracket\tau\rrbracket\}}{\mathsf{dwp}\ (e_1 \leftarrow t_1)\ \&\ (e_2 \leftarrow t_2)\ \{\llbracket \mathsf{unit}\rrbracket\}} \qquad \textsc{logrel-sub}\quad \frac{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\llbracket\tau\rrbracket\} \qquad \tau <: \tau'}{\mathsf{dwp}\ e_1\ \&\ e_2\ \{\llbracket\tau'\rrbracket\}}$$

Figure 5.7: A selection of compatibility rules.

SeLoC propositions. Intuitively, $\llbracket\tau\rrbracket(v_1, v_2)$ expresses that v_1 and v_2 of type τ are indistinguishable by a low-sensitivity attacker. The definition of $\llbracket\tau\rrbracket$ is given in Figure 5.6. We will now explain some interesting cases in detail.

The interpretation $\llbracket \mathsf{int}^{\mathbf{L}} \rrbracket$ contains the pairs of equal integers, while $\llbracket \mathsf{int}^{\mathbf{H}} \rrbracket$ contains the pairs of any two integers. This captures the intuition that a low-sensitivity attacker can observe low-sensitivity integers, but not high-sensitivity integers.

The interpretation $\llbracket \mathsf{ref}\ \tau \rrbracket$ captures that references ℓ_1 and ℓ_2 are indistinguishable iff they always hold values w_1 and w_2 that are indistinguishable at type τ. This is formalized by imposing an invariant that contains both points-to propositions $\ell_1 \mapsto_{\mathsf{L}} w_1$ and $\ell_2 \mapsto_{\mathsf{R}} w_2$, as well as the interpretation of τ that links the values w_1 and w_2. Notice that our interpretation of references does not require the locations ℓ_1 and ℓ_2 themselves to be syntactically equal. This is crucial for modeling dynamic allocation (recall that the allocation oracle described in Section 5.3.1 may depend on the contents of the heap).

The interpretation $\llbracket(\tau \to \tau')^{\chi}\rrbracket$ captures that functions v_1 and v_2 are indistinguishable iff for all inputs w_1 and w_2 indistinguishable at type τ, the behaviors of the expressions $v_1\ w_1$ and $v_2\ w_2$ are indistinguishable at type $\tau' \sqcup \chi$. To formalize what it means for the behavior of expressions (in this case $v_1\ w_1$ and $v_2\ w_2$) to be indistinguishable, we define the *expression interpretation* $\llbracket\tau\rrbracket_{\mathcal{E}} : Expr \times Expr \to iProp$ by lifting the value interpretation using double weakest preconditions.

The interpretation of functions is defined using the *persistence modality* $\square$ of Iris [Jun+18b, Section 2.3]. Intuitively, $\square P$ states that P holds without asserting ownership of any non-shareable resources. Having the persistence modality in this definition is common in logical relations in Iris [KTB17]—it ensures that indistinguishable functions remain indistinguishable forever.

The interpretation of expressions $[\![\_]\!]_{\mathcal{E}}$ generalizes to open terms by considering all well-typed substitutions. A (binary) substitution γ is a function $Var \to Val \times Val$. We write $\gamma_i(e)$ for a term e where each free variable x is substituted by $\pi_i(\gamma(x))$. A substitution γ is well-typed, notation $\llbracket\Gamma\rrbracket(\gamma)$, iff $\forall x.\ \llbracket\tau\rrbracket(\gamma(\Gamma(x)))$. We define the

semantic typing judgment as:

$$\Gamma \models e : \tau \triangleq \forall \gamma.\, (\llbracket \Gamma \rrbracket(\gamma) * I_{\mathfrak{L}}) \twoheadrightarrow \llbracket \tau \rrbracket_{\mathcal{E}}(\gamma_1(e), \gamma_2(e)).$$

Here, $I_{\mathfrak{L}}$ is the invariant on the observable locations (Section 5.4.3).

Theorem 5.7 (Soundness)**.** If $x_1 : \mathsf{int}^{\mathbf{H}}, \ldots, x_n : \mathsf{int}^{\mathbf{H}} \models e : \mathsf{int}^{\mathbf{L}}$ is a derivable in SeLoC, then e is secure, and the configuration $(e[\vec{i}/\vec{x}], \sigma)$ is safe (*i.e.*, cannot get stuck) for any list of integers $\vec{i}$, and any heap σ with $\sigma \sim_{\mathfrak{L}} \sigma$.

Proof. This a direct consequence of Theorem 5.5. □

The *fundamental property* of logical relations states that any program that can be type checked is semantically typed.

Proposition 5.8 (Fundamental property)**.** If $\Gamma \vdash e : \tau$, then $\Gamma \models e : \tau$ is derivable in SeLoC.

Proof. This proposition is proved by induction on the typing judgment $\Gamma \vdash e : \tau$ using so-called *compatibility rules* for each case. A selection of these rules is shown in Figure 5.7. □

5.5.2 Typing via manual proof

When composing the fundamental property (Proposition 5.8) and the soundness theorem (Theorem 5.7) we obtain that any typed program is secure. For instance, it allows us to show that the rand program is secure by type checking it, instead of performing a manual proof as done in Proposition 5.4.

However, semantic typing gives us more—it allows us to combine type-checked code with manually verified code. Let us consider the examples from Section 5.2, which are not typed according to the typing rules, but which we can *prove* to be semantically typed by dropping down to the interpretation of the semantic typing judgment in terms of double weakest preconditions.

Proposition 5.9. $\models \mathit{prog} : \mathsf{ref}\ \mathsf{int}^{\mathbf{L}} \to \mathsf{int}^{\mathbf{H}} \to \mathsf{unit} \times \mathsf{unit}$.

Proof. This is a direct consequence of Proposition 5.6. □

Proposition 5.10. $\models \mathit{awk} : \mathsf{int}^{\mathbf{H}} \to (\mathsf{unit} \to \mathsf{unit})^{\mathbf{L}} \to \mathsf{int}^{\mathbf{L}}$.

Proof. The proposition boils down to showing that for any $i_1, i_2 \in \mathbb{Z}$ and f_1, f_2 with $\llbracket(\mathsf{unit} \to \mathsf{unit})^{\mathbf{L}}\rrbracket(f_1, f_2)$, we have $\mathsf{dwp}\ \mathit{awk}\ i_1\ f_1\ \&\ \mathit{awk}\ i_2\ f_2\ \{v_1\, v_2.\, v_1 = v_2 = 0\}$. We verify this by establishing a monotone protocol similar to the one used in the proof of value-dependent classification in Section 5.4.2. The full proof can be found in the Coq mechanization. □

After establishing the semantic typing for, *e.g.*, *prog* we can use it in any context where a function of the type ref $\mathsf{int}^{\mathbf{L}} \to \mathsf{int}^{\mathbf{H}} \to \mathsf{unit} \times \mathsf{unit}$ is expected. For example:

$$h : \mathsf{int}^{\mathbf{H}}, f : \mathsf{ref}\ \mathsf{int}^{\mathbf{L}} \to \mathsf{int}^{\mathbf{H}} \to \mathsf{unit} \times \mathsf{unit}$$
$$\vdash \mathsf{let}\ x = \mathsf{ref}(0)\ \mathsf{in}\ \mathsf{fork}\ \{f\ x\ h\}; !\, x : \mathsf{int}^{\mathbf{L}}$$

Using the fundamental property (Proposition 5.8) we obtain a semantic typing judgment for the above program. Using Proposition 5.9 we establish that if we substitute *prog* for f, the resulting program will still be semantically typed, and thus secure by the soundness theorem (Theorem 5.7).

The same methodology can be used to assign the types to the safe array operations from Section 5.2.2 via manual proof, and compose them with the type checked set data structure from Section 5.2.1. The proof can be found in the Coq mechanization.

5.6 Modular separation logic specifications

Types provide a convenient way to specify program modules, but are not always strong enough to enable the verification of sophisticated clients. This is particularly relevant if the specification of a program module is to be used in a manual proof or relies on function correctness. We show that in addition to specifications through types, SeLoC can also be used to prove modular specification in separation logic. We demonstrate this approach on dynamically created locks (Section 5.6.1) and dynamically classified references (Section 5.6.2).

5.6.1 Locks

The HeapLang language we consider does not provide locks as primitive constructs, but provides the low-level compare-and-set (CAS) operation with which different locking mechanisms can be implemented. Figure 5.8 displays the implementation and specification of a spin lock. The specification makes use of a relational generalization of the common *lock predicates* in separation logic [Din+10; SB14]. The predicate $\mathsf{isLock}(lk_1, lk_2, R)$ expresses that the pair of locks lk_1 and lk_2 protect the resources R, and the predicate $\mathsf{locked}(lk_1, lk_2)$ expresses that the pair of locks is in acquired state.

To verify that the spin lock implementation conforms to the lock specification, we define the lock predicates using Iris's mechanism for invariants and user-defined ghost state. The proof (and invariant) are generalizations of the ordinary proof (and invariant) for functional correctness in Iris.

The rules of our lock specification are similar to the rules in logics with locks as primitives constructs, such as [MSE18; EM19]. There are two notable exceptions. First, in *loc. cit.* one needs to fix the set of locks and associated resources upfront, whereas in SeLoC one can create locks dynamically and attach an arbitrary resource R to each lock during the proof. Second, since locks are not primitive constructs in SeLoC, the specification also applies to different lock implementations, *e.g.*, a ticket lock, as we have shown in the Coq mechanization.

Implementation of a spin lock

$$\begin{aligned}\mathsf{let}\ \mathsf{newlock}\ () &= \mathsf{ref}(\mathbf{false})\\ \mathsf{let\ rec}\ \mathsf{acquire}\ lk &= \mathsf{if}\ \mathsf{CAS}(lk, \mathbf{false}, \mathbf{true})\ \ \mathsf{then}\ ()\\ &\qquad\qquad \mathsf{else}\ \mathsf{acquire}\ lk\\ \mathsf{let}\ \mathsf{release}\ lk &= lk \leftarrow \mathbf{false}\end{aligned}$$

Modular separation logic specification of locks

$$\textsc{newlock-spec}\ \frac{R}{\mathsf{dwp}\ \mathsf{newlock}\ ()\ \&\ \mathsf{newlock}\ ()\ \{lk_1\ lk_2.\ \mathsf{isLock}(lk_1, lk_2, R)\}}$$

$$\textsc{islock-dup}\ \frac{\mathsf{isLock}(lk_1, lk_2, R)}{\mathsf{isLock}(lk_1, lk_2, R) * \mathsf{isLock}(lk_1, lk_2, R)}$$

$$\textsc{acquire-spec}\ \frac{\mathsf{isLock}(lk_1, lk_2, R)}{\mathsf{dwp}\ \mathsf{acquire}\ lk_1\ \&\ \mathsf{acquire}\ lk_2\ \{R * \mathsf{locked}(lk_1, lk_2)\}}$$

$$\textsc{release-spec}\ \frac{\mathsf{isLock}(lk_1, lk_2, R) \qquad R \qquad \mathsf{locked}(lk_1, lk_2)}{\mathsf{dwp}\ \mathsf{release}\ lk_1\ \&\ \mathsf{release}\ lk_2\ \{\mathsf{True}\}}$$

Figure 5.8: Dynamically allocated locks in SeLoC.

5.6.2 Dynamically classified references

We consider a program module that encapsulates and generalizes dynamically classified references[8] as used in Section 5.2.3. This program module generalizes to clients with multiple threads and different sharing models. For example, clients in which multiple threads read and write to the dynamically classified reference, or in which the data gets classified again. The Coq mechanization contains such an example. The implementation of the module for dynamically classified references is given in Figure 5.9, and its specification[9] is given in Figure 5.9.

The main ingredient of the specification is the representation predicate $\mathsf{val\_dep}(\tau, r_1, r_2)$, which expresses that the dynamically classified references r_1 and r_2 contain related data of type τ at all times. Since $\mathsf{val\_dep}(\tau, r_1, r_2)$ expresses mere knowledge instead of ownership, it is duplicable (VALDEP-DUP). With the repre-

[8] In this context declassification refers to changing the dynamic classification of the reference. It is thus unrelated to static declassification policies [SS09], and the declassify function is unrelated to the eponymous function from [SM03].

[9] The specification in Figure 5.9 is derived from a more general HOCAP-style logically atomic specifications [SBP13], which can be found in Section 5.10 and the Coq mechanization.

$$\begin{aligned}
\textsf{let new_vdep } v &= \left\{\begin{array}{l} \mathit{data} = \textsf{ref}(v); \\ \mathit{is_classified} = \textsf{ref}(\textbf{false}) \end{array}\right\} \\
\textsf{let read } r &= !\, r.\mathit{data} \\
\textsf{let store } r\ v &= r.\mathit{data} \leftarrow v \\
\textsf{let classify } r &= r.\mathit{is_classified} \leftarrow \textbf{true} \\
\textsf{let get_classified } r &= !\, r.\mathit{is_classified} \\
\textsf{let declassify } r\ v &= r.\mathit{data} \leftarrow v; r.\mathit{is_classified} \leftarrow \textbf{false}
\end{aligned}$$

Figure 5.9: Implementation of dynamically classified references.

sentation predicate at hand we can formulate weak specifications for some operations. For instance, the rule READ-SAFE over-approximates the sensitivity-level of the values returned by the read operation, and dually, the rule STORE-SAFE under-approximates the sensitivity-level of the values stored using the store operation. Of course, at times we want to track the precise sensitivity-level. For that we use a *fractional token* $\mathsf{class}_{(r_1,r_2)}(\chi, q)$ with $q \in (0,1]_{\mathbb{Q}}$. This token is reminiscent of fractional permissions in separation logic. The proof rules for declassify and classify (DECLASSIFY-SEQ and CLASSIFY-SEQ) require the full fraction ($q = 1$) since they change the classification. The precise rules for read and store (READ-SEQ and STORE-SEQ) do not change the classification, and thus require an arbitrary fraction. The token is splittable according to CLASS-SPLIT so it can be shared between threads.

Since the rules for declassify and classify require a full fraction ($q = 1$), they do not allow for fine-grained sharing,[10] *i.e.*, they cannot be used to verify a program that runs declassify in parallel with classify. It is good that this is impossible—running these operations in parallel results in a race-condition, making it impossible to know what the final classification would be. However, it *is* possible to verify a program that runs declassify in parallel with read or store (using precise rules for these two operations) by sharing the token via an invariant. To access such a shared token one has to use the more general HOCAP-style logically atomic specifications found in Section 5.10 and the Coq mechanization.

Proof. In order to verify the implementation, we follow the usual approach of defining the representation predicate $\mathsf{val\_dep}(\tau, r_1, r_2)$ and token $\mathsf{class}_{(r_1,r_2)}(\chi, q)$ using Iris's invariant and protocol mechanism. The invariant expresses that, at all times, the fields *is_classified* of both records contain the same Boolean value b, and that the data in the records are related by $[\![\tau \sqcup \chi]\!]$. The relation between the Boolean values b and the security label χ, and the way it evolves, are expressed using a protocol visualized as the transition system in Figure 5.11.

[10] We can still achieve sharing by storing the token $\mathsf{class}_{(r_1,r_2)}(\chi, 1)$ in a lock, as outlined in Section 5.6.1.

NEW-VDEP
$$\frac{[\![\tau \sqcup \chi]\!](v_1, v_2)}{\mathsf{dwp}\ \mathsf{new\_vdep}\ v_1\ \&\ \mathsf{new\_vdep}\ v_2\ \left\{r_1\ r_2.\ \mathsf{val\_dep}(\tau, r_1, r_2) * \mathsf{class}_{(r_1,r_2)}(\chi, 1)\right\}}$$

VALDEP-DUP
$$\mathsf{val\_dep}(\tau, r_1, r_2) \vdash \mathsf{val\_dep}(\tau, r_1, r_2) * \mathsf{val\_dep}(\tau, r_1, r_2)$$

CLASS-SPLIT
$$\mathsf{class}_{(r_1,r_2)}(\chi, q_1) * \mathsf{class}_{(r_1,r_2)}(\chi, q_2) \dashv\vdash \mathsf{class}_{(r_1,r_2)}(\chi, q_1 + q_2)$$

READ-SAFE
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2)}{\mathsf{dwp}\ \mathsf{read}\ r_1\ \&\ \mathsf{read}\ r_2\ \{v_1\ v_2.\ [\![\tau \sqcup \mathbf{H}]\!](v_1, v_2)\}}$$

READ-SEQ
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q)}{\mathsf{dwp}\ \mathsf{read}\ r_1\ \&\ \mathsf{read}\ r_2\ \left\{v_1\ v_2.\ [\![\tau \sqcup \chi]\!](v_1, v_2) * \mathsf{class}_{(r_1,r_2)}(\chi, q)\right\}}$$

STORE-SAFE
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad [\![\tau]\!](v_1, v_2)}{\mathsf{dwp}\ \mathsf{store}\ r_1\ v_1\ \&\ \mathsf{store}\ r_2\ v_2\ \{\mathsf{True}\}}$$

STORE-SEQ
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q) \qquad [\![\tau \sqcup \chi]\!](v_1, v_2)}{\mathsf{dwp}\ \mathsf{store}\ r_1\ v_1\ \&\ \mathsf{store}\ r_2\ v_2\ \left\{\mathsf{class}_{(r_1,r_2)}(\chi, q)\right\}}$$

CLASSIFY-SEQ
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, 1)}{\mathsf{dwp}\ \mathsf{classify}\ r_1\ \&\ \mathsf{classify}\ r_2\ \left\{\mathsf{class}_{(r_1,r_2)}(\mathbf{H}, 1)\right\}}$$

DECLASSIFY-SEQ
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, 1) \qquad [\![\tau]\!](v_1, v_2)}{\mathsf{dwp}\ \mathsf{declassify}\ r_1\ v_1\ \&\ \mathsf{declassify}\ r_2\ v_2\ \left\{\mathsf{class}_{(r_1,r_2)}(\mathbf{L}, 1)\right\}}$$

GET-CLASSIFIED-SEQ
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q)}{\mathsf{dwp}\ \mathsf{get\_classified}\ r_1\ \&\ \mathsf{get\_classified}\ r_2\ \{b_1\ b_2.\ (b_1 = b_2) * \Phi(b_1, \chi, q)\}}$$

$$\textbf{where}\ \Phi(b, \chi, q) \triangleq \mathsf{class}_{(r_1,r_2)}(\chi, q) * ((b = \textbf{false}) \to (\chi = \mathbf{L}))$$

Figure 5.10: Modular specifications of dynamically classified references.

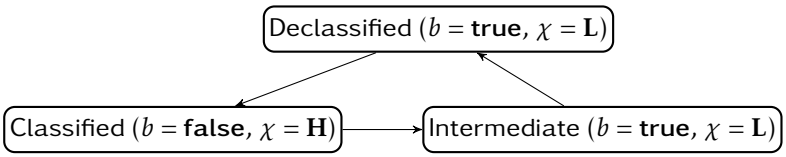


Figure 5.11: State transition system for dynamically classified references.

$$\mathsf{dwp}_{\mathcal{E}}\ e_1\ \&\ e_2\ \{\Phi\} \triangleq \begin{cases} \Rrightarrow_{\mathcal{E}} \Phi(e_1, e_2) & \text{if } e_1, e_2 \in \mathit{Val} \\ \Rrightarrow_{\mathcal{E}} \mathsf{False} & \text{if } e_1 \in \mathit{Val} \text{ xor } e_2 \in \mathit{Val} \\ \forall \sigma_1\ \sigma_2.\ \mathit{SR}(\sigma_1, \sigma_2) \mathrel{-\!\!*} {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \mathsf{red}(e_1, \sigma_1) * \mathsf{red}(e_2, \sigma_2) * & \\ \quad \forall e_1'\ \sigma_1'\ \vec{e_1}\ e_2'\ \sigma_2'\ \vec{e_2}.\ (e_1, \sigma_1) \rightarrow_{\mathsf{t}} (e_1' \vec{e_1}, \sigma_1') \wedge (e_2, \sigma_2) \rightarrow_{\mathsf{t}} (e_2' \vec{e_2}, \sigma_2') \mathrel{-\!\!*} & \\ \quad\quad {}^{\emptyset}\!\Rrightarrow^{\emptyset} \triangleright\, {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \mathit{SR}(\sigma_1', \sigma_2') * \mathsf{dwp}_{\mathcal{E}}\ e_1'\ \&\ e_2'\ \{\Phi\} * & \text{otherwise} \\ \quad\quad\quad \mathop{\Large *}_{(e_1'', e_2'') \in \vec{e_1} \times \vec{e_2}} . \mathsf{dwp}\ e_1''\ \&\ e_2''\ \{\mathsf{True}\} & \end{cases}$$

Figure 5.12: The model of double weakest preconditions.

5.7 Soundness

To prove soundness of SeLoC (Theorem 5.7), we give a model of double weakest preconditions in Iris (Section 5.7.1), and then construct a bisimulation out of this model (Section 5.7.2).

5.7.1 Model of double weakest preconditions

The model of the Iris logic [Kre+17; Jun+18b] consists of three layers:

- The Iris base logic, which contains the standard separation logic connectives (*e.g.*, $*$ and $\mathrel{-\!\!*}$), modalities (*e.g.*, $\triangleright$, $\Box$), and the machinery for user-defined ghost state.
- The invariant mechanism, which is built as a library on top of the Iris base logic.
- The Iris program logic, which is built as a library on top of the Iris base logic and invariant mechanism. It provides weakest preconditions for proving safety and functional correctness of concurrent programs.

We reuse the first two layers of Iris (the base logic and the invariant mechanism), on top of which we model our new notion of double weakest preconditions. This model is inspired by the model of ordinary (unary) weakest preconditions in Iris and the *product program* construction [BCK11]. The definition of $\mathsf{dwp}\ e_1\ \&\ e_2\ \{\Phi\}$ is given in Figure 5.12. Intuitively, it captures that the expressions e_1 and e_2 are executed in lock-step. This is done by case analysis:

- Either, both expressions e_1 and e_2 are values that are related by the postcondition Φ.
- Otherwise, both expressions e_1 and e_2 are reducible, and for any reductions $(e_1, \sigma_1) \to_{\mathsf{t}} (e_1', \sigma_1')$ and $(e_2, \sigma_2) \to_{\mathsf{t}} (e_2', \sigma_2')$, the expressions e_1' and e_2' are still related by dwp. If e_1 and e_2 fork off threads $\vec{e_1'}$ and $\vec{e_2'}$, then all of the forked-off threads are related pairwise by dwp. Moreover, it is required that $|\vec{e_1'}| = |\vec{e_2'}|$; this condition is implicit in the "big separating conjunction" $\mathbin{\ast}_{(e_1'', e_2'') \in \vec{e_1} \times \vec{e_2}}$.

The definition of dwp e_1 & e_2 $\{\Phi\}$ is modeled after the definition of wp e $\{\Phi\}$ in Iris [Jun+18b], but instead of Iris's *state interpretation* $S : State \to iProp$, we have a *state relation* $SR : State \times State \to iProp$ that keeps track of both the left and right-hand side heaps. The details can be found in the Coq formalization.

5.7.2 Constructing a bisimulation

The main challenge of constructing a strong low-bisimulation lies in connecting double weakest preconditions, at the level of separation logic, with strong-low bisimulations, at the meta level. The construction is done as follows:

1. We define a relation $\mathcal{R}$ that "lifts" double weakest preconditions out of the SeLoC logic into the meta-level (Definition 5.11).
2. We show that the dwp predicate is sound w.r.t. the relation $\mathcal{R}$: we can go from a proof of $I_{\mathfrak{L}} \vdash \mathsf{dwp}\ e\ \&\ t\ \{v_1\ v_2.\ v_1 = v_2\}$ in SeLoC to $(e, \sigma_1)\ \mathcal{R}\ (t, \sigma_2)$ for $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$. (Proposition 5.14).
3. We then show that the relation $\mathcal{R}$ satisfies a number of bisimulation-like properties (Lemma 5.15).
4. The relation $\mathcal{R}$ is not a bisimulation because it is not transitive. To fix this, we take its transitive closure $\mathcal{R}^*$, and verify that it is a strong-low bisimulation on configurations (Theorem 5.16).

Definition 5.11. We define the relation $\mathcal{R}$ on configurations of the same size to be the following:[11]

$$
\begin{aligned}
(e_0 e_1 \ldots e_m, \sigma_1)\ \mathcal{R}\ (t_0 t_1 \ldots t_m, \sigma_2) \triangleq \exists n : \mathbb{N}.\ & \\
\mathsf{True} \vdash \left({}^{\top}\!\Rightarrow^{\emptyset} \triangleright {}^{\emptyset}\!\Rightarrow^{\top}\right)^n \Rightarrow_{\top} SR(\sigma_1, \sigma_2) * I_{\mathfrak{L}} * & \\
\mathsf{dwp}\ e_0\ \&\ t_0\ \{v_1\ v_2.\ v_1 = v_2\} * & \\
\mathbin{\ast}_{1 \le i \le m}.\ \mathsf{dwp}\ e_i\ \&\ t_i\ \{\mathsf{True}\} &
\end{aligned}
$$

As was mentioned, $\mathcal{R}$ is defined at the meta-level, *i.e.*, outside SeLoC; in particular the existential quantifier $\exists n : \mathbb{N}$ is at the meta-level. The relation $\mathcal{R}$ relates two configurations if all the threads are related by a double weakest precondition, and execution of the main threads furthermore result in the same value. The invariant $I_{\mathfrak{L}}$ (which has been defined in Section 5.4.3) guarantees that the output locations $\mathfrak{L}$

[11] The definition in Coq includes additional details for bootstrapping the ghost state in Iris. We omit these details on paper, since in this thesis we do not talk about the ghost state implementation in Iris.

always contain the same data between any executions of the two configurations. The existentially quantified natural number n bounds the number of times the definition of double weakest preconditions has been unfolded. It is needed to show that $\mathcal{R}$ is closed under reductions.

The definition of $\mathcal{R}$ uses the iterated version of the modality ${}^{\top}\Rightarrow^{\emptyset} \triangleright {}^{\emptyset}\Rightarrow^{\top}$. We will need the following properties of this modality:[12]

Lemma 5.12. For any propositions P, Q and $n \in \mathbb{N}$, the following holds:

$$(P \twoheadrightarrow Q) \twoheadrightarrow \left(\left({}^{\top}\Rightarrow^{\emptyset} \triangleright {}^{\emptyset}\Rightarrow^{\top}\right)^n P\right) \twoheadrightarrow \left(\left({}^{\top}\Rightarrow^{\emptyset} \triangleright {}^{\emptyset}\Rightarrow^{\top}\right)^n Q\right)$$

Lemma 5.13. If ϕ is a pure predicate ϕ (*i.e.*, ϕ is built out of the intuitionistic logic connectives only), and

$$\left({}^{\top}\Rightarrow^{\emptyset} \triangleright {}^{\emptyset}\Rightarrow^{\top}\right)^n {}^{\top}\Rightarrow^{\emptyset} \phi$$

is derivable, then ϕ holds in the meta-logic (*c.f.* Theorem 2.1).

The relation $\mathcal{R}$ allows one to "lift" double weakest precondition proofs from inside the logic:

Proposition 5.14. If $I_{\mathfrak{L}} \vdash \mathsf{dwp}\ e\ \&\ t\ \{v_1\, v_2.\, v_1 = v_2\}$ is derivable in SeLoC, then $(e, \sigma_1)\ \mathcal{R}\ (t, \sigma_2)$ for any $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$.

Proof. For showing $(e, \sigma_1)\ \mathcal{R}\ (t, \sigma_2)$, pick $n = 0$. Because σ_1 and σ_2 agree on the $\mathfrak{L}$-locations (*i.e.*, $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$), we can establish the state relation $SR(\sigma_1, \sigma_2)$ and the invariant $I_{\mathfrak{L}}$. □

Lemma 5.15. The following properties hold:

1. $\mathcal{R}$ is symmetric;
2. If $(v\vec{e}, \sigma_1)\ \mathcal{R}\ (w\vec{t}, \sigma_2)$, then $v = w$;
3. If $(\vec{e}, \sigma_1)\ \mathcal{R}\ (\vec{t}, \sigma_2)$, then $|\vec{e}| = |\vec{t}|$ and $\sigma_1 \sim_{\mathfrak{L}} \sigma_2$;
4. If $(e_0 \ldots e_i \ldots, \sigma_1)\ \mathcal{R}\ (t_0 \ldots t_i \ldots, \sigma_2)$ and $(e_i, \sigma_1) \rightarrow_{\mathrm{t}} (e_i'\vec{e}, \sigma_1')$, then there exist an t_i', $\vec{t}$ and σ_2' such that:
   - $(t_i, \sigma_2) \rightarrow_{\mathrm{t}} (t_i'\vec{t}, \sigma_2')$;
   - $(e_0 \ldots e_i'\vec{e} \ldots, \sigma_1')\ \mathcal{R}\ (t_0 \ldots t_i'\vec{t} \ldots, \sigma_2')$.

Proof. The proof proceeds by unfolding the definitions of the $\mathcal{R}$ and dwp, appealing to the soundness and monotonicity of the iterated ${}^{\top}\Rightarrow^{\emptyset} \triangleright {}^{\emptyset}\Rightarrow^{\top}$ modality. For item (4), to establish that the configurations after a thread-level step are related, we pick a witness $n+1$, where n is the witness extracted from the relatedness of the original thread pools. □

[12] This modality was first used in [Tim+18], under the name of "future modality". The monotonicity lemma and a version of the soundness lemma were then used by Tassarotti in the Iris Coq formalization [Iri20] to prove the soundness of the weakest precondition in Iris.

By the above lemma, we now know that $\mathcal{R}$ has all the properties of a strong low-bisimulation on configurations (*c.f.* [SS00, Definition 6]), short of being a partial equivalence relation. Since $\mathcal{R}$ is not transitive, we consider its transitive closure $\mathcal{R}^*$, and verify that all the properties of a strong low-bisimulation hold for $\mathcal{R}^*$.

Theorem 5.16. The relation $\mathcal{R}^*$ is *a strong low-bisimulation* on configurations.

Proof. Because $\mathcal{R}$ is symmetric, $\mathcal{R}^*$ is a partial equivalence relation. It remains to verify that all the properties in Lemma 5.15 are preserved under the transitive closure. □

The theorem Theorem 5.16 in combination with Proposition 5.14 implies the soundness of SeLoC (Theorem 5.7).

5.8 Mechanization in Coq

We have mechanized the definition of SeLoC, the type system, the soundness proof, and all examples and derived constructions in the paper and the appendix in Coq. The mechanization has been built on top of the mechanization of Iris [Jun+16; Kre+17; Jun+18b], which readily provides the Iris base logic, the invariant mechanism, and the HeapLang language.

To carry out the mechanization effectively, we have made extensive use of the tactic language MoSeL (formerly Iris Proof Mode) for separation logic in Coq [KTB17; Kre+18]. Using MoSeL we were able to carry out in Coq the typical kind of reasoning steps one would do on paper. This was essential to mechanize the SeLoC logic (1818 line of Coq code), the type system (1355 lines), and all the examples (3223 lines).

5.9 Discussion

In this section we would like to discuss additional topics that are not necessary for the overall understanding of the chapter. First, we discuss the absence of sensitivity labels on reference types (Section 5.9.1). Secondly, we discuss the general rule for branching on high-sensitivity data in SeLoC (Section 5.9.2). Finally, we present the general form of modular specifications for references with value-dependent classifications (Section 5.10).

5.9.1 Sensitivity labels on references and aliasing

Most type systems for non-interference for languages with (higher-order) references annotate reference types with sensitivity labels, and annotate the typing judgment with a *program counter* label [PS03; Ter08; RG18; Zda02]. These annotations are used to prevent leaks via aliasing, while allowing more programs to be typed. Our type system (Section 5.5) does not have such annotations because some programs that are typeable using such annotations are not secure w.r.t. a termination-sensitive notion of

non-interference (*e.g.*, strong low-bisimulation). For example, termination-insensitive type systems usually accept the following program as secure:

$$(\text{if } h \text{ then } f \text{ else } g)\ ()$$

Here, h is a high-sensitivity Boolean, and f and g are functions of type $(\mathsf{unit} \to \mathsf{unit})^{\mathbf{L}}$. Under a termination-sensitive notion of security, the program is not secure because f and g can examine different termination behavior.

Despite this, let us examine why exactly we do not need labels on reference types to prevent leaks via aliasing, and argue that our approach still allows for benign aliasing of references. A classic example of an information leak via aliasing is:

$$\begin{aligned}&\text{let } p_1\ r\ s\ h = r \leftarrow \mathbf{true}; s \leftarrow \mathbf{true};\\ &\qquad\qquad \text{let } x = (\text{if } h \text{ then } r \text{ else } s) \text{ in}\\ &\qquad\qquad x \leftarrow \mathbf{false}; !\, r\end{aligned}$$

Both r and s contain low-sensitivity data, but by aliasing one or the other with x, the program leaks the high-sensitivity value h. In previous approaches such leaks are avoided by tracking aliasing information through sensitivity labels on references. The variable x would be typed as $(\mathsf{ref}\ \mathsf{int}^{\mathbf{L}})^{\mathbf{H}}$ because it was aliased in a high-sensitivity context (branching on h). The consequent assignment $x \leftarrow \mathbf{false}$ is then prevented by the type system since the label on the reference ($\mathbf{H}$) is not a below the label of the values that are stored in the reference ($\mathbf{L}$).

In SeLoC, the variable x will not be typeable at all. To see why that is the case, suppose we want to prove that the program is secure. For this, we let h_1 and h_2 denote high-sensitivity inputs for two runs of the program, and r_1, s_1 (resp. r_2, s_2) denote the low-sensitivity references arguments for the left-hand side program (resp. right-hand side program). Under these high-sensitivity inputs, we need to prove that the bodies of the let-expressions are indistinguishable, *i.e.*,

$$\mathsf{dwp}\ \text{if } h_1 \text{ then } r_1 \text{ else } s_1\ \&\ \text{if } h_2 \text{ then } r_2 \text{ else } s_2 \left\{ [\![\mathsf{ref}\ \mathsf{int}^{\mathbf{L}}]\!] \right\}$$

Proving this proposition, would in particular require proving $\mathsf{dwp}\ r_1\ \&\ s_2 \left\{ [\![\mathsf{ref}\ \mathsf{int}^{\mathbf{L}}]\!] \right\}$, which is impossible in SeLoC.

If we remove the trailing assignment $x \leftarrow \mathbf{false}$ the resulting program p_2 becomes trivially secure, and many termination-insensitive type systems accept it as such:

$$\begin{aligned}&\text{let } p_2\ r\ s\ h = r \leftarrow \mathbf{true}; s \leftarrow \mathbf{true};\\ &\qquad\qquad \text{let } x = (\text{if } h \text{ then } r \text{ else } s) \text{ in}\\ &\qquad\qquad !\, r\end{aligned}$$

Our type system cannot be used to type check this example: as we have just explained, we cannot type the let-expression at all. Despite this, we can fall back on the double weakest preconditions to verify the security of p_2, *i.e.*, we can prove:

$$\begin{aligned}&[\![\mathsf{ref}\ \mathsf{bool}^{\mathbf{L}}]\!](r_1, r_2) * [\![\mathsf{ref}\ \mathsf{bool}^{\mathbf{L}}]\!](s_1, s_2) *\\ &\ [\![\mathsf{bool}^{\mathbf{H}}]\!](h_1, h_2) \vdash \mathsf{dwp}\ p_2\ r_1\ s_1\ h_1\ \&\ p_2\ r_2\ s_2\ h_2\ \{ [\![\mathsf{unit}]\!] \}\end{aligned}$$

by symbolic execution. Using our logic, we can perform a case distinction on the Boolean values h_1 and h_2, which amounts to proving

- $\mathsf{dwp}\ p_2\ r_1\ s_1\ \mathbf{true}\ \&\ p_2\ r_2\ s_2\ \mathbf{true}\ \{[\![\mathsf{unit}]\!]\}$,
- $\mathsf{dwp}\ p_2\ r_1\ s_1\ \mathbf{true}\ \&\ p_2\ r_2\ s_2\ \mathbf{false}\ \{[\![\mathsf{unit}]\!]\}$,
- *etc.*

We solve all these goals by symbolic execution. This example demonstrates the advantages of combining typing with manual proofs.

We believe that the restriction on the typing of the let x-binding is not unreasonable in case of termination-sensitive and progress-sensitive security condition. As we have mentioned, if we take termination and timing behavior into account, the liberal compositional reasoning that is enjoyed by termination-insensitive type systems is no longer sound. In presence of higher-order functions and store, we can write the counterexample from the beginning of this section in the form of p_2 to obtain the program p_3 below:

$$\begin{aligned}&\mathsf{let}\ p_3\ f\ g\ h = r \leftarrow f; s \leftarrow g;\\ &\qquad\qquad \mathsf{let}\ x = (\mathsf{if}\ h\ \mathsf{then}\ r\ \mathsf{else}\ s)\ \mathsf{in}\\ &\qquad\qquad (!\,x)()\end{aligned}$$

The variable x now aliases a reference to a function. If f and g exhibit different termination behavior, then the value of h can be observed by invoking $!\,x$.

5.9.2 Generalized rule for branching

The notion of security that we use (strong low-bisimulation) allows for branching on high-sensitivity data, provided that the timing behavior of the branches is indistinguishable. However, if we branch on a high-sensitivity Boolean, it is insufficient to verify that each individual branch is secure, we also have to verify that the two different branches are indistinguishable for the attacker. This kind of condition is present in the rule LOGREL-IF:

LOGREL-IF

$$\frac{\begin{array}{c}\mathsf{dwp}\ e_1\ \&\ e_2\ \{[\![\mathsf{bool}^\chi]\!]\}\\ \mathsf{dwp}\ t_1\ \&\ t_2\ \{\Phi\} \wedge \mathsf{dwp}\ u_1\ \&\ u_2\ \{\Phi\} \wedge (\chi \not\sqsubseteq \mathbf{L} \rightarrow (\mathsf{dwp}\ u_1\ \&\ t_2\ \{\Phi\} \wedge \mathsf{dwp}\ t_1\ \&\ u_2\ \{\Phi\}))\end{array}}{\mathsf{dwp}\ (\mathsf{if}\ e_1\ \mathsf{then}\ t_1\ \mathsf{else}\ u_1)\ \&\ (\mathsf{if}\ e_2\ \mathsf{then}\ t_2\ \mathsf{else}\ u_2)\ \{\Phi\}}$$

We can speak of two different branches being indistinguishable because we have moved from a unary typing system to a binary logic.

Recall, that our inference rules are interpreted as an separating implication, where the premises are joined together by a separating conjunction. To prove each premise, the user of the rule has to distribute the resources they currently have among the premises. The last four premises in LOGREL-IF, however, are joined by a regular intuitionistic conjunction ($\wedge$). The user still has to prove both of those premises if they wish to apply the rule, but this time they do not have to split their resources, *i.e.*, they are able to reuse the same resource to prove all the premises. This corresponds

to the fact that there are four possible combinations of branches, but only one of the combinations can actually occur.

This general rule LOGREL-IF is used to derive the compatibility rules LOGREL-IF-LOW and LOGREL-IF-HIGH in Figure 5.7 in Section 5.5.

5.10 HOCAP-style modular specifications

We provide modular logically atomic specifications for the module of dynamically classified references (Section 5.6.2) in Figure 5.13. These specifications are stronger than the ones given in Figure 5.13 in the sense that they are *logically atomic*, *i.e.*, they allow one to open invariants around operations. This is achieved using the HOCAP [SBP13] approach to logical atomicity. Note that the weaker specifications in Figure 5.10 (from Section 5.6.2) can be derived from the HOCAP-style specifications in Figure 5.13.

5.11 Related work

5.11.1 Security based on strong low-bisimulations

The security condition we use, a strong low-bisimulation due to Sabelfeld and Sands [SS00], has been studied in a variety of related work. In *loc. cit.* the notion of a strong low-bisimulation is applied to a first-order stateful language with concurrency. It is also shown that this notion implies a scheduler-independent bisimulation known as ρ-specific probabilistic bisimulation. Sabelfeld and Sands presented both strong low-bisimulation on thread pools and configurations. We use the bisimulation relation on configurations because it allows for a flow-sensitive analysis and readily supports dynamic allocation.

Strong low-bisimulations are highly compositional: if a thread e is secure w.r.t. a strong low-bisimulation, then the composition of e with *any other* thread is secure. Unfortunately, this property makes it non-trivial to adapt strong low-bisimulations for analyses that are flow-sensitive in thread composition. We work around this issue by composing the components at the level of the logic (as double weakest preconditions), and not at the level of the bisimulations, despite the fact that we use strong low-bisimulations as an auxiliary notion in our soundness proof. By performing the composition at the level of the logic, we can use Iris invariants and modular specifications to put restrictions onto which threads can be composed.

Another way of enabling flow-sensitive analysis was developed by Mantel *et al.* [MSS11], who relaxed the notion of a strong low-bisimulation to a *strong low-bisimulation modulo modes*. Their approach enables rely-guarantee style reasoning at the level of the bisimulation. Notably, using the notion of strong low-bisimulations modulo modes one can specify that no other threads can read or write to a certain location.

Based on the notion of strong low-bisimulations modulo modes, the Covern project [Mur+16; SM19; MSE18] developed a series of logics for rely/guarantee reasoning. Notably, Murray *et al.* [MSE18] presented the first fully mechanized

VALDEP-PERSISTENT
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2)}{\Box\,\mathsf{val\_dep}(\tau, r_1, r_2)}$$

CLASSIFICATION-AGREE
$$\frac{\mathsf{class}_{(r_1,r_2)}(\chi_1, q_1) \qquad \mathsf{class}_{(r_1,r_2)}(\chi_2, q_2)}{\chi_1 = \chi_2}$$

CLASSIFICATION-OP
$$\mathsf{class}_{(r_1,r_2)}(\chi, q_1) * \mathsf{class}_{(r_1,r_2)}(\chi, q_2) \dashv\vdash \mathsf{class}_{(r_1,r_2)}(\chi, q_1 + q_2)$$

CLASSIFICATION-1-EXCLUSIVE
$$\frac{\mathsf{class}_{(r_1,r_2)}(\chi, 1) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q)}{\mathsf{False}}$$

CLASSIFICATION-AUTH-AGREEE
$$\frac{\mathsf{class\_auth}_{(r_1,r_2)}(\chi_1) \qquad \mathsf{class}_{(r_1,r_2)}(\chi_2, q)}{\chi_1 = \chi_2}$$

CLASSIFICATION-UPDATE
$$\frac{\mathsf{class\_auth}_{(r_1,r_2)}(\chi) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, 1)}{\Rightarrow\!\!\!\!\!\!\Rightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\chi') * \mathsf{class}_{(r_1,r_2)}(\chi', 1)}$$

READ-SPEC
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \\ (\forall \chi\, v_1\, v_2.\, \mathsf{class\_auth}_{(r_1,r_2)}(\chi) * [\![\tau \sqcup \chi]\!](v_1, v_2) \Rrightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\chi) * \Phi(v_1, v_2))}{\mathsf{dwp\ read}\ r_1\ \&\ \mathsf{read}\ r_2\ \{\Phi\}}$$

WRITE-SPEC
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \\ (\forall \chi.\, \mathsf{class\_auth}_{(r_1,r_2)}(\chi) \Rrightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\chi) * [\![\tau \sqcup \chi]\!](v_1, v_2) * \Phi((),()))}{\mathsf{dwp\ store}\ r_1\ v_1\ \&\ \mathsf{store}\ r_2\ v_2\ \{\Phi\}}$$

IS-CLASSIFIED-SPEC
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \\ (\forall \chi\, b.\, \mathsf{class\_auth}_{(r_1,r_2)}(\chi) \Rrightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\chi) * ((b = \mathbf{false} \to \chi = \mathbf{L}) \mathrel{-\!\!*} \Phi(b, b)))}{\mathsf{dwp\ get\_classified}\ r_1\ \&\ \mathsf{get\_classified}\ r_2\ \{\Phi\}}$$

DECLASSIFY-SPEC
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q) \\ (\mathsf{class\_auth}_{(r_1,r_2)}(\chi) * \mathsf{class}_{(r_1,r_2)}(\chi, q) \Rrightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\mathbf{L}) * \mathsf{class}_{(r_1,r_2)}(\mathbf{L}, q) * (\mathsf{class}_{(r_1,r_2)}(\mathbf{L}, q) \mathrel{-\!\!*} \Phi((),())))}{\mathsf{dwp\ declassify}\ r_1\ v_1\ \&\ \mathsf{declassify}\ r_2\ v_2\ \{\Phi\}}$$

CLASSIFY-SPEC
$$\frac{\mathsf{val\_dep}(\tau, r_1, r_2) \qquad \mathsf{class}_{(r_1,r_2)}(\chi, q) \\ (\mathsf{class\_auth}_{(r_1,r_2)}(\chi) * \mathsf{class}_{(r_1,r_2)}(\chi, q) \Rrightarrow \mathsf{class\_auth}_{(r_1,r_2)}(\mathbf{H}) * \Phi((),()))}{\mathsf{dwp\ classify}\ r_1\ \&\ \mathsf{classify}\ r_2\ \{\Phi\}}$$

NEW-VDEP-SPEC
$$\frac{[\![\tau \sqcup \chi]\!](v_1, v_2) \qquad (\forall r_1\, r_2.\, \mathsf{val\_dep}(\tau, r_1, r_2) * \mathsf{class}_{(r_1,r_2)}(\chi, 1) \mathrel{-\!\!*} \Phi(r_1, r_2))}{\mathsf{dwp\ new\_vdep}\ v_1\ \&\ \mathsf{new\_vdep}\ v_2\ \{\Phi\}}$$

Figure 5.13: HOCAP-style specifications for dynamically classified references.

program logic for non-interference of concurrent programs with shared memory, which is also called Covern. While Covern is not a separation logic, it has been extended to allow for flexible reasoning about non-interference in presence of value-dependent classifications [Mur+16]. In terms of the object language, Covern does not support fine-grained concurrency, arrays, or dynamically allocated references. Since Covern does not support fine-grained concurrency, locks are modeled as primitives in the language and logic, while they are derived constructs in our work. As a result of that, Covern's notion of strong low-bisimulations is tied to the operational semantics of locks, *i.e.*, it is considered *modulo* the variables that are held by locks. The set of locks, and the variables they protect, has to be provided statically. Hence their approach does not immediately generalize to support dynamically allocated locks, nor to reason about locks that protect other resources than permissions to write to or read from variables. Value-dependent classifications are also primitive in Covern [Mur+16], while they are derived constructs in our work. Covern has two separate primitive rules for assignment to "normal" variables and for assignment to "control" variables (*i.e.*, variables that signify the classification levels).

5.11.2 Program logics for non-interference

Early work by Beringer and Hofmann [BH07] established a connection between Hoare logic and non-interference. They did so for a first-order sequential language with a simple non-interference condition. Non-interference was encoded through self-composition and renaming, making sure that both parts of the composed program operate on different parts of the heap (something that one gets by construction in separation logic). Notably, they proved the non-interference property of two type systems by constructing models of the type systems in their Hoare logic. They also showed how to extend their approach to object-oriented type systems.

5.11.3 Separation logics for non-interference

Karbyshev *et al.* [Kar+18] devised a compositional type-and-effect system based on separation logic to prove non-interference of concurrent programs with channels. Their system is sound w.r.t. termination-insensitive non-interference allowing for races on low-sensitivity locations. They consider security for arbitrary (deterministic) schedulers, and allow for a *rescheduling* operation in the programming language to prevent scheduler tainting. To achieve that, their logical rule for rescheduling treats the scheduler as a splittable separation logic resource, allowing one to share it between threads. In terms of the object language, they consider a first-order language without dynamic memory allocation, and the concurrency primitives are based on channels with send and receive operations rather than our low-level fine-grained concurrency model. They do not provide a logic for modular reasoning about program modules.

The recently proposed separation logic SecCSL [EM19] enables reasoning about value-dependent information flow control policies through a relational interpretation of separation logic. One of the main advantages of the SecCSL approach is its amenability to automation. However, to achieve that, they restrict to a first-order

separation logic with restricted language features, *i.e.*, a first-order language with first-order references, and a coarse-grained synchronization mechanism. SecCSL does not support dynamically allocated references out of the box. However, we believe that it can be extended to support dynamic allocation, as long as the semantics for allocation are deterministic and do not depend on the global heap.

The security condition in SecCSL [EM19] is non-standard, and is geared to providing meaning to the intermediate Hoare triples. Because of that, their formulation of non-interference is closely intertwined with the semantics of the logic.

Costanzo and Shao [CS14] devised a separation logic for proving non-interference of first-order sequential programs. One of the novelties of their system is the support for declassification in the form of *delimited release* [SM03]. While we do not study declassification policies in this paper, we believe that the approach of Costanzo and Shao can be adapted to our setting, provided that we are willing to relax the notion of a strong low-bisimulation.

5.11.4 Type systems for non-interference

As discussed in the introduction (Section 5.1), a lot of work on non-interference in the programming languages area has focused on type-system based approaches. Such approaches are amendable to high degrees of automation, but lack the ability to reason about functional correctness. Due to an abundance of prior work on in this area, we restrict to directly related work.

Pottier and Simonet developed Flow Caml [PS03], a type system for termination-insensitive non-interference for sequential higher-order language in the spirit of Caml. Soundness w.r.t. non-interference is proven with the *product programs* technique. This kind of self-composition was an inspiration for our model of double weakest preconditions, although we avoid self-composition of programs at the syntactic level.

Terauchi [Ter08] devised a capabilities-based type system for *observational determinism* [ZM03]. Observational determinism is a formulation of non-interference for concurrent programs that is substantially different from the strong low-bisimulation considered in this paper. In particular, under observational determinism, no races on low-sensitivity locations are allowed, ruling out *e.g.*, the rand function from Section 5.3.3.

5.11.5 Logical relation models

The technique of logical relations is widely used for proving the soundness of type systems and logics. The work on step-indexing [Ahm06; App+07] made it possible to scale logical relations to languages with higher-order references and recursive types. Notably, Rajani and Garg [RG18] describe a step-indexed Kripke-style model for two information flow aware type systems for a sequential language with higher-order references. While they do not consider concurrency and their notion of non-interference is different from ours (their notion is termination- and progress-insensitive), their model is similar in spirit. However, we make use of the "logical" approach to step-indexing [DAB09] in Iris to avoid explicit step-indexes in definitions and proofs.

The relational model of our type system is directly inspired by a line of work on interpretation of type systems and logical relations in Iris [KTB17; KSB17; Jun+18a; Jun+21; Tim+18; FKB18], but this previous work focused on reasoning about safety and contextual equivalence of programs, while we target non-interference. For that purpose we developed double weakest preconditions.

The idea of using logical relations to reason about the combination of typed and manually verified code has been used before in the context of Iris. Jung *et al.* [Jun+18a; Jun+21] use it to reason about unsafe code in Rust, and Krogh-Jespersen *et al.* [KSB17] use it in the context of type-and-effect systems.

5.12 Conclusions and future work

We have presented SeLoC—the first separation logic for non-interference that combines type checking and manual proof. It supports fine-grained concurrency, higher-order functions, and dynamic (higher-order) references. The key feature of SeLoC is its novel connective for double weakest preconditions, which in combination with Iris-style invariants, allows for compositional reasoning. We have proved soundness of SeLoC with respect to a standard notion of security.

In future work we want to develop a more expressive type system. To develop such a type system, we want to transfer back reasoning principles from SeLoC into constructs that can be type checked automatically. Moreover, we would like to study declassification in the sense of delimited information release and static declassification policies [SS09; SM03; BNR07; CS14].

Bibliography

[AAV02] Amal Ahmed, Andrew W. Appel, and Roberto Virga. "A Stratified Semantics of General References Embeddable in Higher-order Logic". In: *LICS*. 2002, pp. 75–86. DOI: 10.1109/LICS.2002.1029818.

[AB08] Amal Ahmed and Matthias Blume. "Typed Closure Conversion Preserves Observational Equivalence". In: *ICFP*. 2008, pp. 157–168. DOI: 10.1145/1411204.1411227.

[ADR09] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. "State-dependent representation independence". In: *POPL*. 2009, pp. 340–353. DOI: 10.1145/1480881.1480925.

[Agu+19] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. "A relational logic for higher-order programs". In: *Journal of Functional Programming* 29 (2019), e16. DOI: 10.1017/S0956796819000145.

[Ahm04] Amal Ahmed. "Semantics of types for mutable state". PhD thesis. Princeton University, 2004.

[Ahm06] Amal Ahmed. "Step-indexed syntactic logical relations for recursive and quantified types". In: *ESOP*. Vol. 3924. LNCS. 2006, pp. 69–83. DOI: 10.1007/11693024_6.

[AL91] Martín Abadi and Leslie Lamport. "The existence of refinement mappings". In: *Theoretical Computer Science* 82.2 (1991), pp. 253–284. DOI: 10.1016/0304-3975(91)90224-P.

[AM01] Andrew W. Appel and David McAllester. "An indexed model of recursive types for foundational proof-carrying code". In: *TOPLAS* 23.5 (2001), pp. 657–683. DOI: 10.1145/504709.504712.

[Ami+07] Daphna Amit, Noam Rinetzky, Thomas Reps, Mooly Sagiv, and Eran Yahav. "Comparison under Abstraction for Verifying Linearizability". In: *CAV*. Vol. 4590. LNCS. 2007, pp. 477–490. DOI: 10.1007/978-3-540-73368-3_49.

[App+07] Andrew W. Appel, Paul-André Melliès, Christopher Richards, and Jérôme Vouillon. "A very modal model of a modern, major, general type system". In: *POPL*. 2007, pp. 109–122. DOI: 10.1145/1190216.1190235.

[App14] Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[AS01] Johan Agat and David Sands. "On Confidentiality and Algorithms". In: *S&P*. 2001, pp. 64–77. DOI: 10.1109/SECPRI.2001.924288.

[Bar+12] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. "Probabilistic Relational Reasoning for Differential Privacy". In: *POPL*. 2012, pp. 97–110. DOI: 10.1145/2103656.2103670.

[Bar+13] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. "Easycrypt: A tutorial". In: *FOSAD*. Vol. 8604. LNCS. 2013, pp. 146–166. DOI: 10.1007/978-3-319-10082-1_6.

[Bat+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. "Mathematizing C++ concurrency". In: *POPL*. 2011, pp. 55–66. DOI: 10.1145/1926385.1926394.

[BB20] Lars Birkedal and Aleš Bizjak. *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic*. 2020. URL: https://iris-project.org/tutorial-material.html.

[BBS13] Lars Birkedal, Aleš Bizjak, and Jan Schwinghammer. "Step-Indexed Relational Reasoning for Countable Nondeterminism". In: *Logical Methods in Computer Science* 9.4 (2013). DOI: 10.2168/LMCS-9(4:4)2013.

[BBT07] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. "BI-hyperdoctrines, higher-order separation logic, and abstraction". In: *TOPLAS* 29.5 (2007), p. 24. DOI: 10.1145/1275497.1275499.

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Relational Verification Using Product Programs". In: *FM*. Vol. 6664. LNCS. 2011, pp. 200–214. DOI: 10.1007/978-3-642-21437-0_17.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. "Symbolic Execution with Separation Logic". In: *APLAS*. Vol. 3780. LNCS. 2005, pp. 52–68. DOI: 10.1007/11575467_5.

[Ben04] Nick Benton. "Simple relational correctness proofs for static analyses and program transformations". In: *POPL*. 2004, pp. 14–25. DOI: 10.1145/964001.964003.

[Ber+08] Josh Berdine, Tal Lev-Ami, Roman Manevich, Ganesan Ramalingam, and Mooly Sagiv. "Thread Quantification for Concurrent Shape Analysis". In: *CAV*. Vol. 5123. LNCS. 2008, pp. 399–413. DOI: 10.1007/978-3-540-70545-1_37.

[BGZ09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. "Formal Certification of Code-Based Cryptographic Proofs". In: *POPL*. 2009, pp. 90–101. DOI: 10.1145/1480881.1480894.

[BH07] Lennart Beringer and Martin Hofmann. "Secure Information Flow and Program Logics". In: *CSF*. 2007, pp. 233–248. DOI: 10.1109/CSF.2007.30.

[BH18] Callum Bannister and Peter Höfner. "False Failure: Creating Failure Models for Separation Logic". In: *RAMiCS*. Vol. 11194. LNCS. 2018, pp. 263–279. DOI: 10.1007/978-3-030-02149-8_16.

[BHK18] Callum Bannister, Peter Höfner, and Gerwin Klein. "Backwards and Forwards with Separation Logic". In: *ITP*. Vol. 10895. LNCS. 2018, pp. 68–87. DOI: 10.1007/978-3-319-94821-8_5.

[Bir+11] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. "Step-indexed Kripke models over recursive worlds". In: *POPL*. 2011, pp. 119–132. DOI: 10.1145/1926385.1926401.

[Biz+19] Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. "Iron: managing obligations in higher-order concurrent separation logic". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 65:1–65:30. DOI: 10.1145/3290378.

[BNN16] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. "Relational logic with framing and hypotheses". In: *FSTTCS*. Vol. 65. LIPIcs. 2016, 11:1–11:16. DOI: 10.4230/LIPIcs.FSTTCS.2016.11.

[BNR07] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. "Towards a Logical Account of Declassification". In: *PLAS*. 2007, pp. 61–66. DOI: 10.1145/1255329.1255340.

[BNR13] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. "Local reasoning for global invariants, part I: Region logic". In: *JACM* 60.3 (2013), 18:1–18:56. DOI: 10.1145/2485982.

[BO16] Stephen Brookes and Peter O'Hearn. "Concurrent separation logic". In: *ACM SIGLOG News* 3.3 (2016), pp. 47–65.

[Bor+05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. "Permission accounting in separation logic". In: *POPL*. ACM, 2005, pp. 259–270. DOI: 10.1145/1040305.1040327.

[Boy03] John Boyland. "Checking Interference with Fractional Permissions". In: *SAS*. Vol. 2694. LNCS. 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4.

[Bro07] Stephen Brookes. "A Semantics for Concurrent Separation Logic". In: *TCS* 375.1-3 (2007), pp. 227–270. DOI: 10.1016/j.tcs.2006.12.034.

[BST12] Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. "A Concurrent Logical Relation". In: *CSL*. Vol. 16. LIPIcs. 2012, pp. 107–121. DOI: 10.4230/LIPIcs.CSL.2012.107.

[Bur+10] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. "Line-up: A Complete and Automatic Linearizability Checker". In: *PLDI*. 2010, pp. 330–340. DOI: 10.1145/1806596.1806634.

[Cal+11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. "Compositional Shape Analysis by Means of Bi-Abduction". In: *J. ACM* 58.6 (2011), 26:1–26:66. DOI: 10.1145/2049697.2049700.

[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: *JAR* 61.1-4 (2018), pp. 367–422. DOI: 10.1007/s10817-018-9457-5.

[Čer+10] Pavol Černý, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. "Model Checking of Linearizability of Concurrent List Implementations". In: *CAV*. Vol. 6174. LNCS. 2010, pp. 465–479. DOI: 10.1007/978-3-642-14295-6_41.

[Cha+19] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying Concurrent, Crash-Safe Systems with Perennial". In: *SOSP*. 2019, pp. 243–258. DOI: 10.1145/3341301.3359632.

[Cha11] Arthur Charguéraud. "Characteristic formulae for the verification of imperative programs". In: *ICFP*. ACM, 2011, pp. 418–430. DOI: 10.1145/2034773.2034828.

[Cha20] Arthur Charguéraud. *Separation Logic for Sequential Programs (Functional Pearl)*. To appear in ICFP'20. 2020.

[Çiç+17] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. "Relational Cost Analysis". In: *POPL*. 2017, pp. 316–329. DOI: 10.1145/3009837.3009858.

[Coh+09a] 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. 2009, pp. 23–42. DOI: 10.1007/978-3-642-03359-9_2.

[Coh+09b] Ernie Cohen, Michał Moskal, Stephan Tobies, and Wolfram Schulte. "A Precise Yet Efficient Memory Model For C". In: *ENTCS* 254 (2009), pp. 85–103. DOI: 10.1016/j.entcs.2009.09.061.

[Coq20] The Coq Development Team. *The Coq Proof Assistant, version 8.11.0*. Version 8.11.0. Jan. 2020. DOI: 10.5281/zenodo.3744225.

[ÇPG16] Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. "A type theory for incremental computational complexity with control flow changes". In: *ICFP*. 2016, pp. 132–145. DOI: 10.1145/2951913.2951950.

[CS14] David Costanzo and Zhong Shao. "A Separation Logic for Enforcing Declarative Information Flow Control Policies". In: *POST*. Vol. 8414. LNCS. 2014, pp. 179–198. DOI: 10.1007/978-3-642-54792-8_10.

[DA13] Josiah Dodds and Andrew W. Appel. "Mostly Sound Type System Improves a Foundational Program Verifier". In: *CPP*. Vol. 8307. LNCS. 2013, pp. 17–32. DOI: 10.1007/978-3-319-03545-1_2.

[DAB09] Derek Dreyer, Amal Ahmed, and Lars Birkedal. "Logical step-indexed logical relations". In: *LICS*. 2009, pp. 71–80. DOI: 10.1109/LICS.2009.34.

[Dan+20] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. "RustBelt meets relaxed memory". In: *PACMPL* 4.POPL (2020), 34:1–34:29. DOI: 10.1145/3371102.

[DD15] Brijesh Dongol and John Derrick. "Verifying linearizability: A comparative survey". In: *ACM Computing Surveys* 48.2 (2015), 19:1–19:43. DOI: 10.1145/2796550.

[Del+17] Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. "Concurrent Data Structures Linked in Time". In: *ECOOP*. Vol. 74. LIPIcs. 2017, 8:1–8:30. DOI: 10.4230/LIPIcs.ECOOP.2017.8.

[DGW10] Thomas Dinsdale-Young, Philippa Gardner, and Mark J. Wheelhouse. "Abstraction and Refinement for Local Reasoning". In: *VSTTE*. Vol. 6217. LNCS. Springer, 2010, pp. 199–215. DOI: 10.1007/978-3-642-15057-9_14.

[Din+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. "Concurrent abstract predicates". In: *ECOOP*. Vol. 6183. LNCS. 2010, pp. 504–528. DOI: 10.1007/978-3-642-14107-2_24.

[Din+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. "Views: Compositional Reasoning for Concurrent Programs". In: *POPL*. 2013, pp. 287–300. DOI: 10.1145/2429069.2429104.

[Din+17] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. "Caper - Automatic Verification for Fine-Grained Concurrency". In: *ESOP*. Vol. 10201. LNCS. 2017, pp. 420–447. DOI: 10.1007/978-3-662-54434-1_16.

[DNB12] Derek Dreyer, Georg Neis, and Lars Birkedal. "The impact of higher-order state and control effects on local relational reasoning". In: *Journal of Functional Programming* 22.4-5 (2012), pp. 477–528. DOI: 10.1017/S095679681200024X.

[Dod+09] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. "Deny-Guarantee Reasoning". In: *ESOP*. Ed. by Giuseppe Castagna. Vol. 5502. LNCS. Springer, 2009, pp. 363–377. DOI: 10.1007/978-3-642-00590-9_26.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. "A Local Shape Analysis Based on Separation Logic". In: *TACAS*. Vol. 3920. LNCS. 2006, pp. 287–302. DOI: 10.1007/11691372_19.

[Dre+10] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. "A relational modal logic for higher-order stateful ADTs". In: *POPL*. 2010, pp. 185–198. DOI: 10.1145/1706299.1706323.

[Dre+19] Derek Dreyer, Amin Timany, Robbert Krebbers, Lars Birkedal, and Ralf Jung. *What Type Soundness Theorem Do You Really Want to Prove?* Oct. 2019. URL: https://blog.sigplan.org/2019/10/17/what-type-soundness-theorem-do-you-really-want-to-prove/.

[DRG18] Thomas Dinsdale-Young, Pedro da Rocha Pinto, and Philippa Gardner. "A Perspective on Specifying and Verifying Concurrent Modules". In: *Journal of Logical and Algebraic Methods in Programming* 98 (Aug. 2018), pp. 1–25. DOI: 10.1016/j.jlamp.2018.03.003.

[EM19] Gidon Ernst and Toby Murray. "SecCSL: Security Concurrent Separation Logic". In: *CAV*. Vol. 11562. LNCS. 2019, pp. 208–230. DOI: 10.1007/978-3-030-25543-5_13.

[ER12] Chucky Ellison and Grigore Rosu. "An Executable Formal Semantics of C with Applications". In: *POPL*. 2012, pp. 533–544. DOI: 10.1145/2103656.2103719.

[Fen+09] Xinyu Feng, Zhong Shao, Yu Guo, and Yuan Dong. "Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads". In: *J. Autom. Reasoning* 42.2-4 (2009), pp. 301–347. DOI: 10.1007/s10817-009-9118-9.

[FGK19a] 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.

[FGK19b] Dan Frumin, Léon Gondelman, and Robbert Krebbers. *Semi-Automated Reasoning About Non-Determinism in C Expressions: Coq Development*. Feb. 2019. URL: https://groupoid.moe/wpc/.

[FH92] Matthias Felleisen and Robert Hieb. "The revised report on the syntactic theories of sequential control and state". In: *Theoretical Computer Science* 103.2 (1992), pp. 235–271. DOI: 10.1016/0304-3975(92)90014-7.

[Fil+10] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. "Abstraction for concurrent objects". In: *Theoretical Computer Science* 411.51-52 (2010), pp. 4379–4398. DOI: 10.1016/j.tcs.2010.09.021.

[FKB18] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency". In: *LICS*. 2018, pp. 442–451. DOI: 10.1145/3209108.3209174.

[FKB20a] Dan Frumin, Robbert Krebbers, and Lars Birkedal. *Coq mechanization of SeLoC*. 2020. URL: https://github.com/co-dan/seloc.

[FKB20b] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity". In: *arXiv e-prints*, arXiv:2006.13635 (June 2020), arXiv:2006.13635. arXiv: 2006.13635 [cs.LO].

[FKB21a] Dan Frumin, Robbert Krebbers, and Lars Birkedal. *Appendix and Coq development of ReLoC*. 2021. URL: https://iris-project.org/reloc/.

[FKB21b] Dan Frumin, Robbert Krebbers, and Lars Birkedal. *Compositional Non-Interference for Fine-Grained Concurrent Programs*. To appear in Security & Privacy 2021. 2021. arXiv: 1910.00905 [cs.LO]. URL: http://arxiv.org/abs/1910.00905.

[Geu09] Herman Geuvers. "Proof assistants: History, ideas and future". In: *Sadhana* 34.1 (2009), pp. 3–25. DOI: 10.1007/s12046-009-0001-5.

[Gia+20] Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. "Scala Step-by-Step: Soundness for DOT with Step-Indexed Logical Relations in Iris". In: *PACMPL* 4.ICFP (2020), 114:1–114:29. DOI: 10.1145/3408996.

[Gon+13] Georges Gonthier et al. "A Machine-Checked Proof of the Odd Order Theorem". In: *ITP*. Vol. 7998. LNCS. Springer, 2013, pp. 163–179. DOI: 10.1007/978-3-642-39634-2_14.

[Gor99] Andrew D. Gordon. "Bisimilarity as a Theory of Functional Programming". In: *Theoretical Computer Science* 228.1-2 (1999), pp. 5–47. DOI: 10.1016/S0304-3975(98)00353-3.

[Got+07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. "Local Reasoning for Storable Locks and Threads". In: *APLAS*. Ed. by Zhong Shao. Vol. 4807. LNCS. Springer, 2007, pp. 19–37. DOI: 10.1007/978-3-540-76637-7_3.

[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*. 2014, pp. 429–439. DOI: 10.1145/2594291.2594296.

[GTA19] Simon Gregersen, Søren Eller Thomsen, and Aslan Askarov. "A Dependently Typed Library for Static Information-Flow Control in Idris". In: *POST*. Vol. 11426. LNCS. 2019, pp. 51–75. DOI: 10.1007/978-3-030-17138-4_3.

[HAN08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. "Oracle Semantics for Concurrent Separation Logic". In: *ESOP*. Ed. by Sophia Drossopoulou. Vol. 4960. LNCS. Springer, 2008, pp. 353–367. DOI: 10.1007/978-3-540-78739-6_27.

[Har16] Robert Harper. *Practical Foundations for Programming Languages (2nd. Ed.)* Cambridge University Press, 2016. URL: https://www.cs.cmu.edu/~rwh/pfpl/.

[HBK20] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. "Actris: Session-type based reasoning in separation logic". In: *PACMPL* 4.POPL (2020), 6:1–6:30. DOI: 10.1145/3371074.

[HD11] Chung-Kil Hur and Derek Dreyer. "A Kripke Logical Relation between ML and Assembly". In: *POPL*. 2011, pp. 133–146. DOI: 10.1145/1926385.1926402.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Rosu. "Defining the Undefinedness of C". In: *PLDI*. 2015, pp. 336–345. DOI: 10.1145/2737924.2737979.

[Hin+21] Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. *Machine-Checked Semantic Session Typing*. To appear at CPP'21. 2021.

[Hoa+11] Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. "Concurrent Kleene algebra and its foundations". In: *The Journal of Logic and Algebraic Programming* 80.6 (2011), pp. 266–296. DOI: 10.1016/j.jlap.2011.04.005.

[HS08] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. "A Scalable Lock-Free Stack Algorithm". In: *SPAA*. 2004, pp. 206–215. DOI: 10.1145/1007912.1007944.

[HW90] Maurice Herlihy and Jeannette Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *TOPLAS* 12.3 (1990), pp. 463–492. DOI: 10.1145/78969.78972.

[IC20] Ninety Nine Percent Invisible and Kevin Caners. Fraktur. Feb. 2020. URL: https://99percentinvisible.org/episode/fraktur/.

[IO01] Samin Ishtiaq and Peter O'Hearn. "BI as an assertion language for mutable data structures". In: *POPL*. 2001, pp. 14–26. DOI: 10.1145/360204.375719.

[Iri20] Iris team. *The Iris Project website and Coq development*. 2020. URL: https://iris-project.org/.

[ISO12] ISO. *ISO/IEC 9899-2011: Programming Languages – C*. ISO Working Group 14, 2012.

[JP11] Bart Jacobs and Frank Piessens. "Expressive modular fine-grained concurrency specification". In: *POPL*. 2011, pp. 271–282. DOI: 10.1145/1926385.1926417.

[JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. "A Quick Tour of the VeriFast Program Verifier". In: *APLAS*. Vol. 6461. LNCS. 2010, pp. 304–311. DOI: 10.1007/978-3-642-17164-2_21.

[JSV10] Patricia Johann, Alex Simpson, and Janis Voigtländer. "A generic operational metatheory for algebraic effects". In: *LICS*. 2010, pp. 209–218. DOI: 10.1109/LICS.2010.29.

[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*. 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*. 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: *PACMPL* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154.

[Jun+18b] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 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: *PACMPL* 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: The promise and the challenge.* To appear in CACM. 2021.

[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*. Ed. by Peter Müller. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 17:1–17:29. DOI: 10.4230/LIPIcs.ECOOP.2017.17.

[Kan+15] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. "A Formal C Memory Model Supporting Integer-Pointer Casts". In: *POPL*. 2015, pp. 326–335. DOI: 10.1145/2737924.2738005.

[Kar+18] Aleksandr Karbyshev, Kasper Svendsen, Aslan Askarov, and Lars Birkedal. "Compositional Non-interference for Concurrent Programs via Separation and Framing". In: *POST*. Vol. 10804. LNCS. 2018, pp. 53–78. DOI: 10.1007/978-3-319-89722-6_3.

[Khy+17] Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew Parkinson. "Proving linearizability using partial orders". In: *ESOP*. Vol. 10201. LNCS. 2017, pp. 639–667. DOI: 10.1007/978-3-662-54434-1_24.

[Knu02] Donald Knuth. "All Questions Answered". In: *Notices of the AMS* 49.3 (2002), pp. 318–324.

[Koz94] Dexter Kozen. "A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events". In: *Information and Computation* 110.2 (1994), pp. 366–390. DOI: 10.1006/inco.1994.1037.

[Kre+17] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. "The essence of higher-order concurrent separation logic". In: *ESOP*. Vol. 10201. LNCS. 2017, pp. 696–723. DOI: 10.1007/978-3-662-54434-1_26.

[Kre+18] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. "MoSeL: A general, extensible modal framework for interactive proofs in separation logic". In: *PACMPL* 2.ICFP (2018), 77:1–77:30. DOI: 10.1145/3236772.

[Kre13] Robbert Krebbers. "Aliasing Restrictions of C11 Formalized in Coq". In: *CPP*. Vol. 8307. LNCS. 2013, pp. 50–65. DOI: 10.1007/978-3-319-03545-1_4.

[Kre14] Robbert Krebbers. "An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C". In: *POPL*. 2014, pp. 101–112. DOI: 10.1145/2535838.2535878.

[Kre15] Robbert Krebbers. "The C Standard Formalized in Coq". PhD thesis. Radboud University Nijmegen, 2015.

[Kre16] Robbert Krebbers. "A Formal C Memory Model for Separation Logic". In: *JAR* 57.4 (2016), pp. 319–387. DOI: 10.1007/s10817-016-9369-1.

[KSB17] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. "A Relational Model of Types-and-Effects in Higher-Order Concurrent Separation Logic". In: *POPL*. 2017, pp. 218–231. DOI: 10.1145/3009837.3009877.

[KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive proofs in higher-order concurrent separation logic". In: *POPL*. 2017, pp. 205–217. DOI: 10.1145/3009837.3009855.

[KW06] Vasileios Koutavas and Mitchell Wand. "Small bisimulations for reasoning about higher-order imperative programs". In: *POPL*. 2006, pp. 141–152. DOI: 10.1145/1111037.1111050.

[KW13] Robbert Krebbers and Freek Wiedijk. "Separation Logic for Non-local Control Flow and Block Scope Variables". In: *FoSSaCS*. Vol. 7794. LNCS. 2013, pp. 257–272. DOI: 10.1007/978-3-642-37075-5_17.

[Lah+17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. "Repairing Sequential Consistency in C/C++11". In: *PLDI*. 2017, pp. 618–632. DOI: 10.1145/3062341.3062352.

[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.

[LC15] Luísa Lourenço and Luís Caires. "Dependent Information Flow Types". In: *POPL*. 2015, pp. 317–328. DOI: 10.1145/2676726.2676994.

[Ler09] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *CACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814.

[LF13] Hongjin Liang and Xinyu Feng. "Modular verification of linearizability with non-fixed linearization points". In: *PLDI*. 2013, pp. 459–470. DOI: 10.1145/2491956.2462189.

[Liu+09] Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. "Model Checking Linearizability via Refinement". In: *FM*. Ed. by Ana Cavalcanti and Dennis R. Dams. Vol. 5850. LNCS. 2009, pp. 321–337. DOI: 10.1007/978-3-642-05089-3_21.

[Mal14] Gregory Malecha. "Extensible Proof Engineering in Intensional Type Theory". PhD thesis. Harvard University, 2014.

[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*. 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: *PACMPL* 3.POPL (2019), 67:1–67:32. DOI: 10.1145/3290380.

[Mit86] John Mitchell. "Representation Independence and Data Abstraction". In: *POPL*. 1986, pp. 263–276. DOI: 10.1145/512644.512669.

[MJP19] Glen Mével, Jacques-Henri Jourdan, and François Pottier. "Time Credits and Time Receipts in Iris". In: *ESOP*. Vol. 11423. LNCS. Springer, 2019, pp. 3–29. DOI: 10.1007/978-3-030-17184-1_1.

[MM11] Yannick Moy and Claude Marché. *The Jessie Plugin for Deduction Verification in Frama-C, Tutorial and Reference Manual*. 2011.

[MS91] John M. Mellor-Crummey and Michael L. Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors". In: *TOCS* 9.1 (1991), pp. 21–65. DOI: 10.1145/103727.103729.

[MS96] Maged Michael and Michael Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In: *PODC*. 1996, pp. 267–275. DOI: 10.1145/248052.248106.

[MSE18] Toby C. Murray, Robert Sison, and Kai Engelhardt. "COVERN: A Logic for Compositional Verification of Information Flow Control". In: *EuroS&P*. 2018, pp. 16–30. DOI: 10.1109/EuroSP.2018.00010.

[MSS11] Heiko Mantel, David Sands, and Henning Sudbrock. "Assumptions and Guarantees for Compositional Noninterference". In: *CSF*. 2011, pp. 218–232. DOI: 10.1109/CSF.2011.22.

[Mur+16] Toby Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. "Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference". In: *CSF*. 2016, pp. 417–431. DOI: 10.1109/CSF.2016.36.

[Nak00] Hiroshi Nakano. "A Modality for Recursion". In: *LICS*. 2000, pp. 255–266. DOI: 10.1109/LICS.2000.855774.

[NBG13] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. "Dependent type theory for verification of information flow and access control policies". In: *TOPLAS* 35.2 (2013), 6:1–6:41. DOI: 10.1145/2491522.2491523.

[NBN19] Mohammad Nikouei, Anindya Banerjee, and David A. Naumann. "Data Abstraction and Relational Program Logic". In: *arXiv e-prints*, arXiv:1910.14560 (Oct. 2019), arXiv:1910.14560. arXiv: 1910.14560 [cs.LO].

[NDR11] Georg Neis, Derek Dreyer, and Andreas Rossberg. "Non-Parametric Parametricity". In: *Journal of Functional Programming* 21.4-5 (2011), pp. 497–562. DOI: 10.1017/S0956796811000165.

[NMS16] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. "An Operational Semantics for C/C++11 Concurrency". In: *OOPSLA*. 2016, pp. 111–128. DOI: 10.1145/2983990.2983997.

[Nor98] Michael Norrish. "C Formalised in HOL". PhD thesis. University of Cambridge, 1998.

[Nor99] Michael Norrish. "Deterministic Expressions in C". In: *ESOP*. Vol. 1576. LNCS. 1999, pp. 147–161. DOI: 10.1007/3-540-49099-X_10.

[OHe07] Peter O'Hearn. "Resources, Concurrency, and Local Reasoning". In: *Theoretical Computer Science* 375.1-3 (2007), pp. 271–307. DOI: 10.1016/j.tcs.2006.12.035.

[OHe19] Peter O'Hearn. "Separation logic". In: *CACM* 62.2 (2019), pp. 86–95. DOI: 10.1145/3211968.

[OP99] Peter O'Hearn and David Pym. "The logic of bunched implications". In: *Bulletin of Symbolic Logic* 5.2 (1999), pp. 215–244. DOI: 10.2307/421090.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *CSL*. Vol. 2142. LNCS. 2001, pp. 1–19. DOI: 10.1007/3-540-44802-0_1.

[PA93] Gordon Plotkin and Martín Abadi. "A logic for parametric polymorphism". In: *TLCA*. Vol. 664. LNCS. 1993, pp. 361–375. DOI: 10.1007/BFb0037118.

[PB05] Matthew J. Parkinson and Gavin M. Bierman. "Separation logic and abstraction". In: *POPL*. ACM, 2005, pp. 247–258. DOI: 10.1145/1040305.1040326.

[PBO07] Matthew Parkinson, Richard Bornat, and Peter O'Hearn. "Modular verification of a non-blocking stack". In: *POPL*. Ed. by Martin Hofmann and Matthias Felleisen. ACM, 2007, pp. 297–302. DOI: 10.1145/1190216.1190261.

[Pit00] Andrew M. Pitts. "Operational Semantics and Program Equivalence". In: *APPSEM*. Vol. 2395. LNCS. 2000, pp. 378–412. DOI: 10.1007/3-540-45699-6_8.

[Pit05] Andrew M. Pitts. "Typed Operational Reasoning". In: *Advanced Topics in Types and Programming Languages*. Ed. by Benjamin C. Pierce. MIT Press, 2005. Chap. 7, pp. 245–289.

[Plo76] Gordon Plotkin. "A powerdomain construction". In: *SIAM Journal on Computing* 5.3 (1976), pp. 452–487.

[PS03] François Pottier and Vincent Simonet. "Information flow inference for ML". In: *TOPLAS* 25.1 (2003), pp. 117–158. DOI: 10.1145/596980.596983.

[PS98] Andrew Pitts and Ian Stark. "Operational Reasoning for Functions with Local State". In: *Higher Order Operational Techniques in Semantics*. New York, NY, USA: Cambridge University Press, 1998, pp. 227–274. URL: http://dl.acm.org/citation.cfm?id=309656.309671.

[Rad+18] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. "Monadic refinements for relational cost analysis". In: *PACMPL* 2.POPL (2018), 36:1–36:32. DOI: 10.1145/3158124.

[RDG14] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. "TaDA: A logic for time and data abstraction". In: *ECOOP*. Vol. 8586. LNCS. 2014, pp. 207–231. DOI: 10.1007/978-3-662-44202-9_9.

[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.

[Rey74] John C. Reynolds. "Towards a theory of type structure". In: *Programming Symposium, Proceedings Colloque sur la Programmation, Paris*. Vol. 19. LNCS. 1974, pp. 408–423.

[RG18] Vineet Rajani and Deepak Garg. "Types for Information Flow Control: Labeling Granularity and Semantic Models". In: *CSF*. 2018, pp. 233–246. DOI: 10.1109/CSF.2018.00024.

[Rin+19] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. "QED at Large: A Survey of Engineering of Formally Verified Software". In: *Found. Trends Program. Lang.* 5.2-3 (2019), pp. 102–281. DOI: 10.1561/2500000045.

[Roc17] Pedro da Rocha Pinto. "Reasoning with Time and Data Abstractions". PhD thesis. Imperial College London, June 2017.

[Sam+20a] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. "The high-level benefits of low-level sandboxing". In: *PACMPL* 4.POPL (2020), 32:1–32:32. DOI: 10.1145/3371100.

[Sam+20b] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. *RefinedC: An Extensible Refinement Type System for C Based on Separation Logic Programming*. In submssion. 2020. URL: https://plv.mpi-sws.org/refinedc/.

[SB14] Kasper Svendsen and Lars Birkedal. "Impredicative concurrent abstract predicates". In: *ESOP*. Vol. 8410. LNCS. 2014, pp. 149–168. DOI: 10.1007/978-3-642-54833-8_9.

[SBP13] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. "Modular Reasoning about Separation of Concurrent Data Structures". In: *ESOP*. Vol. 7792. LNCS. 2013, pp. 169–188. DOI: 10.1007/978-3-642-37036-6_11.

[SGD17] David Swasey, Deepak Garg, and Derek Dreyer. "Robust and Compositional Verification of Object Capability Patterns". In: *PACMPL* 1.OOPSLA (2017), 89:1–89:26. DOI: 10.1145/3133913.

[SM03] Andrei Sabelfeld and Andrew C. Myers. "A Model for Delimited Information Release". In: *ISSS*. Vol. 3233. LNCS. 2003, pp. 174–191. DOI: 10.1007/978-3-540-37621-7_9.

[SM19] Robert Sison and Toby Murray. "Verifying That a Compiler Preserves Concurrent Value-Dependent Information-Flow Security". In: *ITP*. Vol. 141. LIPIcs. 2019, 27:1–27:19. DOI: 10.4230/LIPIcs.ITP.2019.27.

[SMS20] Daniel Schoepe, Toby Murray, and Andrei Sabelfeld. "VERONICA: Expressive and Precise Concurrent Information Flow Security (Extended Version with Technical Appendices)". In: *arXiv e-prints*, arXiv:2001.11142 (Jan. 2020), arXiv:2001.11142. arXiv: 2001.11142 [cs.LO].

[Smy76] Michael Smyth. "Powerdomains". In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 1976, pp. 537–543.

[SP07] Eijiro Sumii and Benjamin C. Pierce. "A bisimulation for type abstraction and recursion". In: *JACM* 54.5 (2007), p. 26. DOI: 10.1145/1284320.1284325.

[SS00] Andrei Sabelfeld and David Sands. "Probabilistic Noninterference for Multi-Threaded Programs". In: *CSFW*. 2000, pp. 200–214. DOI: 10.1109/CSFW.2000.856937.

[SS09] Andrei Sabelfeld and David Sands. "Declassification: Dimensions and principles". In: *JCS* 17.5 (2009), pp. 517–548. DOI: 10.3233/JCS-2009-0352.

[Ste+15] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. "Compositional CompCert". In: *POPL*. 2015, pp. 275–287. DOI: 10.1145/2676726.2676985.

[STS15] Steven Schäfer, Tobias Tebbi, and Gert Smolka. "Autosubst: Reasoning with de Bruijn terms and parallel substitutions". In: *ITP*. Vol. 9236. LNCS. 2015, pp. 359–374. DOI: 10.1007/978-3-319-22102-1_24.

[SV11] Bas Spitters and Eelis Van der Weegen. "Type Classes for Mathematics in Type Theory". In: *Mathematical Structures in Computer Science* 21.4 (2011), pp. 795–825. DOI: 10.1017/S0960129511000119.

[SV20] Alex Simpson and Niels Voorneveld. "Behavioural Equivalence via Modalities for Algebraic Effects". In: *TOPLAS* 42.1 (2020), 4:1–4:45. DOI: 10.1145/3363518.

[TB19] Amin Timany and Lars Birkedal. "Mechanized Relational Verification of Concurrent Programs with Continuations". In: *PACMPL* 3.ICFP (2019), 105:1–105:28. DOI: 10.1145/3341709.

[TDB13] Aaron Turon, Derek Dreyer, and Lars Birkedal. "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency". In: *ICFP*. 2013, pp. 377–390. DOI: 10.1145/2500365.2500600.

[Ter08] Tachio Terauchi. "A Type System for Observational Determinism". In: *CSF*. 2008, pp. 287–300. DOI: 10.1109/CSF.2008.9.

[Tim+18] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. "A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST". In: *PACMPL* 2.POPL (2018), 64:1–64:28. DOI: 10.1145/3158152.

[Tim18] Amin Timany. "Contributions in Programming Langues Theory: Logical Relations and Type Theory". PhD thesis. KU Leuven, 2018.

[TJH17] Joseph Tassarotti, Ralf Jung, and Robert Harper. "A Higher-Order Logic for Concurrent Termination-Preserving Refinement". In: *ESOP*. Vol. 10201. LNCS. 2017, pp. 909–936. DOI: 10.1007/978-3-662-54434-1_34.

[Tre86] R. Kent Treiber. *Systems Programming: Coping With Parallelism*. Tech. rep. Thomas J. Watson Research Center, 1986.

[Tur+13] Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. "Logical relations for fine-grained concurrency". In: *POPL*. 2013, pp. 343–356. DOI: 10.1145/2429069.2429111.

[Vaf08] Viktor Vafeiadis. "Modular fine-grained concurrency verification". PhD thesis. University of Cambridge, 2008.

[Vaf09] Viktor Vafeiadis. "Shape-Value Abstraction for Verifying Linearizability". In: *VMCI*. Vol. 5403. LNCS. 2009, pp. 335–348. DOI: 10.1007/978-3-540-93900-9_27.

[VB21] Simon Friis Vindum and Lars Birkedal. *Contextual Refinement of the Michael-Scott Queue (Proof Pearl)*. To appear at CPP'21. 2021.

[VYY09] Martin Vechev, Eran Yahav, and Greta Yorsh. "Experience with Model Checking Linearizability". In: *SPIN*. Vol. 5578. LNCS. 2009, pp. 261–278. DOI: 10.1007/978-3-642-02652-2_21.

[Yan07] Hongseok Yang. "Relational separation logic". In: *Theoretical Computer Science* 375.1-3 (2007), pp. 308–334. DOI: 10.1016/j.tcs.2006.12.036.

[Zda02] Stephan A. Zdancewic. "Programming Languages for Information Security". PhD thesis. Cornell University, 2002.

[ZM03] Steve Zdancewic and Andrew C. Myers. “Observational Determinism for Concurrent Program Security”. In: *CSF*. 2003, p. 29. DOI: 10.1109/CSFW.2003.1212703.

[ZM07] Lantian Zheng and Andrew C. Myers. “Dynamic Security Labels and Static Information Flow Control”. In: *International Journal of Information Security* 6.2-3 (2007), pp. 67–84. DOI: 10.1007/s10207-007-0019-9.

Summary

Verification of concurrent programs is known to be a challenging task, in particular due to the intricate interactions that can be exhibited between the components of a concurrent system. In this thesis we develop program logics aimed at verifying properties of concurrent programs. Our approach is based on *concurrent separation logic*, which is a widely employed family of program logics for reasoning about stateful concurrent programs. We use the lens of concurrent separation logic to study three program properties.

Firstly, we look at *safety*: is a program free of run-time errors and undefined behavior? More specifically, a safe program does not dereference dangling pointers and does not exhibit problematic data races. Secondly, we consider *refinement* of programs: can one program be substituted for another one? Formally, if the first program refines the second one, then the set of observable behaviors of the first program is a subset of the observable behaviors of the second program. Lastly, we look at *security*: does a program leak sensitive information? Under the formulation of non-interference, a program is secure if it behaves the same way under different sensitive inputs.

For each of the properties we develop a concurrent separation logic. In order to study these properties and to construct appropriate logics, we use a methodology that informs and guides the design of the verification methods. Firstly, the logics that we develop support local and compositional reasoning, which enables us to construct robust and reusable proofs about program modules that can be combined together into a proof about a program as a whole. Secondly, the program logics are connected to clearly stated properties via their soundness theorems. A soundness theorem states that if a proof about the whole program can be derived in the logic, then the program satisfies the desired property. The property itself is formulated directly against the operational semantics, without referencing the logic. Finally, the logics themselves, together with the soundness theorems, are fully mechanized in the Coq proof assistant using the Iris framework. The mechanizations in Coq are constructed in such a way that proofs about specific programs *inside the logic* (as opposed to proofs *about* the logic) can be carried out interactively by the user in Coq. We demonstrate the viability of the approach taken in this thesis by exercising our logics on a number of examples and case studies.

Samenvatting

Verificatie van parallelle programma's is een uitdagende opgave, in het bijzonder vanwege de ingewikkelde interacties die tussen de verschillende onderdelen van een parallel (*concurrent*) systeem kunnen optreden. In dit proefschrift ontwikkelen wij programmalogica's om eigenschappen van parallelle programma's te bewijzen. Onze aanpak is gebaseerd op *concurrent separation logic*, een veelgebruikte familie van programmalogica's voor het redeneren over parallelle programma's. Wij gebruiken inzichten van *concurrent separation logic* om drie eigenschappen van programma's te bestuderen.

Ten eerste kijken wij naar *betrouwbaarheid* (*safety*): kunnen er geen fouten optreden tijdens het uitvoeren van een programma (*undefined behavior*)? In het bijzonder maakt een betrouwbaar programma geen gebruik van *dangling pointers* en heeft het geen problematische *data races*. Ten tweede onderzoeken wij *verfijning* (*refinement*) van programma's: kan een programma door een ander vervangen worden? Formeel gesproken, als het eerste programma het tweede programma verfijnt, dan is de verzameling van waarneembare gedragingen van het eerste programma een deelverzameling van de verzameling van waarneembare gedragingen van het tweede programma. Ten slotte onderzoeken wij *veiligheid* (*security*): kan een programma vertrouwelijke informatie lekken? Volgens het formalisme van *non-interference* is een programma veilig indien het gedrag hetzelfde is bij verschillende vertrouwelijke invoer.

Voor elke eigenschap ontwikkelen wij een *concurrent separation logic*. Om deze eigenschappen te onderzoeken en om deze logica's te construeren, volgen wij een specifieke methodologie die het ontwerp van de programmalogica's stuurt. Ten eerste, de logica's ondersteunen lokaal en compositioneel redeneren. Dat stelt ons in staat om robuuste en herbruikbare bewijzen over programma modules te maken, en deze bewijzen samen te stellen tot een bewijs over het hele programma. Ten tweede, elke logica heeft een correctheidsstelling die garandeert dat een bewijs in de logica de beoogde eigenschap van een programma impliceert, uitgedrukt middels de operationele semantiek van de programmeertaal. Ten slotte zijn de ontwikkelde logica's en hun correctheidsstellingen volledige geformaliseerd in het Coq bewijssysteem. Hiervoor maken wij gebruik van het Iris framework voor hogere-orde concurrent separation logic. De formalisaties in Coq zijn zo ontwikkeld dat bewijzen over concrete programma's *binnen de logica* (in tegenstelling tot bewijzen *over* de logica zelf) interactief met Coq gemaakt kunnen worden. Wij demonstreren de schaalbaarheid van onze aanpak aan de hand van een scala aan voorbeelden en casestudy's.

Titles in the IPA Dissertation Series since 2018

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems*. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces*. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network*. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems*. Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations*. Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries*. Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

A. Serrano Mena. *Type Error Customization for Embedded Domain-Specific Languages*. Faculty of Science, UU. 2018-21

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow*. Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning*. Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management*. Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers*. Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems*. Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities*. Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis*. Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Griffioen. *A Unit-Aware Matrix Language and its Application in Control and Auditing*. Faculty of Science, UvA. 2019-11

A.A. Sawant. *The impact of API evolution on API consumers and how this can be affected by API producers and language designers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12

W.H.M. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views*. Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Co-simulation Construction*. Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming*. Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models*. Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems*. Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing*. Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations*. Faculty of Mathematics and Computer Science, TU/e. 2020-09

D.Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

Research Data Management

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of Radboud University, The Netherlands.[13]

The following research datasets have been produced during this PhD research:

- Chapter 2: Dan Frumin (2020): Background on separation logic, Coq code in a Git repository. https://github.com/co-dan/thesis/tree/master/prelim
- Chapter 3: Dan Frumin, Léon Gondelman, Robbert Krebbers (2019): λMC: a monadic translation of mini C into Iris's HeapLang, Coq code in a Git repository. https://gitlab.mpi-sws.org/iris/c/
- Chapter 4: Dan Frumin, Robbert Krebbers, Lars Birkedal (2020), ReLoC: a logic for proving contextual refinements, Coq code in a Git repository. https://gitlab.mpi-sws.org/iris/reloc/
- Chapter 5: Dan Frumin, Robbert Krebbers, Lars Birkedal (2020), SeLoC: a logic for proving non-interference, Coq code in a Git repository. https://github.com/co-dan/SeLoC

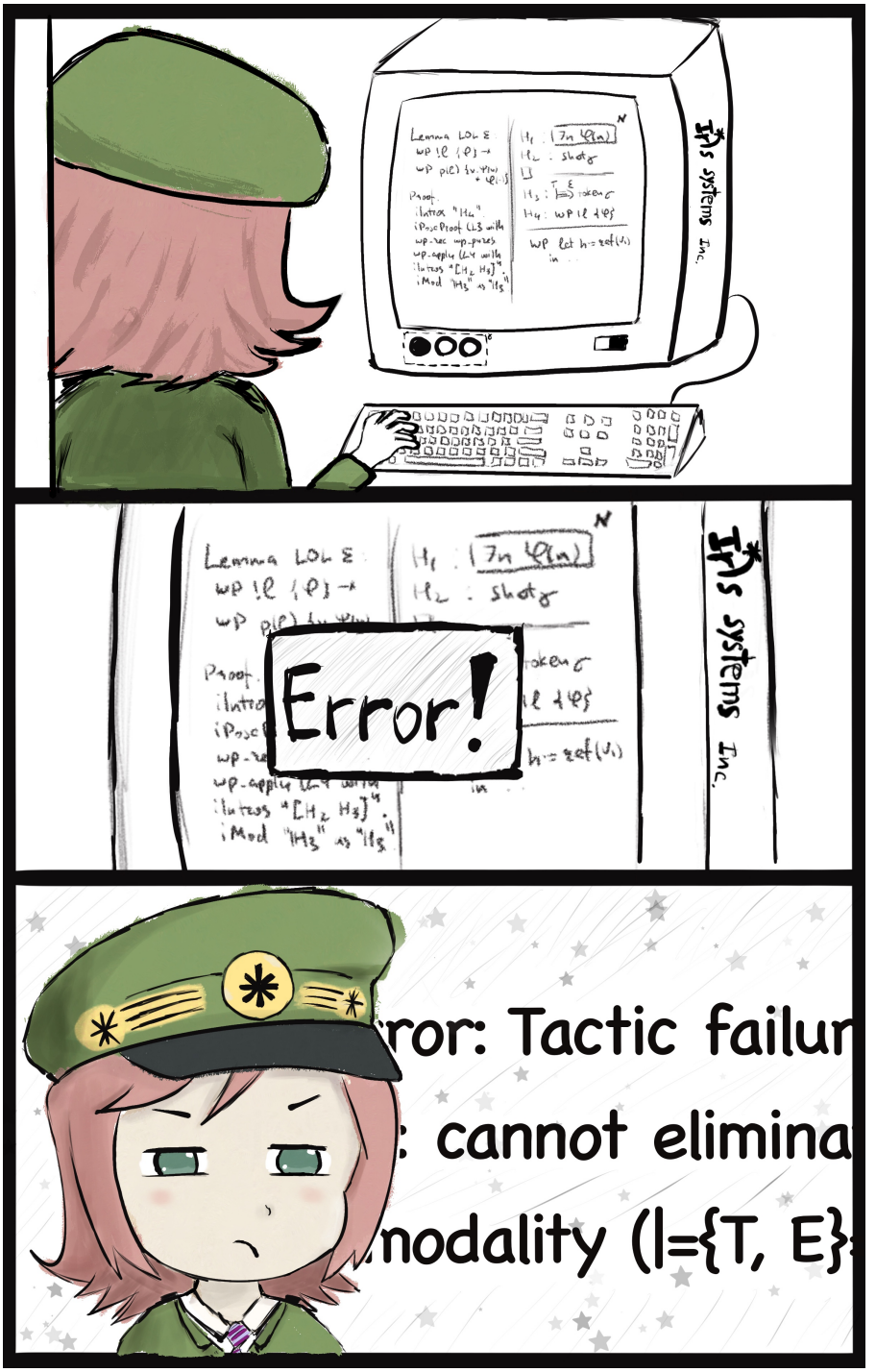
Additionally, an archive with all the Coq formalizations, with versions corresponding to the ones presented in this thesis, is available at https://github.com/co-dan/thesis and https://doi.org/10.5281/zenodo.4445839.

[13] ru.nl/icis/research-data-management/, last accessed January 20th, 2021.

About the author

Dan Frumin was born in 1993 in Krasnojarsk, Russia. Before starting school, he moved with his family to Moscow. In 2014 he graduated from the Higher School of Economics with a bachelor degree in computer science. In 2016 he obtained the degree of Master of Logic *cum laude* at the University of Amsterdam. Afterwards, in the same year, he started his PhD at the Radboud University under the supervision of Herman Geuvers and Freek Wiedijk. Shortly after, Robbert Krebbers also became his official supervisor.

As of September 2020, Dan works as a postdoctoral researcher in the group of Jorge Pérez at the University of Groningen. Dan had his PhD defense in March 2021. He lives happily ever after.[citation needed]


Iris systems Inc.
Error!
Iris systems Inc.
ror: Tactic failur
: cannot elimina
nodality (|={T, E}